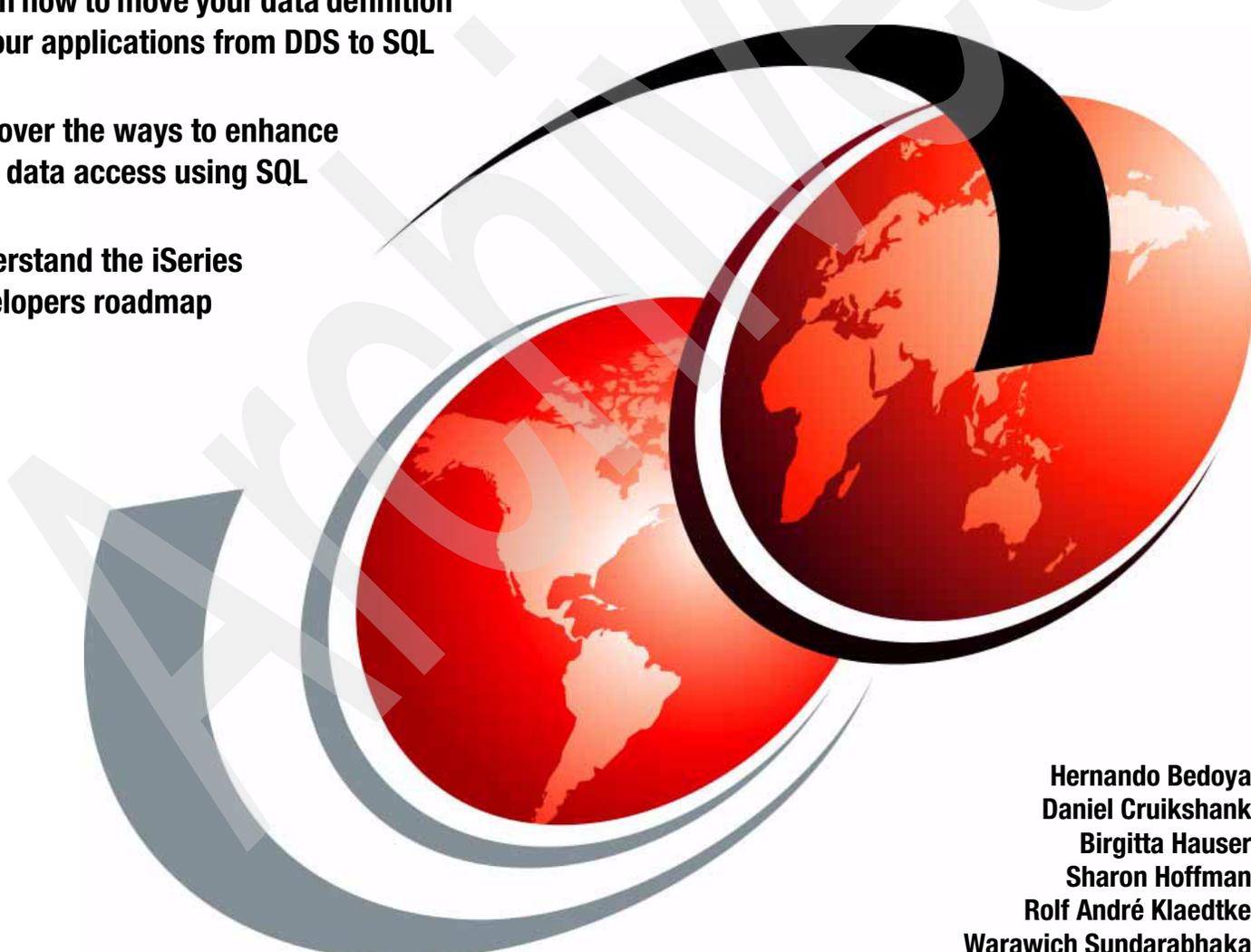


# Modernizing IBM *e*server iSeries Application Data Access - A Roadmap Cornerstone

Learn how to move your data definition  
of your applications from DDS to SQL

Discover the ways to enhance  
your data access using SQL

Understand the iSeries  
developers roadmap



Hernando Bedoya  
Daniel Cruikshank  
Birgitta Hauser  
Sharon Hoffman  
Rolf André Klaedtke  
Warawich Sundarabhaka

**Redbooks**





International Technical Support Organization

**Modernizing IBM @server iSeries Application Data  
Access - A Roadmap Cornerstone**

February 2005

Archived

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

Archived

**First Edition (February 2005)**

This edition applies to Version 5, Release 3, Modification 0 of i5/OS, Program Number 5722-SS1.

© Copyright International Business Machines Corporation 2005. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Notices</b> .....	vii
Trademarks .....	viii
<b>Preface</b> .....	ix
The team that wrote this redbook .....	ix
Become a published author .....	xi
Comments welcome .....	xi
<b>Part 1. Introduction and background</b> .....	1
<b>Chapter 1. iSeries Developer Roadmap - The big picture</b> .....	1
1.1 Introduction to the iSeries Developer Roadmap .....	2
1.1.1 Why a roadmap .....	2
1.1.2 Why care about it .....	3
1.1.3 The goal .....	4
1.2 What is in the roadmap .....	4
1.2.1 Better tools .....	5
1.2.2 Better user interface .....	6
1.2.3 Better architecture .....	7
1.2.4 Better portability .....	8
1.2.5 Better scalability .....	8
1.3 What is in this book .....	9
<b>Chapter 2. Why modernize with SQL and DB2 UDB for iSeries</b> .....	11
2.1 Background .....	12
2.1.1 A short look at the history of SQL .....	12
2.1.2 The main parts of SQL .....	12
2.2 Reasons to modernize .....	12
2.2.1 Standard compliancy .....	12
2.2.2 Openness .....	13
2.2.3 Performance .....	13
2.2.4 Available skills .....	15
2.2.5 Functionality .....	16
2.2.6 Data integrity .....	17
<b>Part 2. Data definition</b> .....	19
<b>Chapter 3. Approaches and options</b> .....	21
3.1 Data definition considerations .....	22
3.2 Accessing the database data .....	23
3.2.1 Native record level access .....	23
3.2.2 Data access with SQL .....	23
3.3 Methodology for the modernization .....	25
3.3.1 Reverse engineering DDS to SQL DDL (stage 1) .....	25
3.3.2 Creating I/O modules to access DB data (stage 2) .....	26
3.3.3 Moving business rules into the database (stage 3) .....	26
3.3.4 Externalizing data access (stage 4) .....	26
<b>Chapter 4. Modernizing database definitions</b> .....	29
4.1 Reverse engineering DDS to SQL DDL .....	30

4.1.1	Classify the existing environment . . . . .	30
4.1.2	Establishing a list of all DDS files to be converted . . . . .	31
4.1.3	Establishing naming conventions for SQL objects . . . . .	32
4.1.4	Converting the DDS to SQL DDL . . . . .	34
4.1.5	Reviewing the generated SQL DDL . . . . .	36
4.1.6	Creating the new DB2 schema on the iSeries server . . . . .	40
4.1.7	Create all existing DDS logical files over the new SQL tables . . . . .	43
4.1.8	Migrate data and test existing programs . . . . .	46
4.2	Comparing the SQL objects and the DDS files . . . . .	48
4.2.1	SQL tables compared with physical files . . . . .	48
4.2.2	SQL indexes compared with keyed logical files . . . . .	48
4.2.3	SQL views compared with logical files . . . . .	49
4.2.4	SQL data types . . . . .	49
4.3	SQL system catalogs: Definitions . . . . .	50
4.3.1	SQL system catalogs: Example . . . . .	52
4.4	Partitioned tables . . . . .	53
<b>Part 3. Data access</b>	. . . . .	<b>55</b>
<b>Chapter 5. Creating I/O modules to access SQL objects</b>	. . . . .	<b>57</b>
5.1	Introduction . . . . .	58
5.2	Establish naming conventions . . . . .	58
5.3	Create SQL views based on business requirements . . . . .	59
5.4	Create service programs to access data from the SQL views . . . . .	62
5.5	Convert legacy programs to use service programs . . . . .	63
<b>Chapter 6. Moving business rules to the database</b>	. . . . .	<b>65</b>
6.1	Database normalization . . . . .	66
6.2	Referential integrity . . . . .	67
6.3	Constraints . . . . .	67
6.4	Constraint coexistence considerations . . . . .	68
6.5	Column-level security . . . . .	69
6.6	Column encryption . . . . .	69
6.7	Automatic key generation and unique identifiers . . . . .	70
6.8	Accessing non-relational data . . . . .	72
6.8.1	User defined table functions for accessing non-relational data . . . . .	72
6.8.2	Datalink . . . . .	74
6.8.3	Large Object Support . . . . .	75
<b>Chapter 7. Embedded SQL</b>	. . . . .	<b>77</b>
7.1	How to get started . . . . .	78
7.2	Creating a SQLRPG - Program/service program/module . . . . .	79
7.3	Compile command CRTSQLRPGI . . . . .	80
7.3.1	Missing compile options in the SQL compile command . . . . .	80
7.3.2	Important compile options for SQL statements . . . . .	81
7.3.3	SET OPTION statement . . . . .	82
7.4	Error handling - SQLCA (SQL communications area) . . . . .	82
7.4.1	SQLCODE . . . . .	84
7.4.2	SQLSTATE . . . . .	85
7.5	Host variables . . . . .	86
7.5.1	Single field host variable . . . . .	86
7.5.2	Host structure . . . . .	89
7.5.3	Host structure array . . . . .	89
7.5.4	Naming considerations for host variables . . . . .	91

7.6 Exploiting SQL scalar functions in RPG . . . . .	91
7.7 Static SQL without cursor . . . . .	94
7.7.1 Static SQL returning a single row . . . . .	94
7.7.2 Processing non-Select statements with static SQL without cursor . . . . .	95
7.8 Using a cursor . . . . .	98
7.8.1 The DECLARE statement . . . . .	99
7.8.2 The OPEN statement . . . . .	103
7.8.3 The FETCH statement . . . . .	104
7.8.4 Types of cursors . . . . .	107
7.8.5 Updating or deleting rows using a cursor . . . . .	108
7.9 Dynamic SQL . . . . .	109
7.9.1 Defining the character string containing the SQL statement . . . . .	110
7.9.2 The EXECUTE IMMEDIATE statement . . . . .	110
7.9.3 Combining the SQL statements PREPARE and EXECUTE . . . . .	111
7.9.4 Combining the SQL statements PREPARE and DECLARE . . . . .	113
7.9.5 The SQL descriptor area . . . . .	114
<b>Chapter 8. Externalizing data access . . . . .</b>	<b>117</b>
8.1 Trigger programs . . . . .	118
8.1.1 Activation time of trigger programs . . . . .	119
8.1.2 Trigger events . . . . .	121
8.1.3 External triggers . . . . .	122
8.1.4 SQL triggers . . . . .	131
8.1.5 Getting information about triggers . . . . .	138
8.2 Stored procedures . . . . .	138
8.2.1 External stored procedures . . . . .	139
8.2.2 SQL stored procedures . . . . .	143
8.2.3 SQL statement CREATE PROCEDURE . . . . .	144
8.2.4 Procedure signature and overloading . . . . .	148
8.2.5 Deleting or replacing a stored procedure . . . . .	151
8.2.6 Getting information about stored procedures . . . . .	152
8.3 User defined functions . . . . .	152
8.3.1 External user defined functions . . . . .	154
8.3.2 SQL user defined scalar functions . . . . .	159
8.3.3 User defined table functions . . . . .	160
8.3.4 User defined function signature and overloading . . . . .	160
8.3.5 Deleting or replacing a user defined function . . . . .	162
8.3.6 Getting information about user defined functions . . . . .	163
8.4 SQL programming language . . . . .	163
8.4.1 Compound statement . . . . .	164
8.4.2 Control statements . . . . .	164
8.4.3 Error handling in SQL . . . . .	167
<b>Chapter 9. Other considerations . . . . .</b>	<b>171</b>
9.1 Comparing RPG and SQL data types . . . . .	172
9.1.1 Character data types . . . . .	173
9.1.2 Character fields with fixed and varying length . . . . .	176
9.1.3 Numeric data types . . . . .	178
9.1.4 Date and time data types . . . . .	185
9.2 NULL values . . . . .	190
9.2.1 Handling NULL values in RPG with native I/O . . . . .	191
9.2.2 Using indicator variables in SQL . . . . .	193
9.2.3 Particular characteristics of NULL values in SQL statements . . . . .	195

9.3 Date and time calculation . . . . .	196
9.3.1 Converting from numeric/character date values to real date values . . . . .	197
9.3.2 Converting from date fields to character or numeric representation . . . . .	202
9.3.3 Checking for a valid date or time . . . . .	205
9.3.4 Retrieving current date and time . . . . .	207
9.3.5 Adding and subtracting date and time values . . . . .	210
9.3.6 Calculating date and time differences . . . . .	213
9.3.7 Extracting a portion of a date, time, or timestamp . . . . .	219
9.3.8 Additional SQL scalar functions for date calculation . . . . .	220
<b>Part 4. Tools</b> . . . . .	<b>223</b>
<b>Chapter 10. DB2 Development Tools</b> . . . . .	<b>225</b>
10.1 WebSphere Development Studio Client for iSeries (WDSC) . . . . .	226
10.2 iSeries Navigator . . . . .	234
10.2.1 Database Navigator . . . . .	234
10.2.2 Run SQL Scripts . . . . .	235
10.2.3 Visual Explain . . . . .	235
10.2.4 Graphical iSeries System Debugger . . . . .	238
10.3 OS/400 utilities . . . . .	243
10.4 DB2 Development Center . . . . .	243
10.5 DB2 Query Management Facility . . . . .	244
10.5.1 Migrating existing queries . . . . .	244
10.5.2 Creating new queries . . . . .	248
<b>Related publications</b> . . . . .	<b>255</b>
IBM Redbooks . . . . .	255
Other publications . . . . .	255
Online resources . . . . .	256
How to get IBM Redbooks . . . . .	257
Help from IBM . . . . .	257
<b>Index</b> . . . . .	<b>259</b>

# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive Armonk, NY 10504-1785 U.S.A.*

*The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law.* INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

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

## COPYRIGHT LICENSE:

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

## Trademarks

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

AS/400®	Integrated Language Environment®	RPG/400®
DB2 Universal Database™	IBM®	System/38™
DB2®	Language Environment®	SQL/400®
DRDA®	Net.Data®	VisualAge®
@server®	OS/390®	WebSphere®
@server®	OS/400®	z/OS®
ibm.com®	QMF™	zSeries®
iSeries™	Redbooks™	
i5/OS™	Redbooks (logo)  ™	

The following terms are trademarks of other companies:

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

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

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

# Preface

In 1978 IBM® introduced the System/38™ as part of its midrange platform hardware base. One of the many outstanding features of this system was the built-in Relational Database Management System (RDMS) support. The system included a utility for defining databases, screens, and reports. This utility used a form named Data Description Specifications (DDS) to define the database physical (PF) and logical (LF) files (base tables, views, and indexes). This form was columnar in design and similar in style to the RPG/III programming language (widely used on IBM midrange platforms).

In 1988, IBM announced the AS/400®. This was a single system that contained emulation environments for the System/3x line of hardware products. The OS/400® operating system also contained a built-in RDMS; however, IBM offered Structured Query Language (SQL) as an alternative to DDS for creating databases. In addition, SQL Data Manipulation Language (DML) statements were made available as an ad hoc query language tool. These statements could also be embedded and compiled within high level language (HLL) programs.

SQL Data Definition Language (DDL) has become the industry standard for defining RDMS databases. DDL statements consist of CREATE statements for defining database objects, ALTER statements for customizing existing objects (for example, adding a constraint), and GRANT statements for authorizing the access or permissions to database objects.

Many customers are in the process of modernizing their database definition and the database access. This IBM Redbook will help you understand how to reverse engineer a DDS created database, and provide tips and techniques for modernizing applications to use SQL as the database access method.

## The team that wrote this redbook

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



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



**Daniel R. Cruikshank** has been an IT professional since 1972. Over his career, Dan has been involved in many application migrations from various platforms to the IBM iSeries family (that is, System 38, AS/400, etc.). Since 1993, he has been focused primarily on resolving iSeries (AS/400) application and system performance issues at several IBM customer accounts. In 1999 he also took on the role of instructor for the IBM DB2 UDB for iSeries SQL Optimization Workshop. More recently he has taken on architectural and advisory roles within several major DB2 UDB for

iSeries reengineering projects. He is frequently called upon by the IBM Rochester Project office to assist in critical customer situations worldwide involving SQL and DB2 UDB for iSeries.



**Birgitta Hauser** has been a Software Engineer since 1996, focusing on RPG and SQL development on iSeries at Lunzer + Partner GmbH in Germany. She graduated with a business economics, and started programming on the AS/400 in 1992. She is responsible for the complete RPG, ILE, and Database programming conceptions for Lunzer + Partner's own Warehouse Management Software Package. She also works in education as a trainer for RPG and SQL developers. Since 2002 she has frequently spoken at COMMON

User Group in Germany. In addition, she is chief-editor of the iSeries Nation Network (iNN, the German part of iSeries Nation) - eNews and the author of several papers focusing on RPG and SQL.



**Rolf André Klaedtke** is an independent IT Specialist and owner of RAK Software, Consulting & Publishing in Kreuzlingen, Switzerland. Coming from a commercial background, he has accumulated almost 20 years of experience in the IT industry, mostly as a software developer on IBM's S/38 and AS/400, but also as a Client/Server and Web developer using various DBMS, tools, and languages. He is an author and the publisher of PowerTimes, a free technical journal for software/Web developers, and has organized technical conferences and user group meetings in Switzerland.



**Sharon Hoffman** is an iSeries writer and educator based in Southern California. She has worked with IBM midrange systems since 1981. Her background includes extensive application development experience as well as creation and delivery of technical education. Sharon is a Senior Technical Editor for iSeries NEWS and also writes regularly for the magazine. In addition, Sharon is an instructor for a variety of e-learning courses offered by iSeries Network, and a regular speaker at iSeries industry events.



**Warawich Sundarabhaka** is an Advisory IT Specialist with IBM Integrated Technology Services in Thailand. He has been with IBM since 1991. He has over 20 years of experience in the computer field and has worked with the AS/400 system since 1988. His areas of iSeries expertise include performance management, and application development using Java™, RPG, COBOL, and DB2 UDB for iSeries. He has taught iSeries courses for IBM Thailand education.

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

Thomas Gray  
Marvin Kulas  
Joanna Mischczyk  
International Technical Support Organization, Rochester Center

Antonia Seymour  
IBM Rochester iSeries Services Group

Jarek Miszczyk  
Kent Milligan  
IBM Rochester

George Farr  
Claus Weiss  
IBM Toronto

Julie Czubik  
International Technical Support Organization, Poughkeepsie Center

## Become a published author

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

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

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

[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

## Comments welcome

Your comments are important to us!

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

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

[ibm.com/redbooks](http://ibm.com/redbooks)

- ▶ Send your comments in an email to:

[redbook@us.ibm.com](mailto:redbook@us.ibm.com)

- ▶ Mail your comments to:

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

Archived



# Part 1

# Introduction and background

In this part we discuss and explain the iSeries developer roadmap. We also discuss the main reasons why SQL is the best approach for data definition and data access in the modernization process.

Archived



# iSeries Developer Roadmap - The big picture

From time to time, it is a good idea to lean back and have a look at the path of one's professional life. Where are we coming from? Where are we going? Have we reached our goals so far? What needs still to be done?

This chapter provides a general overview of the iSeries Developer Roadmap. We explain the reasons for the roadmap, its contents, and why it may be important to you and your organization. It may help you to evaluate your current position and provide some guidance as to the next steps to follow.

However, we do not cover too many details, and we provide no instructions on how to implement its single steps in this chapter. The other chapters in this book provide more technical details and in-depth information on how to concretely transpose the theory into practice.

Additional information on the iSeries Developer Roadmap can be found on the following web Site:

<http://www-1.ibm.com/servers/eserver/iseries/roadmap>

Part of the information in this chapter was adapted from the above site.

## 1.1 Introduction to the iSeries Developer Roadmap

We all know that technology evolves on an almost daily basis. Just some years ago, having a portable phone meant carrying a heavy, bulky case with an expensive phone that did not connect you to the world from everywhere. Nowadays, almost everybody has a small, convenient cell phone in their pocket. Many even received it for free just for signing up with a phone company.

Programming languages and the related programming models have changed as well. From the beginning many years ago, there were several paradigm shifts, and today graphical user interfaces and object-oriented programming have become widely used mainstream technologies.

The IBM @server™ iSeries and its predecessors have gained a strong reputation in the industry for featuring a very stable, robust technology, and for being ideally suited as a business computer system.

### 1.1.1 Why a roadmap

Mentioning the iSeries, in many, evokes the picture of a classical, green screen application. There is no doubt about it that in many cases this certainly is still true, despite the fact that IBM and its partners have been offering modern, graphically oriented development and operating environments for the iSeries for some years now.

In the introduction of this chapter, we already pointed out that technology evolves rapidly. Application development methodologies and database access methods are no exception to this, and may need some serious overhaul. To help companies take the next step in modernizing their applications and their development environments, IBM has created the iSeries Developer Roadmap.

This roadmap has been specifically designed to take into consideration the extent to which your shop is probably presently entrenched in a 5250 application model. The large amount of experience accumulated in relation to the traditional programming languages that support your green screen applications has also been accounted for. As mentioned in the opening paragraph of this chapter, the roadmap's aim is to help you evaluate your current position, and to see the next steps to be taken. In the following chapters we provide the reasons why we think the roadmap is important, and throughout the rest of this book we provide help and guidance on the way to modernize your applications and database access.

In Figure 1-1 on page 3 you see a graphical overview of the roadmap.

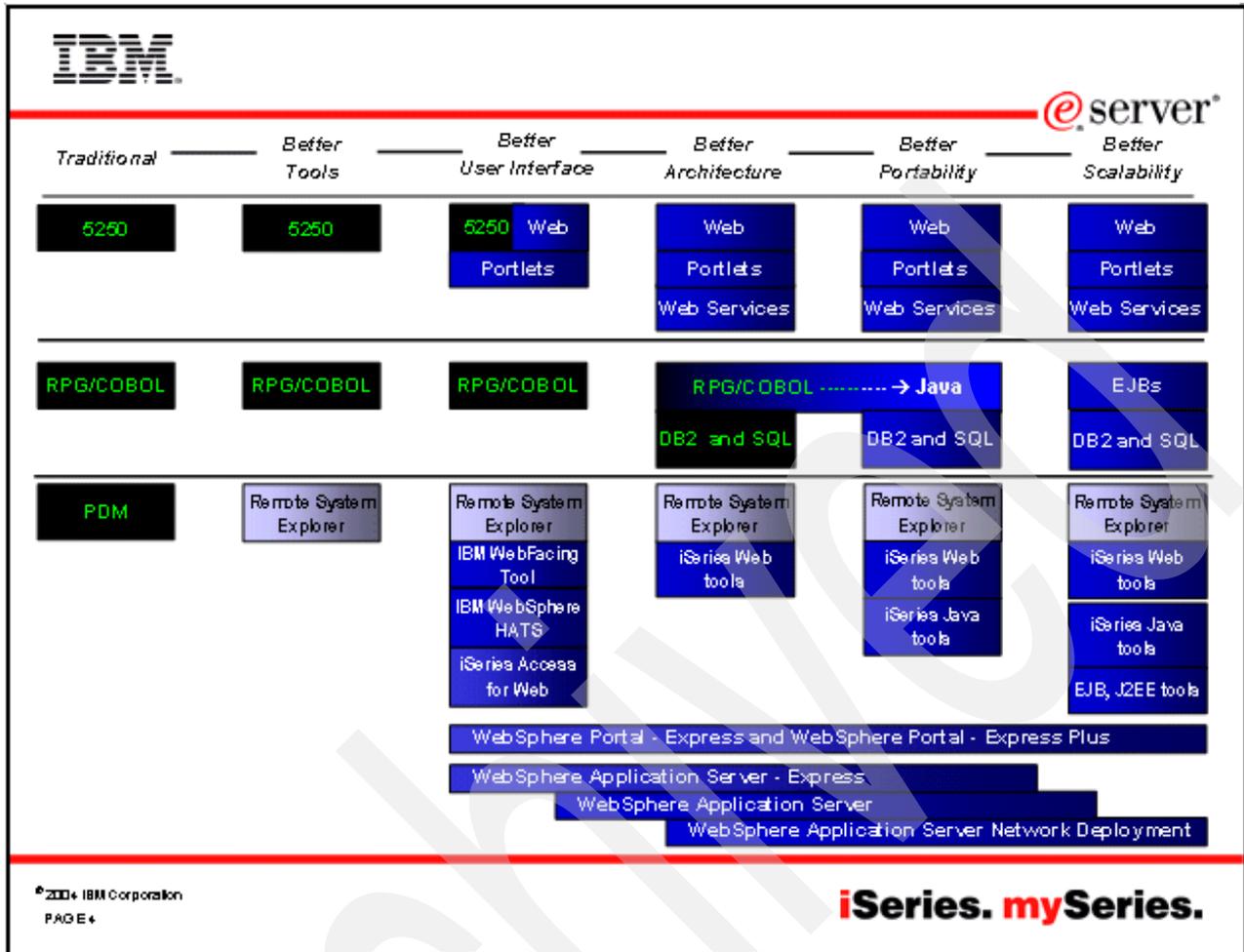


Figure 1-1 iSeries Developer Roadmap: The big picture

### 1.1.2 Why care about it

Maybe the question is less *why* but *when* you should care about the roadmap and the issues it is addressing. The field of Information Technology has always been a fast-changing one, but some technologies and methodologies are here to stay. They will evolve but not go away anytime soon.

As an individual, you may want to enhance your knowledge and learn new technologies in order to strengthen or improve your position either in your current job or in the job market. New programming languages and tools offer better support for the kinds of requests coming from users you may have to satisfy, if not now yet then possibly in the near future. In short, software development techniques, languages, and tools, as well as hardware technology, are continually evolving and becoming more efficient, and you should as well in order not to be left behind.

The reasoning for companies is almost the same as for individuals. Spending money for new hardware may give you better response and execution times. But adapting your database access methods and the architecture of your applications may gain you more flexibility when it comes to adapting your business to a changing market. You may increase the level of overall security and eliminate the source of errors by taking advantage of data integrity features built into your database, contemporarily cutting down development times, because

certain business rules do not have to be written in each application program anymore. Data extraction for reports and exchange of data with other systems is easier when the tools and systems involved use the same standard language: SQL.

The list of advantages is certainly not endless, but it is long enough to prove that having a serious look at the roadmap and evaluating your current position and the possible next steps to take is more than just a time-consuming exercise.

You may find that you are already at an advanced level. Fine. Isn't it nice to receive a confirmation that you are on the right path? If you are not at an advanced level, this book, along with more resources available from IBM, are here to help you on your way to move into the Web application world... in a staged, non-disruptive manner.

### 1.1.3 The goal

Simply put, the goal is to become a Web application-driven enterprise, taking advantage of the scalability and flexibility gains offered by modern technologies, both in hardware and in software engineering. Adapting your database access and software development practices is as much a necessity as changing your business to changing market requirements.

For example, some years ago it was considered normal to order products by writing a letter or a postcard. Then there were fax machines and now many companies accept orders 24 hours a day during the whole year through their Web sites. Companies who did not keep up with the changes could have faced receiving no more orders.

## 1.2 What is in the roadmap

The roadmap consists of discrete, achievable steps that move developers and applications on the path to an excellently implemented Web future.

Many IT shops and Business Partners that use the iSeries platform today are to be found on the left side of the chart. Typical development tasks still involve building and maintaining green-screen applications using long-available compilers, such as RPG and COBOL, via traditional 5250 tools such as Programming Development Manager (PDM), Source Entry Utility (SEU), Screen Design Aid (SDA), and Report Layout Utility (RLU), some of which are more than 20 years old.

The first step involves embracing modern tools to do the same development work previously accomplished via PDM, SEU, SDA, and RLU (see 1.2.1, "Better tools" on page 5).

The next step (explained in 1.2.2, "Better user interface" on page 6), which is considered to be urgent by end users (and also the most visible one), is a better user interface (UI) than the generations-old green screen. For most applications, this is best addressed by moving to a browser-based user interface.

In 1.2.3, "Better architecture" on page 7, we discuss a significant step where—from scratch or from cut-and-paste—you create a Web application, an application enhancement, or even a new Web service. Separating the business logic from the user interface is a very important experience, and is fundamental to a superior architecture that allows for logic reuse.

*Better portability* involves a move from creating business logic in traditional languages to writing it in Java. You use simple, standard Java—referred to as Java 2 Standard Edition (J2SE)—that accesses data in the familiar SQL ways. This step is introduced in 1.2.4, "Better portability" on page 8.

Finally, on the far right, there is *better scalability*, introduced in 1.2.5, “Better scalability” on page 8. For companies requiring highly scalable Web applications or the full object-oriented and functional power of J2EE, the J2SE Java code is replaced with Enterprise JavaBeans (EJBs) and Message Driven Beans.

### 1.2.1 Better tools

The first step in the iSeries Developer Roadmap does not involve any change to the applications in use today. Rather, it enables the replacement of the traditional development tools with more exciting and modern tools that support the same code base. Ultimately, applications will still be written in a traditional language, such as RPG or COBOL, and will continue to utilize a green-screen user interface through DDS.

#### Overview of the WebSphere® Studio family of tools

In June 2002, IBM introduced Eclipse technology and new Eclipse-based tools. As discussed in the next section, Eclipse was donated by IBM to the open source community. IBM’s offer in the tools area is known under the name WebSphere Studio and comes in different versions. Figure 1-2 shows an overview of the WebSphere Studio family of products, which are all based on Eclipse technology.

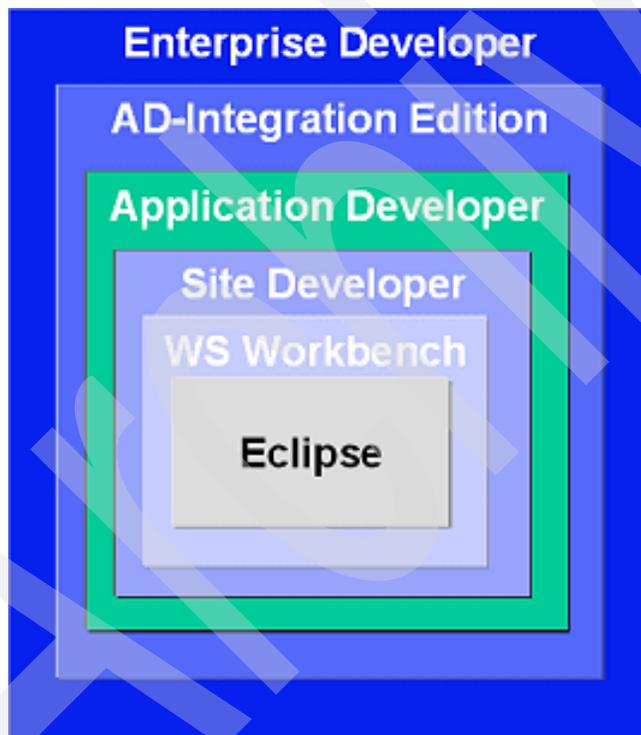


Figure 1-2 Overview of the WebSphere Studio family of products

*Site Developer* is IBM’s entry-level offering. It is used for building dynamic Web sites out of non-EJB Java. *Application Developer* extends *Site Developer* and adds support for EJBs. *Application Developer - Integration Edition* extends *Application Developer* and adds support for JCA connectors and for workflow. Finally, the *Enterprise Developer* further extends the tool and adds support for zSeries® and Enterprise Generation Language (EGL), the follow-on to VisualAge® Generator.

## A few words about Eclipse

Eclipse is an open source software development project dedicated to providing a robust, full-featured, commercial-quality, industry platform for the development of highly integrated tools. Eclipse was developed by IBM and donated to the open source community. Eclipse can be downloaded for free, including the source code. Eclipse has generated extraordinary excitement both in the development and in the tools community. It is written in Java, and can be extended by tools that are also written in Java. These tools are known as plug-ins. Out of the box, Eclipse offers an integrated development environment (IDE) that has built-in support for teams and projects and a robust and revolutionary user interface framework. It even features tools built-in to create Eclipse plug-ins. Furthermore, there are extensive and powerful tools built-in for developing Java applications with Eclipse.

Several excellent books on Eclipse have been written, and an impressive amount of free and commercial plug-ins are available.

To download Eclipse or read more about it, go to the following Web site:

<http://www.eclipse.org>

To find plug-ins and other useful information on Eclipse, check out the following Web site:

<http://www.eclipse-plugins.info>

## Tools in the iSeries development context

The traditionally used tools PDM, SEU, SDA, RLU, and the system debugger are replaced by the Remote System Explorer (RSE) in the WebSphere Development Studio Client for iSeries. RSE offers new tools for the development of RPG, COBOL, C, C++, CL, SQL, and DDS, and significantly increases productivity over the host-based 3GL tools. Furthermore, RSE introduces the same Interactive Development Environment (IDE) used further to the right of the roadmap, which means the learning curves incurred now will be useful all the way down the Web-enablement path.

Learning RSE also opens opportunities to access the next generation of third-party tools that are built on top of Eclipse. Furthermore, RSE works not only with OS/400 files, commands, and jobs, but also with IFS files and Qshell commands, and with Linux® files and commands that reside in their own logical partition (LPAR). That is, from a Microsoft® Windows® workstation, you can remotely access and edit files and run commands. RSE even works with the files and commands in remote UNIX®, Windows, or Linux servers, as well as with local Windows files and commands. Ultimately, as Java and Web services technologies are further adopted, this consistent support across file systems and command shells will be very important.

## 1.2.2 Better user interface

The next step in the roadmap deals specifically with creating better user interfaces on existing applications. There are the following three IBM options that can be used to “re-face” the application.

- ▶ IBM WebFacing Tool for iSeries
- ▶ WebSphere Host Access Transformation Services (HATS)
- ▶ iSeries Access for Web

All three produce a Web user interface (UI) from a 5250 UI, with no impact to underlying application logic. They produce UIs that run on WebSphere Application Server - Express (or later releases) or on any operating system that can support WebSphere Application Server.

The IBM WebFacing Tool for iSeries converts the display file source descriptions (DSPF DDS), at development time, into a Web application that uses Java Server Pages (JSP). The conversion is refined by the CODE Designer tool to add Web settings (via special comments) into the DDS source, which affects the result of the conversion. The CODE Designer tool is the follow-on to SDA, offering a 5250 WYSIWYG view of the application UI.

For more information on the WebFacing tool we recommend the IBM Redbook *The IBM WebFacing Tool: Converting 5250 Applications to Browser-based GUIs*, SG24-6801.

HATS, the second “re-facing” option, is part of Host Integration Solution for iSeries. It converts a 5250 or 3270 datastream, at runtime, to a browser-based interface that runs in WebSphere Application Server. Because it is a runtime conversion, it instantly transforms screens so that they can be displayed on the Web. HATS developers can easily refine, in a repeatable manner, the conversion results to improve the Web UI. The HATS development environment plugs into WebSphere Development Studio Client for iSeries.

More information on HATS is available at the following Web site:

<http://www-306.ibm.com/software/webservers/hats>

At first glance, iSeries Access for Web seems similar to HATS in implementation. They both perform 5250 to HTML conversion at execution time. However, the key strengths of iSeries Access for Web are all of the additional things that it does, in addition to datastream transformation. There are many “operational” capabilities inside the iSeries Access for Web tool that allow a user to browse job queues and output queues, display message queues, etc. While browsing a spooled file, it is possible to view the output in .pdf format and then e-mail it to other users. It is a very powerful tool for remote operations, as well as being a transformation tool.

Additional information on iSeries Access for Web is available here:

<http://www-1.ibm.com/servers/eserver/series/access/web>

### 1.2.3 Better architecture

This is the first step in the roadmap that involves re-working the application. In fact, new business logic may have to be developed as well. The goal is to modularize the code—splitting the business logic and user interfaces, and isolating functions such as database access and printing. At this step the code is moved into the most current release of its compiler language, RPG IV and ILE COBOL, for example. This is also the point at which you would identify those functions that have historically been done inside the application code, but should now ideally be moved to the database. Examples of this include referential integrity, constraints, and stored procedures, just to mention a few of them.

By re-architecting the application into a modular one, you also allow for the replacement and/or addition of modern technologies such as browser-based interfaces and distributed database activity.

#### **From DDS to SQL**

For detailed information on how to move from DDS to SQL (that is, modernize your database description and access), refer to Chapter 4, “Modernizing database definitions” on page 29.

#### **Separate business logic from presentation**

Separating the business logic from the presentation, especially the user interface, is a very important step, and is fundamental to a superior architecture that allows for logic reuse, easier maintenance, and more flexibility, for example, when it comes to adapt the business to

new interfaces such as mobile clients (such as Personal Digital Assistants (PDAs) or cell phones).

## Struts

An even better architecture can be achieved through the usage of Apache's Struts. Struts is a very popular open source Web application framework that has become a standard, and more and more companies are using it. Struts goes beyond the scope of this book. Please check out the following WEB site for more information:

<http://struts.apache.org>

Struts implements the Model-View-Controller (MVC) design pattern. For an overview of Struts and the MVC design pattern you may also check out the following Web site:

<http://publib.boulder.ibm.com/infocenter/iadthelp/index.jsp?topic=/com.ibm.etools.struts.doc/html/cstruse0001.htm>

For even more information and detailed descriptions or tutorials, there are a number of excellent books available.

## 1.2.4 Better portability

This step along the roadmap is designed for those who prefer that their application, or pieces of their application, has the ability to move and execute on multiple platforms. This step is also for those shops that might have a requirement to integrate some Java components into their traditional applications.

To achieve better portability, the business logic is written in J2SE Java (Java 2 Standard Edition), not in RPG or COBOL, which gives you the option of porting and deploying the code to any server that runs a Java Virtual Machine (JVM). In other words, your code does not have to reside or execute on an iSeries server. This opens interesting perspectives to solution providers, who can extend the market for their applications.

Using Java to write the business code also allows for the incorporation of objects and components, as well as many Java industry tools and standards that are available, such as design patterns and the Unified Modeling Language (UML). The UML is an object-oriented analysis and design language from the Object Management Group (OMG).

For more information on the UML and the OMG, visit the following WEB sites:

<http://www.uml.org>

<http://www.omg.org>

For a succinct description of the main concepts of the UML, we recommend the book *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, written by Martin Fowler (ISBN 0-321-19368-7).

## 1.2.5 Better scalability

In this final step, the J2SE Java logic is replaced with full-blown Java 2 Enterprise Edition (J2EE) Enterprise JavaBeans (EJBs) and Message Driven Beans (MDBs). This allows full exploitation of the power of J2EE, for functionality and for object-oriented concepts. It also permits the enterprise to tap into the J2EE developer community.

EJBs are beyond the scope of this book, but here is a short explanation and a pointer towards more information.

An EJB consists of server-side Java logic that implements a business object, exposing a simple set of methods to the rest of the application, while internally handling all the complexity of multiple data sources and transactions.

For more information (both a high-level overview as well as implementation examples) on Enterprise JavaBeans, we recommend the IBM Redbook *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819.

## 1.3 What is in this book

The iSeries Developer Roadmap, which we introduced in this chapter, embraces programming languages, tools, and database access, covering these topics from various points of view.

In this book we concentrate on the database aspect of the roadmap. In Part 2, “Data definition” on page 19, we define the terms used and look at the differences between DDS and SQL, showing you (among other topics) how to reverse engineer DDS described files to SQL.

In Part 3, “Data access” on page 55, we concentrate on data access; that is, we show you how to access data using native I/O methods and embedded SQL, as well as how to externalize data access.

The book closes with an overview of available development tools.

Archived

## Why modernize with SQL and DB2 UDB for iSeries

This chapter describes the benefits of moving the data definition and data access to SQL. It provides a list of key points and discusses each one in more detail.

But let us start with some simple questions: Do you remember the 5 1/4 inch floppy disks on earlier PCs or even the 8 inch floppy disks on the S/38? They were first replaced with 3 1/2 inch disks, and nowadays almost all software that you purchase comes on a CD-ROM.

Would you consider installing any software on any system using those small disks, except a really small application that fits on a single one? Probably not, because besides the fact that the initially mentioned disk drives are long gone and storage capacity was rather low, there is at least one other good reason: Performance. Technology has evolved, and the same is true for DB2 UDB on the IBM @server iSeries.

The list of key points that we think are the main reasons why you should consider moving your data definitions and data access to SQL are the following:

- ▶ **Standard-compliance:** SQL is a widely used standard.
- ▶ **Openness:** Modernizing your database provides you with more and better options to access your database using third-party tools.
- ▶ **Performance:** IBM is investing money on improving database access through SQL, not elsewhere.
- ▶ **Available skills:** In the long run, it might be easier to find developers on the market with Java and SQL rather than RPG/COBOL and DDS knowledge.
- ▶ **Functionality:** Some new functions require SQL.
- ▶ **Data integrity:** Concentrating part of your business rules in the database can cut development time and prevent bad surprises.

## 2.1 Background

To start with, let us take a very short look at the history and the components of SQL.

### 2.1.1 A short look at the history of SQL

In 1970, Dr. E.F. Codd, an employee of IBM, presented a relational model for databases. His ideas were the groundwork for all modern Relational Database Management Systems (RDBMS). The Structured English Query Language (SEQUEL) was developed in 1974 by D.D. Chamberlin, an employee at IBM's lab in San Jose (California) and renamed Structured Query Language (SQL) three years later. The first commercial database with relational capabilities was introduced with IBM's System/38, the predecessor of the AS/400 and iSeries.

The language SQL is not proprietary. This, and the fact that both the American National Standards Institute (ANSI) and the International Standards Organization (ISO) formed SQL Standards committees in 1986 and 1987, were major reasons for SQL to become a widely accepted standard that is implemented in almost all RDBMSs.

At this time, three standards have been published by the ANSI-SQL group:

- ▶ SQL89 (SQL1)
- ▶ SQL92 (SQL2)
- ▶ SQL99 (SQL3)

### 2.1.2 The main parts of SQL

SQL includes a Data Definition Language (DDL) and a Data Manipulation Language (DML). DDL contains statements to create, modify, and drop table and index definitions, as well as to grant and revoke authorities on these objects. DML contains statements to insert, retrieve, update, and delete database content.

For more information on SQL, you may refer to the following publications:

<http://publib.boulder.ibm.com/pubs/html/as400/infocenter.html>

This is the iSeries Information Center Entry Page, where you can find a lot of iSeries-related information.

## 2.2 Reasons to modernize

In this section we take a more detailed look at the key points enumerated in the introduction of this chapter.

### 2.2.1 Standard compliancy

In the introduction above, we learned that SQL is a widely used standard. But what does that mean to your business? Possibly a lot:

- ▶ It is fairly easy to find books, training classes, and other resources on SQL in case you need some advice or want to improve your knowledge of SQL.
- ▶ Most software tools support SQL, but the same is not true for DDS.
- ▶ You gain in portability. Your database definition and part of your business rules (defined using Referential Integrity; see 2.2.6, "Data integrity" on page 17, for more details) can be extracted easily and ported onto another platform or another DBMS, if that is required.

Under 2.2.4, “Available skills” on page 15, we provide you with more reasons, namely the fact that in the long run it might be easier to find human resources with knowledge of SQL rather than DDS.

## 2.2.2 Openness

Modernizing your database gives you more options, such as easier access through third-party development (Microsoft Visual Studio, Sybase PowerBuilder), reporting (Business Objects Crystal Reports), or database design tools. Some of the tools on the market do offer an interface for DDS-described DB2 UDB databases, but that is not the rule.

Furthermore, most Web-based application development tools offer built-in support for data access through SQL, that is, they often generate all necessary SQL code for you. For example, it is quite common to find configurations where the main business applications are running on the iSeries, but where an Intel®-based server running MS Windows Server is used for some Windows server based applications and/or for simple file serving. In such an environment, access to DB2 UDB for iSeries is easier if it is done using SQL.

## 2.2.3 Performance

DB2 UDB for iSeries provides two query engines to process queries: The Classic Query Engine (CQE) and the SQL query engine (SQE). Queries that originate from non-SQL interfaces such as the OPNQRYF command, Query/400, and the QQQQry API are processed by CQE.

SQL-based interfaces, such as ODBC, JDBC, CLI, Query Manager, Net.Data®, RUNSQLSTM, and embedded or interactive SQL, run through the new SQE. The routing decision for processing the query by either CQE or SQE is under the control of the system and can neither be controlled nor influenced by the user or the application. However, a better understanding of the engines and of the process that determines which path a query takes can lead to a better understanding of a query's performance.

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

Figure 2-1 on page 14 shows a high-level overview of the architecture of DB2 UDB for iSeries before OS/400 V5R2. The optimizer and database engine are implemented at different layers of the operating system. The interaction between the optimizer and the database engine occurs across the Machine Interface (MI).

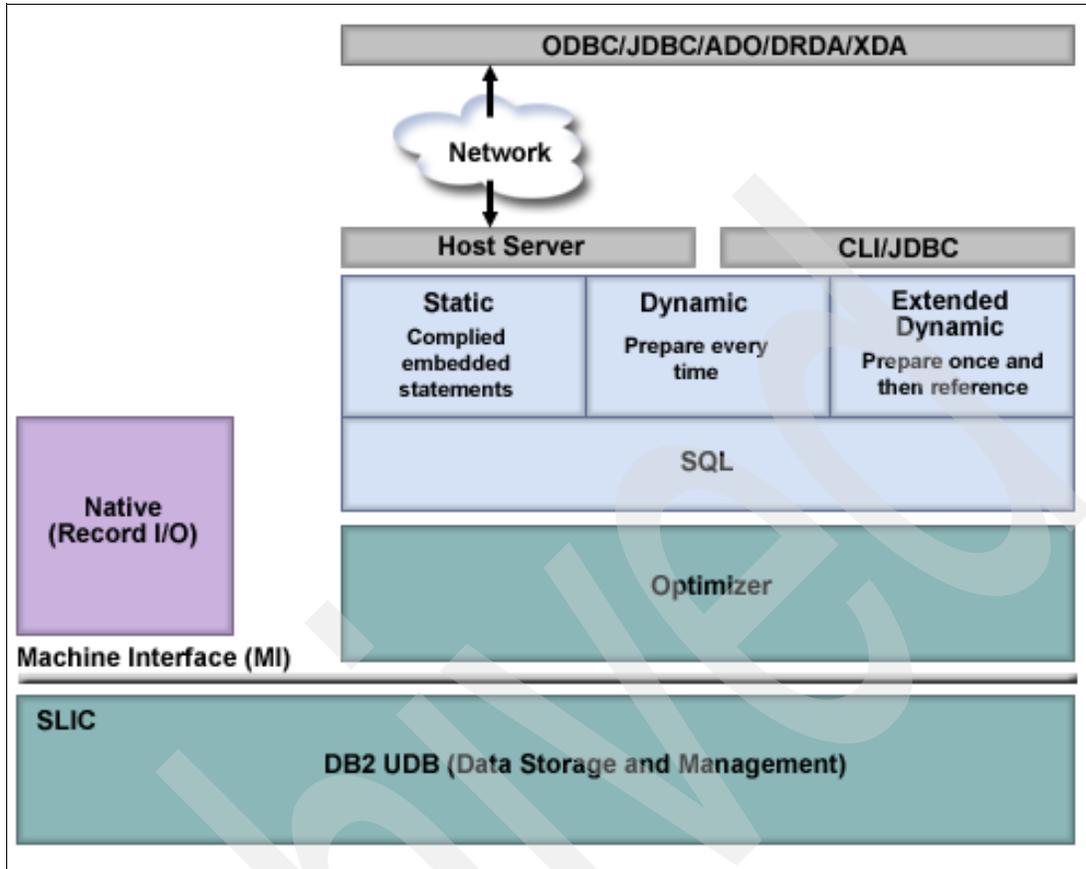


Figure 2-1 Overview of the architecture of DB2 UDB for iSeries before OS/400 V5R2

Figure 2-2 on page 15 shows an overview of the DB2 UDB for iSeries architecture on OS/400 V5R2 and i5/OS™ V5R3, and where each SQE component fits. The functional separation of each SQE component is clearly evident. In line with design objectives, this division of responsibility enables IBM to more easily deliver functional enhancements to the individual components of SQE, as and when required. Notice that most of the SQE Optimizer components are implemented below the MI. This translates into enhanced performance efficiency.

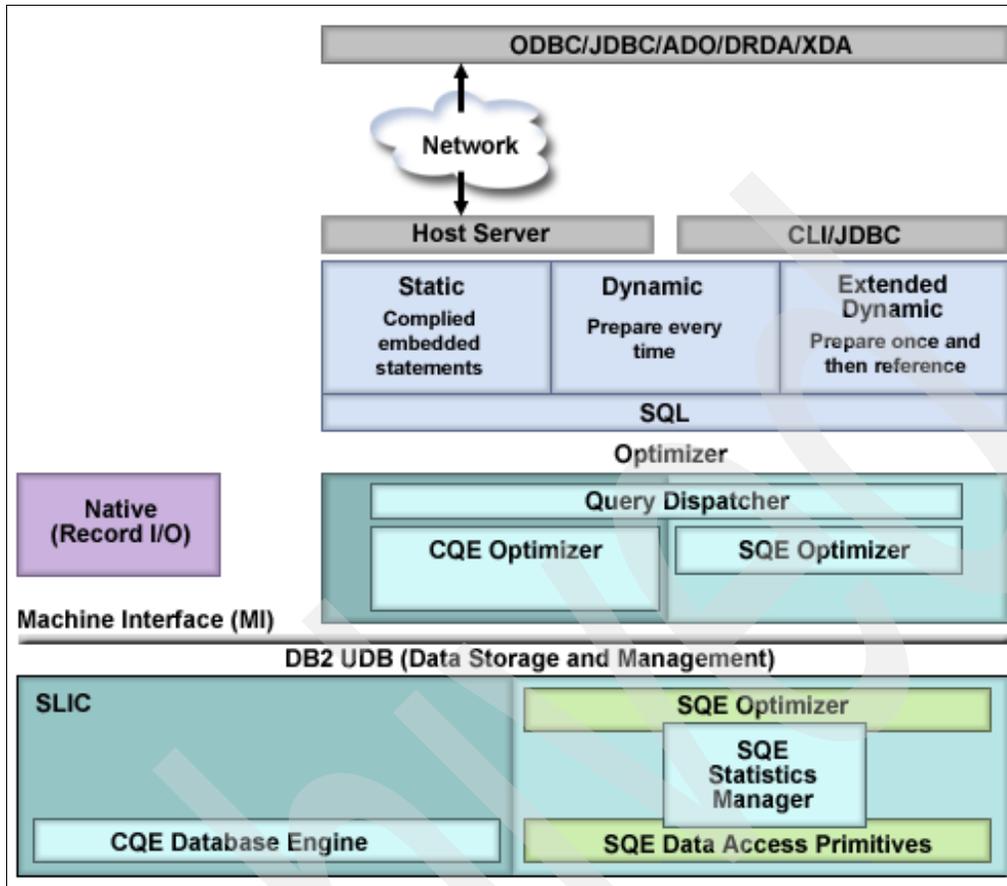


Figure 2-2 Overview of the architecture of DB2 UDB on OS/400 V5R2 and i5/OS V5R3

There are good reasons to assume that more resources will be invested into improving and enhancing database access through SQL-based interfaces. This is another good reason for considering SQL.

## 2.2.4 Available skills

For a business to be successful, more than a good product or services to offer are needed. The people that run a business are one, if not the most important, asset. Constantly improving their knowledge and skills means investing in the company, and is as important as improving the quality of the products and services offered.

But it is also very important to look forward, trying to see the trends. Those who make the best guess about what customers want to buy tomorrow and start preparing for that market today, clearly have an advantage. Of course, looking into the future is not possible, but the more flexible a business is, the better it can be adapted to new challenges. The same is true for human resources.

Accordingly, the tools and techniques used to create software have evolved dramatically over the years. Nobody would seriously consider writing a large business application using punch cards anymore. Modern programming languages and tools provide possibilities that only a few years earlier were maybe imaginable but not realizable for most of us.

Combining the above statements with the topic of this section, we think it is important to have a look at the job market and the availability of the needed skills. Since the first release of RPG

and COBOL, many other new programming languages have come and gone; some have remained and are widely used today. One of these languages is Java.

It appears clear that today more software developers are learning Java and SQL than RPG or COBOL. This means that in the long run it might be easier to find software developers with Java and SQL rather than RPG/COBOL and DDS knowledge. Modernizing the most important applications using SQL rather than DDS is a first step to make sure that these applications cannot only fulfill their purpose, but that there are also the necessary people with the right skill sets to maintain and enhance them.

## 2.2.5 Functionality

Some functions in DB2 UDB for IBM @server iSeries require the use of SQL. Among these are:

- ▶ New data types such as BLOB, CLOB, DBCLOB, and datalink.
  - Large object data types store data ranging in size from zero bytes to 2 gigabytes. The three large object data types have the following definitions:
    - Character Large Objects (CLOBs): A character string made up of single-byte characters with an associated code page. This data type is appropriate for storing text-oriented information where the amount of information can grow beyond the limits of a regular VARCHAR data type (upper limit of 32 K bytes). Code page conversion of the information is supported.
    - Double Byte Character Large Objects (DBCLOBs): A character string made up of double-byte characters with an associated code page. This data type is appropriate for storing text-oriented information where double-byte character sets are used. Again, code page conversion of the information is supported.
    - Binary Large Objects (BLOBs): A binary string made up of bytes with no associated code page. This data type can store binary data larger than VARBINARY (32 K limit). This data type is good for storing image, voice, graphical, and other types of business or application-specific data.
  - A datalink value is an encapsulated value that contains a logical reference from the database to a file stored outside the database.
- ▶ Auto-incrementing of keys (sequence objects and identity column attributes): Very often, a new row in a table must receive a unique numerical value as a record ID. Instead of writing code to create such a value, which in fact is a counter, let the database do this automatically.
- ▶ Column-level triggers: In V5R1 IBM introduced support for SQL triggers in DB2 UDB for iSeries, which allows you to write triggers using extensions to the SQL as defined by the SQL standard. The greatest advantage of using SQL triggers is portability. You can often use the same SQL trigger across other RDBMSs.
  - For more information on triggers in DB2 UDB for iSeries, we recommend reading the relevant chapter in the redbook *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database™ for iSeries*, SG24-6503-01.
- ▶ Encryption and decryption functions: The ability to encrypt and decrypt data at the column level has been enhanced with the addition of new SQL scalar functions. It is now possible to invoke a DB2 SQL statement like the following:

```
INSERT INTO orders VALUES (ENCRYPT('1234-4567-8900-0001'), 'JOHN DOE')
```

Where the first value would represent a credit card number. Only those users and applications with access to the encryption key (or password) can see the unencrypted (or

decrypted) credit card number. Others may access the table but have no access to the encrypted values.

- ▶ **Encoded Vector Indices:** An encoded vector index (EVI) is an index object that is used by the query optimizer and database engine to provide fast data access in decision support and query reporting environments. EVIs are a complementary alternative to existing index objects (binary radix tree structure - logical file or SQL index) and are a variation on bitmap indexing. Because of their compact size and relative simplicity, EVIs provide for faster scans of a table that can also be processed in parallel.
- ▶ **DB2 SMP (parallel database processing):** The DB2 UDB Symmetric Multiprocessing feature (SMP) provides the optimizer with additional methods for retrieving data that include parallel processing. Symmetrical multiprocessing is a form of parallelism achieved on a single server where multiple (CPU and I/O) processors that share memory and disk resource work simultaneously toward achieving a single end result. This parallel processing means that the database manager can have more than one (or all) of the server processors working on a single query simultaneously. The performance of a CPU-bound query can be significantly improved with this feature on multiple-processor servers by distributing the processor load across more than one processor.

Note, however, that the parallel implementation differs for both the SQL query engine and the Classic Query Engine.

## 2.2.6 Data integrity

It is general wisdom that saving data is crucial to the survival of any business. It is equally important to make sure that the data itself is correct: What good would it be to save all the information about an order, except the information relating to the customer itself? The common term for this is data integrity.

Modern RDBMSs such as DB2 UDB support data integrity through the following features:

- ▶ **Journaling:** A journal is a chronological record of changes made to a set of data. Journals record every change in a table so that in the case of a major failure, all the data can be recovered using the latest save of the database and then applying the changes recorded in the journal to the recovered database table.
- ▶ **Constraints:** Table constraints are used to enforce restrictions on the data allowed in particular columns of particular tables.
  - A table can have one **PRIMARY KEY** consisting of one or more columns.
  - A set of one or more columns may be declared as **UNIQUE**, which means that there may be no more than one row with a given value for certain columns, which are those that form the key for the table (a social security number may be a unique key, because there cannot be two people with the same social security number).
  - Columns may have a **CHECK** constraint, which would specify the values allowed for that column (for example, a field that holds a code representing the gender information of an employee can contain '1' for 'Female' or '2' for 'Male' but not '3').
- ▶ **Referential integrity:** Referential integrity (RI) is a type of constraint that deals with relationships between tables. To reuse the example from the beginning of this section, there would be a referential integrity check tying the order table to the customer table. Each order would contain a valid customer number from the Customer table as a **FOREIGN KEY**. The RI constraint would ensure that a customer cannot be deleted while there are open orders of that particular customer in the order table.
- ▶ **Commitment Control:** Commitment Control is a mechanism to handle multiple table transactions as a single unit of work. For example, the bank transfer of a salary payment involves at least two table updates: First the deduction on the bank account of the

employer, and second the credit to the employee's bank account. If there is a power failure exactly in the middle between these two updates, the whole transaction would fail and be rolled back.

- ▶ **Triggers:** Triggers are user-written programs that are run automatically whenever a change is made to a table. Triggers can be defined to run BEFORE or AFTER an INSERT, an UPDATE, or a DELETE. They are useful for tasks such as enforcing business rules, validating input data, and keeping an audit trail. Such a program could, for example, automatically send a message to a user when a value has been changed in a certain table.

Traditionally, referential integrity rules and check constraints are tied into the application program. Moving these business rules out of the application program into the database using SQL constraints offers these advantages:

- ▶ Less coding required, because the rules do not have to be written in the program, making the program smaller and therefore easier to understand and maintain.
- ▶ Better performance, because the DBMS handles these rules faster than a user-written application program.
- ▶ Better portability, because the business rules are not hidden in the program but a part of the database.
- ▶ More security: Business and data integrity rules defined in the database provide more security because they cannot be circumvented by a faulty or incompletely written application.

In DB2 UDB for iSeries, once these relationships and rules are defined to the database manager, the system automatically ensures that they are enforced at all times, regardless of the interface used to change the data (an application, the Data File Utility, Interactive SQL, and so on).

To read more about the advanced functions supported in DB2 UDB for iSeries, refer to the Redbook *Advanced Functions and Administration on DB2 Universal Database for iSeries*, SG24-4249.

Furthermore, we recommend reading the redbook *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503, or at least the chapter "Transaction management in stored procedures" for more information on transaction management.

For more information on journaling on the IBM eServer iSeries, refer to the redbook *Striving for Optimal Journal Performance on DB2 Universal Database for iSeries*, SG24-6286.

# Data definition

In this part we discuss the issues, steps, and a methodology to modernize the data definition of your existing database.

In Chapter 3, “Approaches and options” on page 21, we concentrate on the different approaches and options available, and we introduce a proposed methodology that is used throughout the book.

In Chapter 4, “Modernizing database definitions” on page 29, we concentrate on illustrating how to reverse engineer an existing DDS-created database to SQL.

Archived

## Approaches and options

This chapter explores the issues you need to consider as you begin incorporating new database development techniques with existing iSeries application and data definitions. It is important to address these considerations, because most iSeries shops already have significant investments in DDS and applications that manipulate data. Rather than abandon this investment, we encourage you to learn about capabilities that complement and enhance the functions that are available using traditional iSeries coding techniques in DDS, RPG, COBOL, and other languages. In addition, while there are many benefits to using new database functions and leveraging SQL to manipulate data, there are also many situations where languages such as RPG or COBOL are more appropriate than any of the other options.

This chapter gives you an overview of the different approaches and options that you have as you begin the process of database modernization. We also introduce a methodology for this modernization that we will be explaining throughout the book.

In this chapter, the following topics are discussed:

- ▶ Data definition considerations
- ▶ The different methods to access the database
- ▶ Methodology for the modernization

### 3.1 Data definition considerations

One of the first steps to learning SQL on the iSeries is to understand the SQL terminology. SQL and traditional iSeries development technologies use different terminology to refer to the exact same database structures; for example, an iSeries a physical file is a table in SQL terms, a field is a column, a record is a row, a library is a schema, and so forth. The terms and their SQL equivalents are found in Table 3-1.

Table 3-1 SQL terms and iSeries terms

SQL term	iSeries term
Table	Physical file
View	Non keyed logical file
Index	Keyed logical file
Column	Field
Row	Record
Schema	Library, collection, schema
Log	Journal
Isolation Level	Commitment control level

Most database objects can be used interchangeably whether they are created using DDS or SQL. For example, a physical file created using DDS can be manipulated using SQL, and likewise, a table created using SQL Data Definition Language (DDL) has an external definition that can be used in RPG and COBOL programs and is indistinguishable from one created using DDS, as illustrated in Figure 3-1.

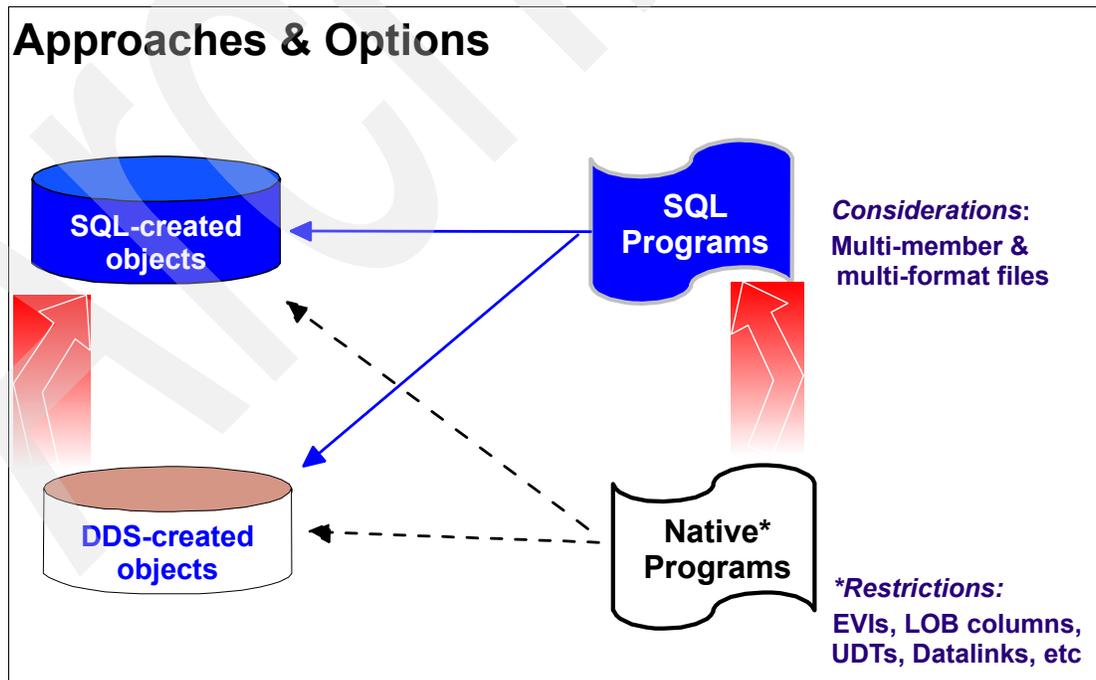


Figure 3-1 Options and approaches

When you create a table using SQL you are creating a physical file object.

However, there are some differences between SQL and DDS data definitions. For example, SQL views are very similar in concept to iSeries logical files, but logical files have some capabilities, such as support for multiple formats, that are not available for views. Likewise, views have some capabilities, such as the ability to maintain summary information, that are not available for logical files. In Chapter 4, “Modernizing database definitions” on page 29, we explore these issues in more detail.

There are also differences in data retrieval capabilities between SQL and languages such as COBOL and RPG. For example, SQL does not support the concept of multi-member files, so before you can use SQL to access any member except the first member of a multi-member file, you will need to define an SQL Alias that points to the specific member and assigns it a name. This will be discussed in Part 3, “Data access” on page 55.

In addition to different terminology for database structures such as files and fields, SQL does support some data types, such as datalinks, that are not available in DDS. There are also some restrictions on the HLL such as RPG and COBOL to define Encoded Vector Indices, access LOB columns, and use User Defined Types or datalinks.

## 3.2 Accessing the database data

To access database data on the iSeries we can differentiate between two methods:

- ▶ Native database file operations or native I/O
- ▶ SQL

While native database file operations through high-level languages (HLL) such as RPG or COBOL have been since the inception of the iSeries, SQL is a standard programming language that can be used for all databases and can be embedded in all programming languages. SQL provides not only functions to define database objects, but also to manipulate database data.

### 3.2.1 Native record level access

Native record level access or native I/O is the traditional way to access database files from HLL like RPG or Cobol.

At compile time the physical and logical files that are used in the program must be existent, because the file descriptions are bound into the program, module, or service program object. It is only possible to access different files with the same structure dynamically, either through overriding them (CL command OVRDBF) or using the keywords EXTFILE or EXTMBR for a user-opened file in the F-Specifications.

Through operations codes like READ or CHAIN the complete record can be accessed and processed. You only can get access on selected fields when using a logical file with a field selection; otherwise the whole record is read.

### 3.2.2 Data access with SQL

SQL statements are executed by a database manager. One of the functions of the database manager is to transform the specification of a result table into a sequence of internal operations that optimize data retrieval. This transformation occurs when the SQL statement is prepared. This transformation is also known as binding.

All executable SQL statements must be prepared before they can be executed. The result of preparation is the executable or operational form of the statement. Depending on the method

of preparing an SQL statement and the persistence of its operational, we can differentiate between three methods:

- ▶ Static SQL
- ▶ Dynamic SQL
- ▶ Extended dynamic SQL

Figure 3-2 gives an overview over the different access methods.

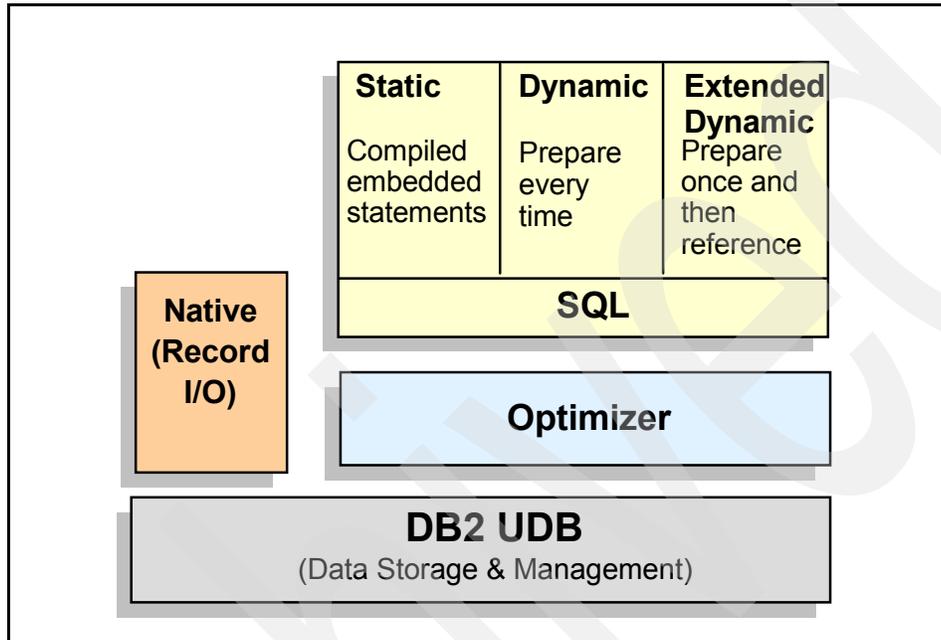


Figure 3-2 Overview of database access methods

### Static SQL

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

The host application source code must be processed by an SQL pre-compiler before compiling the host application program itself. The SQL pre-compiler checks the syntax of the embedded SQL statements and replaces SQL statements with calls to corresponding SQL function programs. If the tables used in the embedded SQL statements are not available at compile time, a SQL warning is sent, but the program, module, or service program object is nevertheless generated. In this case the access plan cannot be built at compile time but at runtime.

The pre-compiler is a part of the IBM licensed product DB2 Query Manager and SQL Development Kit for AS/400 (5769-ST1), which must be available during the compile time. The runtime support is included in the operating system. That means that compiled programs or service programs containing embedded SQL statements can be executed even without a SQL licence.

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

### Dynamic SQL

Programs containing embedded dynamic SQL statements must be precompiled like those containing static SQL, but unlike static SQL, the dynamic SQL statements are checked,

constructed, and prepared at run time. The source form of the statement is a character or graphic string that is passed to the database manager by the program using the static SQL PREPARE or EXECUTE IMMEDIATE statement. The operational form of the statement persists for the duration of the connection or until the last SQL program leaves the call stack.

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

### **Extended dynamic SQL**

An extended dynamic SQL statement is neither fully static nor fully dynamic.

The QSQRCE API (Process Extended Dynamic SQL) provides users with extended dynamic SQL capability. Like dynamic SQL, statements can be prepared, described, and executed using this API. Unlike dynamic SQL, SQL statements prepared into a package by this API persist until the package or statement is explicitly dropped.

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

For more information about SQL in DB2 UDB for iSeries, refer to the *DB2 Universal Database for iSeries SQL Reference* in the iSeries Information Center at:

<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm?info/db2/rbafzmst.html>

## **3.3 Methodology for the modernization**

The IBM Rochester iSeries Services Group has developed a methodology for reverse engineering a DDS-created database along with modernizing applications to use SQL as the database access method. We will be using this methodology in this book. This methodology consists of four stages, which will be briefly described in this section. In the following chapters we go into much more detail.

The four stages of the methodology are:

1. Reverse engineering DDS to SQL DDL
2. Creating I/O modules to access DB2 data
3. Moving business rules to the database
4. Implementing advanced database functions

This methodology is based on existing applications that access the database via high level language (HLL) I/O operations commonly referred to as native support.

### **3.3.1 Reverse engineering DDS to SQL DDL (stage 1)**

The main objective of this stage is to replace the DDS-created physical files and access paths with SQL-created tables and indexes. There may be some instances where programs need to be recompiled. At this point legacy programs can continue to use native I/O techniques. Database changes will be done via SQL.

In Chapter 4, "Modernizing database definitions" on page 29, we cover all of the steps involved in this first stage of the process.

### 3.3.2 Creating I/O modules to access DB data (stage 2)

The main goal of this stage is to minimize the impact of change on the business. This is achieved in two ways:

- ▶ By de-coupling the database access from the application program.
- ▶ By utilizing SQL views as the only way to access the data. Adding new columns to the database has no impact on existing views, thus eliminating the need to recreate the views and supporting programs. These views can be accessed via service programs or directly through ODBC or JDBC SQL statements.

The process involves a phased approach to replace native I/O operations with SQL data access methods. The strategy of using I/O modules is to limit the SQL optimization knowledge to the database programming group. This will allow the application programmers to focus on solutions to business requirements without a need to understand the complexities of database optimization.

The I/O modules mask the complexity of the database from the application programmer. For example, a HLL program may be performing several read operations to multiple files to fill a subfile. This could be replaced by a single call to an I/O module, which performs a single SQL fetch operation to a join view and returns a single host array (multiple occurrence data structure in RPG) to the caller.

In addition, the I/O modules allow the database programmer to take advantage of database functions (that is, date and time data types, variable length fields, identity columns, etc.), thus eliminating many common HLL programming requirements. This includes programming required to format date and time data, formatting address lines, etc.

In Chapter 5, "Creating I/O modules to access SQL objects" on page 57, we cover the steps required for this stage.

### 3.3.3 Moving business rules into the database (stage 3)

The objective of this stage is to start leveraging the advantages of DB2 UDB for iSeries by moving some of the business logic into the database by using:

- ▶ Referential integrity
- ▶ Check constraints
- ▶ Column-level security
- ▶ Column encryption
- ▶ Automatic key generation
- ▶ Accessing non-relational data

Since many customers have never taken the time to truly normalize their existing DDS database, this is a good opportunity to do so.

### 3.3.4 Externalizing data access (stage 4)

The objective here is to continue to replace application function with equivalent database function. This includes the use of database triggers, user defined functions (UDF), User Defined Types (UDT), stored procedures, etc.

Database triggers can be used to perform additional updates to auxiliary tables, possibly eliminating overnight batch updates. They can also be used to initiate asynchronous background tasks via data queues.

User defined functions can provide support for actions required on each row returned in a result set. Rather than performing this operation in the HLL program, the function is executed as the result set is being created.

Archived

Archived

## Modernizing database definitions

Starting from V5R1 of OS/400, IBM introduces the ability to reverse engineer database objects created via Data Definition Specification (DDS) to SQL Data Definition Language (DDL) schema. There are several reasons behind this enhancement, some of which are:

- ▶ Provide access to database enhancements that are only being made to SQL-created databases.
- ▶ Provide portability capabilities to PC-based data modeling tools that only support SQL.
- ▶ Provide compatibility with other IBM DB2 Universal Database (UDB) products and strategic application development platforms such as WebSphere Development Studio Client.
- ▶ Provide an easy way to move the data definition to SQL.

In this chapter we describe the methods and considerations in the process of modernization of the database definition.

**Note:** There is not a unique solution that fulfills all the application development environments. We introduce some guidelines to help you to modernize your applications to use and exploit the features of SQL.

## 4.1 Reverse engineering DDS to SQL DDL

In the methodology proposed in 3.3, “Methodology for the modernization” on page 25, the first step is to modernize the DDS-created databases. The main objective of this stage is to replace the DDS-created physical files and logical files (access paths) with SQL-created tables and indexes. There may be some instances where programs need to be recompiled. At this point legacy programs can continue to use native I/O techniques. One of the goals of this step is to minimize the impact of the conversion to the existing programs.

The complexity of this stage is dependent on the current condition of the existing database. If all files are fully described DDS files then there should be few, if any, exceptions. This step requires iSeries Navigator V5R1 or later. If you are not familiar with iSeries Navigator, we recommend that you read the redbook *DB2 Universal Database for iSeries Administration The Graphical Way on V5R3* - SG24-6092.

The following is an overview of the steps required in this stage:

1. Classify the existing environment.
2. Establish a list of all DDS files to be converted.
3. Establish naming conventions for SQL objects.
4. Convert the DDS to SQL DDL.
5. Review the generated SQL DDL.
6. Create the new DB2 Schema (collection) on the iSeries server.
7. Create all existing DDS logical files over the new SQL tables.
8. Migrate data and test existing programs.

### 4.1.1 Classify the existing environment

The first step in this stage is to classify and understand the existing environment that is going to be converted to SQL. The amount of effort required in the conversion is going to be determined by the environment. The different environments can be classified as the following ones:

- ▶ Class 0 - Program described files.

In this environment the DDS would contain some key fields and one great big field. The great big field would be specified in RPG in an I spec and in COBOL in the FD section.

- ▶ Class 1 - Mix of program described files and externally described.

This environment usually has no normalization in place.

- ▶ Class 2 - Externally described no referential integrity.

In this environment there is some sort of normalization, journaling is usually not used, and the physical files or logical files have unique keys defined.

- ▶ Class 3 - Externally described and some referential integrity constraints.

In this environment there is more of a degree of normalization in place. Transaction files usually are in 2NF and master files in 3NF. Primary and foreign key constraints are defined, journaling is used, and commitment control is probably not.

- ▶ Class 4 - Externally described, referential integrity, and some business logic has been moved to the database.

In this environment the database is highly normalized. Some of the business logic has been moved to the database using RI, triggers, stored procedures, UDTs, and UDFs. Journaling and commitment control are used.

Figure 4-1 illustrates the different environments and the amount of effort required for each environment.

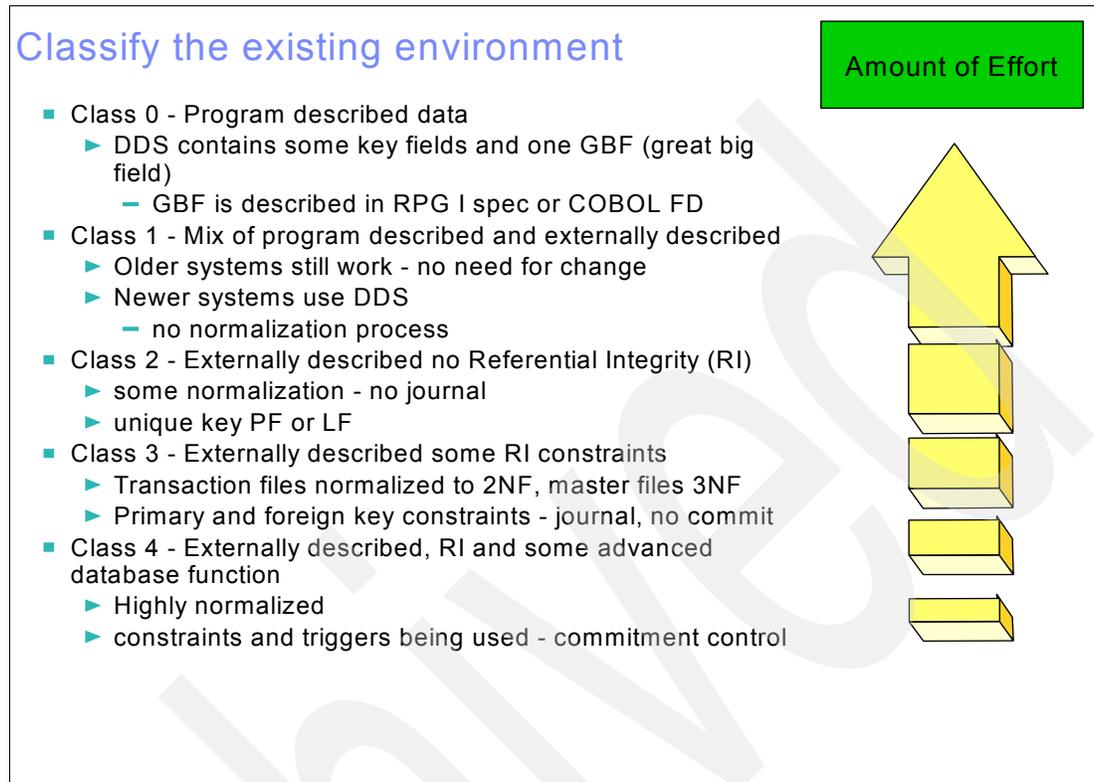


Figure 4-1 Classify the existing environment

### 4.1.2 Establishing a list of all DDS files to be converted

After determining the environment that we have, the next step in this stage is to establish a list of physical and logical files to be converted to SQL. If you want to do a pilot, choose a physical file with a reasonable amount of associated logical files and programs related to it.

The list gathers statistics to determine the physical files with the fewest, most, and average number of dependent logical files and programs. Choose a physical file with the average number of associated logical files and programs. This will be the pilot file for re-engineering.

The Table 4-1 shows an example of this list.

Table 4-1 Example of the list of DDS files

Physical file	No. of LFs	No. of programs	Start of conversion	Conversion completed	Length of conversion
PF1	2	4	8/5/2004	8/6/2004	1 days
PF2	12	24	8/6/2004	8/12/2004	6 days

The iSeries system catalogs have information of the database objects. You can query the system catalog tables to find out, for example, the number of physical files to convert using the following SQL statement:

```
select count(*) from qsys2.SYSTABLES
  where table_schema = 'APILIB' and
        table_type = 'P' and
```

```
file_type = 'D'
```

To find out the list of the physical files in a given library, use the following SQL statement:

```
select table_name, table_type, file_type from qsys2.SYSTABLES
where table_schema = 'APILIB' and
table_type = 'P' and
file_type = 'D'
order by table_name
```

In the previous example, we wanted to build the list of all data physical files in library APILIB. We queried from the system catalogs, SYSTABLES, in QSYS2 schema. The TABLE\_TYPE is 'P' for physical file, and the FILE\_TYPE is 'D' for data. You have to select FILE\_TYPE if your library also contains source physical files. Figure 4-2 shows result of the query.

TABLE_NAME	TABLE_TYPE	FILE_TYPE
CSTMR	P	D
DSTRCT	P	D
ITEM	P	D
ORDERS	P	D
ORDLIN	P	D
PARTS	P	D
STOCK	P	D

Figure 4-2 List of data physical files from SYSTABLES

This query can help you find out what physical files/logical files are in a library. It can be used to make the list of DDS files to be converted. Later in this chapter we cover the system catalog tables in more detail.

### 4.1.3 Establishing naming conventions for SQL objects

DB2 UDB on iSeries provides long name support for SQL objects and column names in a table. SQL objects and column names have a maximum length of 128 and 30 characters, respectively. But many OS/400 utilities, commands, and interfaces only support a 10-character length. This may become an issue when using native commands to access SQL objects, but we will show how to circumvent this issue.

To circumvent the long and short name, the FOR COLUMN clause on the CREATE TABLE statement allows you to specify a short name for your long column names that can be used on the interfaces that cannot support field names longer than 10 characters. If a short name is not specified, the system will automatically generate one. The CREATE TABLE statement, however, does not allow you to specify a short name for the table name. Again, the system does generate a short name automatically, but the short name is not user-friendly. For example, when you create a table named CUSTOMER\_MASTER, OS/400 automatically generates a short 10-character name, CUSTO00001, which is the first five characters of the table name and a unique 5-digit number. It might be different each time a specific object is created, depending on creation order and what other objects share the same 5-character prefix. In this case you can use the RENAME TABLE SQL statement.

To circumvent the long and short name, you can use SQL DDL statements to specify your own short name, as shown in the following examples:

- ▶ RENAME TABLE (table and view) and RENAME INDEX.

```
CREATE TABLE MYSCHEMA.CUSTOMER_MASTER
(CUSTOMER_NAME FOR COLUMN CUSNAM CHAR(20),
```

```
CUSTOMER_CITY FOR COLUMN CUSCTY CHAR(40))
```

```
RENAME TABLE MYSCHEMA/CUSTOMER_MASTER TO SYSTEM NAME CUSTMST
```

- ▶ ALIAS can be used with the short table name.

```
CREATE ALIAS MYSCHEMA/CUSTOMER_MASTER FOR MYSCHEMA/CUSTMST
```

- ▶ For the column name, you can use the FOR COLUMN clause when creating the table or view.

```
CREATE TABLE MYSCHEMA/CUSTOMER_MASTER (CUSTOMER_NUMBER FOR COLUMN  
CUSTNO INTEGER NOT NULL WITH DEFAULT, CUSTOMER_NAME FOR COLUMN  
CUSTNAM VARCHAR (50 ) NOT NULL WITH DEFAULT)
```

- ▶ For procedures and functions, you can use SPECIFIC clause at creation time.

```
CREATE PROCEDURE MYSCHEMA/MY_PROCEDURE(IN param1 CHARACTER (10 ),  
OUT param2 CHARACTER (10 )) LANGUAGE RPG SPECIFIC MYSCHEMA/MYPROC  
NOT DETERMINISTIC NO SQL EXTERNAL NAME MYSCHEMA/MYRPG PARAMETER  
STYLE GENERAL
```

It is important to create a glossary of reserved words for naming objects. This glossary would contain the complete word; how it is used; and standard 2, 3, and/or 4 character abbreviations, as shown in Example 4-1.

*Example 4-1 The example of a glossary or data dictionary*

---

Cross reference

APLIB = DB2\_APP

CUSTMAST = CUSTOMER\_MASTER

Standard abbreviations

Application codes - 1 to 3 characters

Table names - 2 to 4 characters

Functions - 3 to 4 characters

Examples

Account = A, AC, ACC, ACCT

Amount = AM, AMT, AMNT

Customer = C, CM, CUS, CUST

Date = D, DT, DTE

Suffix ids

X1 = binary radix index

PK = Primary key

E1 = EVI

---

The following are some suggested guidelines for establishing SQL table and index naming conventions.

**Note:** These suggestions are not meant to replace your existing standards and conventions. The message that we want to highlight is that there should be some naming conventions in place.

- ▶ Avoid using the object type as part of the object name. For example, do not use the words FILE, TABLE, or INDEX as part of the name.
- ▶ Use the table name and a suffix for SQL indexes. Do not be concerned about the length of the name, as indexes cannot be specified in an SQL statement. On the iSeries server, indexes provide statistics and can be used to implement a query. For example, CUSTMST\_X001 is a radix index over CUSTMST, or CUSTMST\_V001 is an Encoded Vector index over CUSTMST.

## 4.1.4 Converting the DDS to SQL DDL

Now we are ready to convert the DDS-created files to SQL. To do this we use iSeries Navigator as the tool to convert the object definitions from DDS to SQL. The tool generates the SQL statement from the existing database objects. This process is often referred to as *reverse engineering*.

Let us illustrate how to do this conversion for our APILIB library.

- ▶ To access the iSeries Navigator tool, expand an iSeries connection and select **Databases** → **Database** (S104RT9M) → **Schemas**. Then right-click the library, **APILIB**, and select **Generate SQL**, as shown in Figure 4-3.

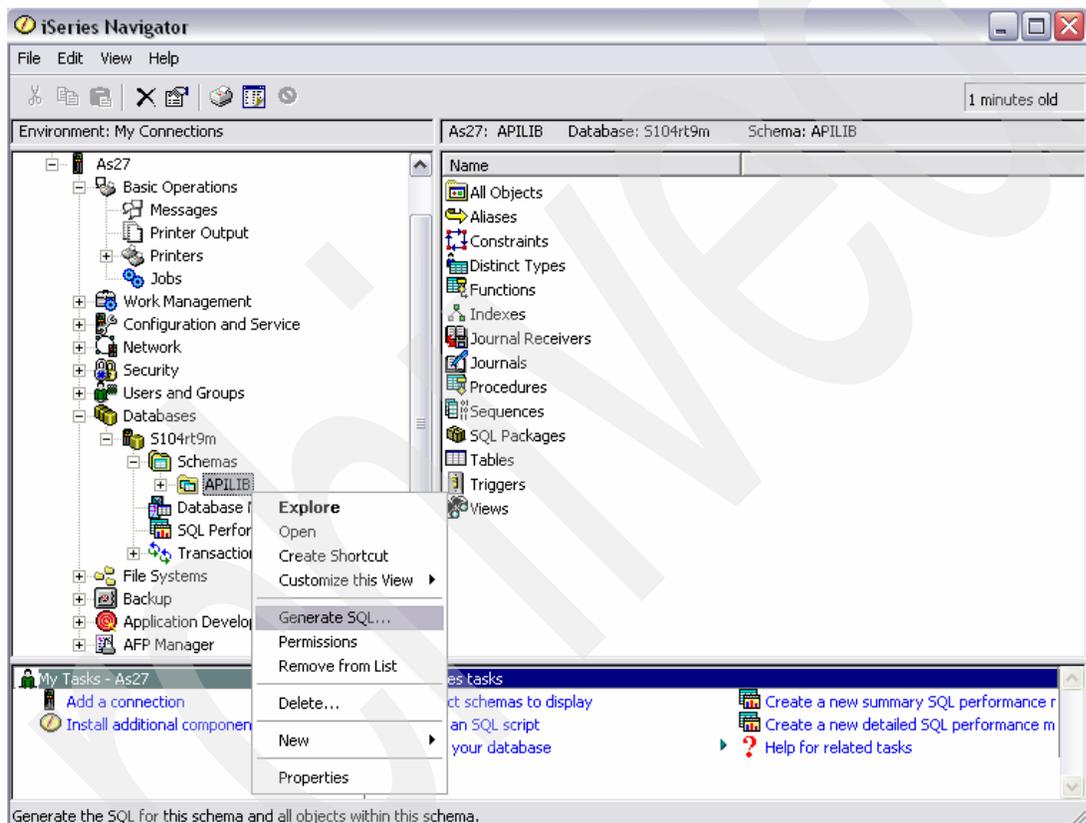


Figure 4-3 Select a library to generate SQL DDL

- ▶ The Generate SQL window (Figure 4-4 on page 35) displays the database object to be converted. You can remove unwanted objects being converted, such as source physical files.

**Important:** You should be aware that the Generate SQL option can only convert the SQL-supported objects. Thus:

- ▶ Unsupported objects such as multiple format logical files are not displayed and converted in the Generate SQL window.
- ▶ Keyed logical files are converted to SQL views.

The generated SQL source can be stored in either a source physical file member on the iSeries server or in the Run SQL scripts window or the IFS. On the Output tab, choose either **Open in Run SQL Scripts** or **Write to file**. Then click **Generate**.

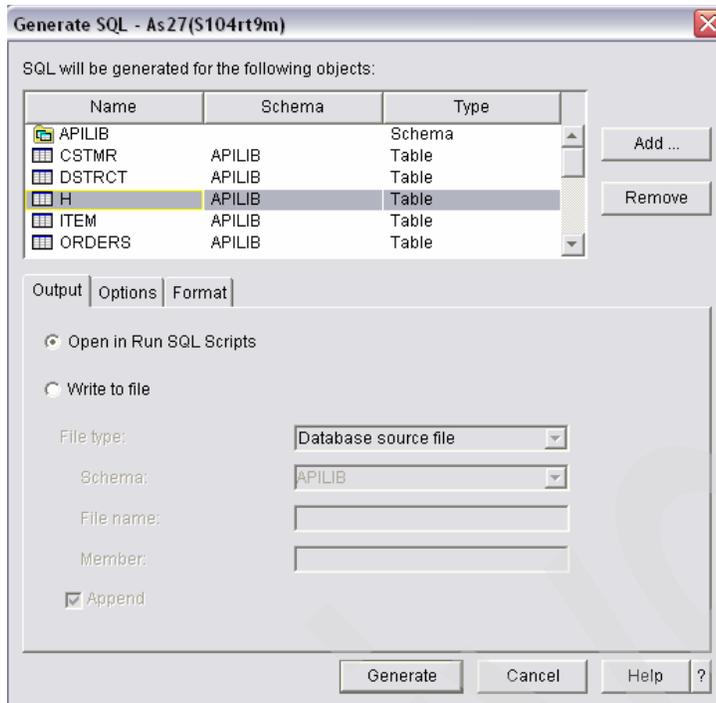


Figure 4-4 Generate SQL window

- ▶ The RUN SQL Scripts window (Figure 4-5) shows the generated SQL scripts. These are SQL DDL statements, which can be saved on your PC and executed to create new schema and other database objects from this window. You can also save the SQL scripts in the source physical file and execute by calling the RUNSQLSTM command.

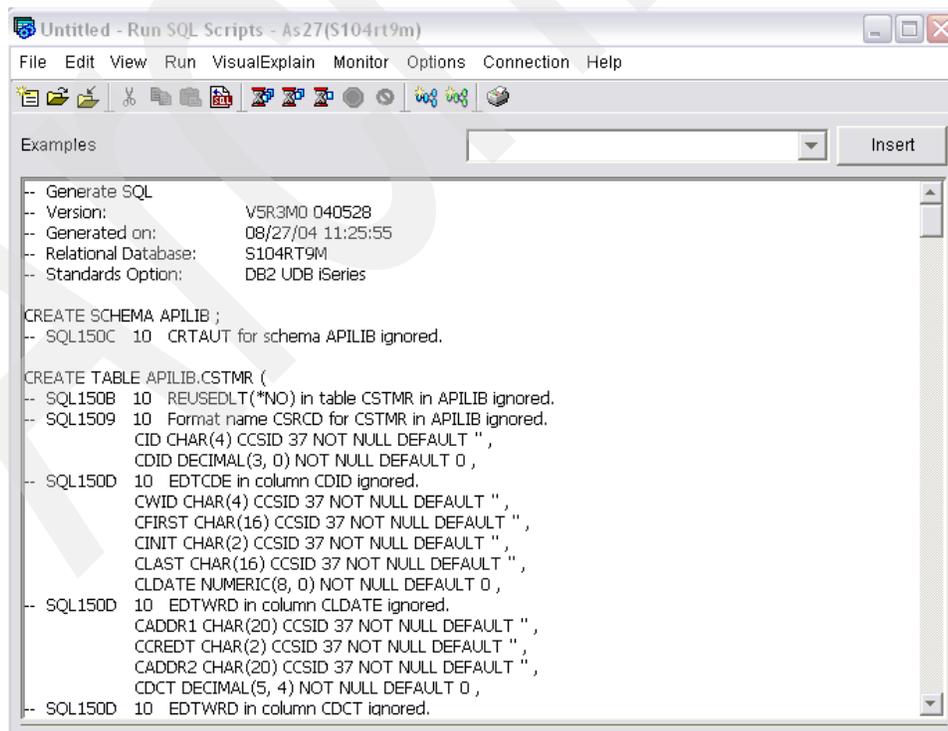


Figure 4-5 The generated SQL scripts

You can choose to generate a SQL script for an individual database object. Right-click the desired object and repeat the steps as previously described.

**Note:** The Generate SQL option on iSeries Navigator really invokes an API called QSQGNDDL on the iSeries. This API can be used directly in an CL program if you prefer.

### Manage the generated sources

As mention earlier, you can choose to store the generated sources in either source physical file members on the iSeries server or in directories on a PC or in the IFS. You should have policies to control those sources as other applications codes. For example, if you are familiar with the development environment on an iSeries server, you may be comfortable saving the sources in source physical files and using OS/400 utilities to administer things such as security and version control. If you are familiar with and use PC development tools then you may prefer to store them and manage the SQL source code on a PC or on the IFS.

## 4.1.5 Reviewing the generated SQL DDL

Now that we have generated the equivalent SQL code to reverse engineer the database it is time to review the generated SQL code. There are some changes that have to be made to this generated code in order to be able to execute it successfully. For example, you may need to change the name of the new schema for all the objects. We discuss those issues in this section.

The first thing to do in this step is to look for warning messages in the generated code. These warning messages are shown because not all of the DDS keywords have their equivalent or can be converted to SQL, such as EDTCDE. Let us illustrate this with the following example.

### Example: Converting DDS to SQL and reviewing warning messages

Example 4-2 shows the DDS description of the Order Header File ORDHDR.

*Example 4-2 DDS description for physical file ORDHDR*

---

A			UNIQUE
A	R	ORDHDRF	
A		ORHNBR	5 COLHDG('ORDER NUMBER')
A		CUSNBR	5 COLHDG('CUSTOMER NUMBER')
A		ORHDTE	L COLHDG('ORDER DATE')
A		ORHDLY	L COLHDG('ORDER DELIVERY')
A		SRNBR	10 COLHDG('ORDER SALESREP')
A		ORTOT	11P 2 COLHDG('ORDER TOTAL')
A	K	ORHNBR	

---

Example 4-3 shows the SQL statements generated to define the original Order Header File ORDHDR.

*Example 4-3 Created SQL script for the DDS-described physical file ORDHDR*

---

```

- Generate SQL
-- Version:                V5R3M0 040528
-- Generated on:           08/18/04 10:05:32
-- Relational Database:    S104RT9M
-- Standards Option:       DB2 UDB iSeries

```

```

CREATE TABLE ITS04710/ORDHDR (
-- SQL150B 10 REUSEDLT(*NO) in table ORDHDR in ITS04710 ignored.
-- SQL1509 10 Format name ORDHDRF for ORDHDR in ITS04710 ignored.
  ORHNBR CHAR(5) CCSID 37 NOT NULL DEFAULT '' ,
  CUSNBR CHAR(5) CCSID 37 NOT NULL DEFAULT '' ,
  ORHDTE DATE NOT NULL DEFAULT CURRENT_DATE ,
  ORHDLY DATE NOT NULL DEFAULT CURRENT_DATE ,
  SRNBR CHAR(10) CCSID 37 NOT NULL DEFAULT '' ,
  ORHTOT DECIMAL(11, 2) NOT NULL DEFAULT 0 ,
  PRIMARY KEY( ORHNBR ) ) ;

LABEL ON TABLE ITS04710/ORDHDR
  IS 'Order Header without Alias Field Definitions' ;

LABEL ON COLUMN ITS04710/ORDHDR
( ORHNBR IS 'ORDER NUMBER' ,
  CUSNBR IS 'CUSTOMER NUMBER' ,
  ORHDTE IS 'ORDER DATE' ,
  ORHDLY IS 'ORDER DELIVERY' ,
  SRNBR IS 'ORDER SALESREP' ,
  ORHTOT IS 'ORDER TOTAL' ) ;

LABEL ON COLUMN ITS04710/ORDHDR
( ORHNBR TEXT IS 'ORDER NUMBER' ,
  CUSNBR TEXT IS 'CUSTOMER NUMBER' ,
  ORHDTE TEXT IS 'ORDER DATE' ,
  ORHDLY TEXT IS 'ORDER DELIVERY' ,
  SRNBR TEXT IS 'ORDER SALESREP' ,
  ORHTOT TEXT IS 'ORDER TOTAL' ) ;

```

Note from the previous SQL code the following SQL warning messages:

- SQL1508** REUSEDLT(\*NO) in table ORDHDR in ITS04710 ignored.
- While creating a physical file with CRTPF you can specify the Reuse deleted record option (REUSEDLT). The default value is \*NO. That means that when a record is deleted only the first bit will be changed. To delete the record physically you have to execute the CL command RGZPFM (Reorganize Physical File Member).
- For SQL tables you do not have this option. When a record is deleted in a SQL table, the allocated storage will be reused when a new row is written. It is not possible to reactivate deleted records.
- SQL1509** Format name ORDHDRF for ORDHDR in ITS04710 ignored.
- The format name and the table name of the new SQL table will be identical.

**Note:** SQL tables are always created with reuse deleted records \*YES, so that it is not possible to reactivate deleted records.

In SQL tables table name and format name are identical.

Example 4-4 on page 38 contains a modified version of the SQL script in Example 4-3 on page 36. The table is generated with the format name, and the long column names are added into the CREATE TABLE statement. A new statement to rename the table to a long SQL name and to set the system name of the table to the old file name is added. All other statements are not changed.

*Example 4-4 Necessary changes in the SQL script for file ORDHDR*

---

```
-- Generate SQL
-- Version:                V5R3M0 040528
-- Generated on:           08/18/04 10:05:32
-- Relational Database:    S104RT9M
-- Standards Option:       DB2 UDB iSeries

CREATE TABLE ITS04710/ORDHDRF (
  Order_Number for ORHNBR CHAR(5) CCSID 37 NOT NULL DEFAULT '' ,
  Customer_Number for CUSNBR CHAR(5) CCSID 37 NOT NULL DEFAULT '' ,
  Order_Date for ORHDTE DATE NOT NULL DEFAULT CURRENT_DATE ,
  Order_Delivery for ORHDLY DATE NOT NULL DEFAULT CURRENT_DATE ,
  Order_SalesRep for SRNBR CHAR(10) CCSID 37 NOT NULL DEFAULT '' ,
  Order_Total for ORHTOT DECIMAL(11, 2) NOT NULL DEFAULT 0 ,
  PRIMARY KEY( Order_Number ) ) ;

RENAME TABLE ITS04710/ORDHDRF
  TO Order_Header
  FOR SYSTEM NAME ORDHDR;

LABEL ON TABLE ITS04710/ORDHDR
  IS 'Order Header' ;

LABEL ON COLUMN ITS04710/ORDHDR
( ORHNBR IS 'ORDER NUMBER' ,
  CUSNBR IS 'CUSTOMER NUMBER' ,
  ORHDTE IS 'ORDER DATE' ,
  ORHDLY IS 'ORDER DELIVERY' ,
  SRNBR IS 'ORDER SALESREP' ,
  ORHTOT IS 'ORDER TOTAL' ) ;

LABEL ON COLUMN ITS04710/ORDHDR
( ORHNBR TEXT IS 'ORDER NUMBER' ,
  CUSNBR TEXT IS 'CUSTOMER NUMBER' ,
  ORHDTE TEXT IS 'ORDER DATE' ,
  ORHDLY TEXT IS 'ORDER DELIVERY' ,
  SRNBR TEXT IS 'ORDER SALESREP' ,
  ORHTOT TEXT IS 'ORDER TOTAL' ) ;
```

---

### **Unsupported DDS keywords**

As stated above, not all of the DDS keywords have their equivalent or can be converted to SQL. The unsupported DDS keywords and file attributes that need to be considered are described in this section.

#### ***File-level keywords***

The file-level keywords are:

- ▶ Files that use any of the following keywords: ALTSEQ, FCFO, FIFO, LIFO  
These keywords will be ignored.
- ▶ Join logical files with JDFTVAL or JDUPSEQ  
A LEFT OUTER JOIN clause will be generated, but the join default value will be the null value and the JDUPSEQ keyword will be ignored.

### **Field-level keywords**

The field-level keywords are:

- ▶ Physical or logical files that use any of the following keywords: CHECK, CHKMSGID, CMP, DATFMT, DIGIT, EDTCDE, EDTWRD, TIMFMT, RANGE, REFSHIFT, UNSIGNED, VALUES, or ZONE

These keywords will be ignored.

- ▶ Logical files that use any of the following keywords: CCSID or TRNTBL

These keywords will be ignored.

Generally, it is not possible to stop using the existing traditional HLL programs and turn to SQL-based applications. To co-exist with the existing applications, the SQL scripts may need to be modified for some reasons before creating the new SQL database. Some of the issues that you will have to address are covered in the next sections.

### **Record format level check**

The existing HLL programs may not be able to access the new database objects in the case of the record format level check. This problem can be solved by doing one of the following:

- ▶ Re-compile the existing HLL programs in order to know the new database definitions.
- ▶ Use the command OVRDBF to specify LVLCHK(\*NO). You may need to modify your applications, such as the CL programs, to use this option.
- ▶ Use the command CHGPF to permanently change LVLCHK(\*NO). Note that unpredictable results may be obtained in the future due to applications changed.

### **Record format name**

The record format name of the SQL-created table is the same as the file name. In the existing applications, the DDS files may be accessed by using either the file name or the record format name. We recommend using the following steps to solve this issue:

1. Let us assume that we have a DDS PF named CSTMR and its record format name is CSRCD. Let us use the record format to create the new table.

```
CREATE TABLE APILIB.CSRCD (...
```

2. Taking advantage of the SQL long name support, you can optionally re-name the CSRCD table to have a more meaningful name such as CUSTOMER\_MASTER, by using the SQL RENAME statement.

```
RENAME TABLE APILIB.CSRCD TO CUSTOMER_MASTER
```

3. Rename the table to use the existing name.

```
RENAME TABLE APILIB.CUSTOMER_MASTER TO SYSTEM NAME CSTMR
```

Finally, the new SQL table, CSTMR, is created and has the record format named CSRCD.

### **Multiple member physical or logical file**

SQL does not have the concept of multi-member as the DDS PF/LF file. The generated table contains one member, and the MAXMBRS attribute cannot be changed. You may choose one of the two following recommendations:

- ▶ Use the existing DDS PF/LF files and create an SQL alias for each physical/logical file member to be used by SQL, as illustrated in the following example:

```
CREATE ALIAS MYSCHEMA/JANSALES FOR SALES(JANUARY)
CREATE ALIAS MYSCHEMA/FEBSALES FOR SALES(FEBRUARY)
.....
```

```
CREATE ALIAS MYSCHEMA/DECSALES FOR SALES(DECEMBER)
```

The existing HLL programs still keep running with the existing PF/LF files.

- ▶ Use the CREATE TABLE .... LIKE statement to create multiple tables with the identical column definitions.

```
CREATE TABLE MYSCHEMA.JANSALES LIKE MYSCHEMA.SALES
CREATE TABLE MYSCHEMA.FEBSALES LIKE MYSCHEMA.SALES
.....
CREATE TABLE MYSCHEMA.DECSALES LIKE MYSCHEMA.SALES
```

You need to modify the traditional programs to access new tables.

### Multiple format logical file

Application developers who have been in the S/38, AS/400, and iSeries development arena have been defining and using multiple format logical files for many years. Those developers know the power of this feature, but they also know that SQL cannot process a multiple format file. SQL does not understand what a multiple format logical file is.

In our conversion process there are some alternatives. You can create a view with join and/or UNION operators. Note that even a join view with or without UNION does not provide a direct equivalent to a multiple format logical file. However, for general purposes, joins and unions can combine data from different tables in a way that serves to what a multiple format logical file provides. For this issue, you may choose one of the two following proposals:

- ▶ Create DDS multiple format logical files over the new SQL tables that are used by the existing HLL programs, which is our proposal for this first stage.
- ▶ Modify the existing applications. An example of application changes is to modify the HLL programs to use new I/O module to access new SQL database objects, something that will be done on our second stage of the methodology.

### Keyed logical files

It is important to note that the Generate SQL option converts the keyed logical files into SQL views. It is necessary to manually create the indexes using the CREATE INDEX SQL statement.

### Join logical file

As of V5R2, join logical files are converted into SQL views. A view does not contain key fields. You have to generate the CREATE INDEX statements based on the DDS keyed logical file information. Indexes can be created manually via iSeries Navigator and then reverse engineered into SQL source.

The generated SQL scripts for join logical files will create SQL views that have definitions equivalent to the DDS join logical file without the key values. In this case as well you have to generate the CREATE INDEX statements based on the key values of the logical file. Note that it is decision of the optimizer to use or not to use the indexes that you create.

## 4.1.6 Creating the new DB2 schema on the iSeries server

Now that we have the SQL scripts and they have been reviewed and modified to address the issues, it is time to start creating the environment for the reengineered database. The starting point of this step is to create a DB2 schema to contain the new database objects created by SQL. In this section we explain the schema concept and the journaling or logging mechanisms that it has.

On DB2 UDB for iSeries, a schema is used to group related database objects. A DB2 UDB for iSeries schema is actually a collection of DB2 objects and OS/400 objects. When the CREATE SCHEMA statement is executed, the following objects are created:

- ▶ OS/400 library
- ▶ OS/400 journal and journal receiver
- ▶ DB2 views containing schema-wide catalog

Figure 4-6 shows the journal, journal receiver, and the DB2 views created when we execute the CREATE SCHEMA MYSCHEMA SQL statement.

### The library

The library is the logical "container" of the objects and is where the objects are stored. DB2 object names have to be unique within this container. The DB2 views created as part of the schema are a set of views that describe tables, views, indexes, packages, procedures, functions, triggers, and constraints. These views are built over the base set of catalog tables in libraries QSYS and QSYS2 and only include information on objects contained in that schema.

### Journal and journal receiver objects

DB2 UDB for iSeries logs change to a table through a process called journaling. The OS/400 journal records database object changes by sending information to the journal receiver. Thus, a journal receiver is analogous to a log file found in other RDBMs. When a table is created into the schema it is automatically journaled to the journal object created by DB2 UDB for iSeries during execution of the CREATE SCHEMA statement.

SQL Name	Type	Owner	Text
QSQJRN	Journal	OAK	COLLECTION - created by SQL
QSQJRN0001	Journal Receiver	OAK	COLLECTION - created by SQL
SYSCHKCST	View	OAK	SQL catalog view
SYSCOLUMNS	View	OAK	SQL catalog view
SYSCST	View	OAK	SQL catalog view
SYSCSTCOL	View	OAK	SQL catalog view
SYSCSTDEP	View	OAK	SQL catalog view
SYSINDEXES	View	OAK	SQL catalog view
SYSKEYCST	View	OAK	SQL catalog view
SYSKEYS	View	OAK	SQL catalog view
SYSPACKAGE	View	OAK	SQL catalog view
SYSREFCST	View	OAK	SQL catalog view
SYSTABLEDEP	View	OAK	SQL catalog view
SYSTABLES	View	OAK	SQL catalog view
SYSTRIGCOL	View	OAK	SQL catalog view
SYSTRIGDEP	View	OAK	SQL catalog view
SYSTRIGGERS	View	OAK	SQL catalog view
SYSTRIGUPD	View	OAK	SQL catalog view
SYSVIEWDEP	View	OAK	SQL catalog view
SYSVIEWS	View	OAK	SQL catalog view

Figure 4-6 The objects after creating the schema

Even though DB2 automatically starts journaling for the table object, it is the user's responsibility to manage the journal and journal receiver objects. As you can imagine, these journal receiver objects containing the database changes can become quite large, so ignoring the auto-created journal receivers is not an option unless you have unlimited disk space. However, the journal receiver objects should not be deleted arbitrarily to save disk space. In addition, even though journaling can be stopped for a table, it is not recommended, since applications accessing non-journaled objects are unable to specify an isolation level, and the applications cannot issue commit and rollback operations.

Most iSeries customers use their journal receivers as a core part of their database backup and recovery process. One approach would be to save a complete copy of the table backup media once a week and then on a nightly basis just save the changes to the table (that is, the journal receiver) to back up media and then repeat this process every week. Once the journal receiver has been backed up, the journal receiver object can be deleted. More information about the proper steps for saving and deleting journal receiver objects can be found in the *Backup and Recovery Guide* in the iSeries Information Center. The online version of the iSeries Information Center can be found at:

<http://www.iseries.ibm.com/infocenter>

It is possible to have the system automatically delete the journal receiver object by specifying DLTRCV(\*YES) on the CHGJRN CL command or setting the equivalent option on the iSeries Navigator interface. However, this option should only be used after reading the Backup and Recovery Guide and understanding the behavior and implications of this option.

You will also notice over time that multiple journal receiver objects will appear in the schema created. That is because DB2 UDB for iSeries creates the journal object with the system-managed receiver option (for example, MNGRCV(\*YES) ) on the CHGJRN CL command). The graphical journal management interface in Figure 4-7 on page 43 denotes a system-managed journal receiver with the System radio button selected in the Receivers managed by section. With this option specified, DB2 UDB automatically creates a new journal receiver each time the system is restarted (that is, system IPL) and whenever the attached receiver reaches its size threshold. The current journal receiver is detached and a new one is created.

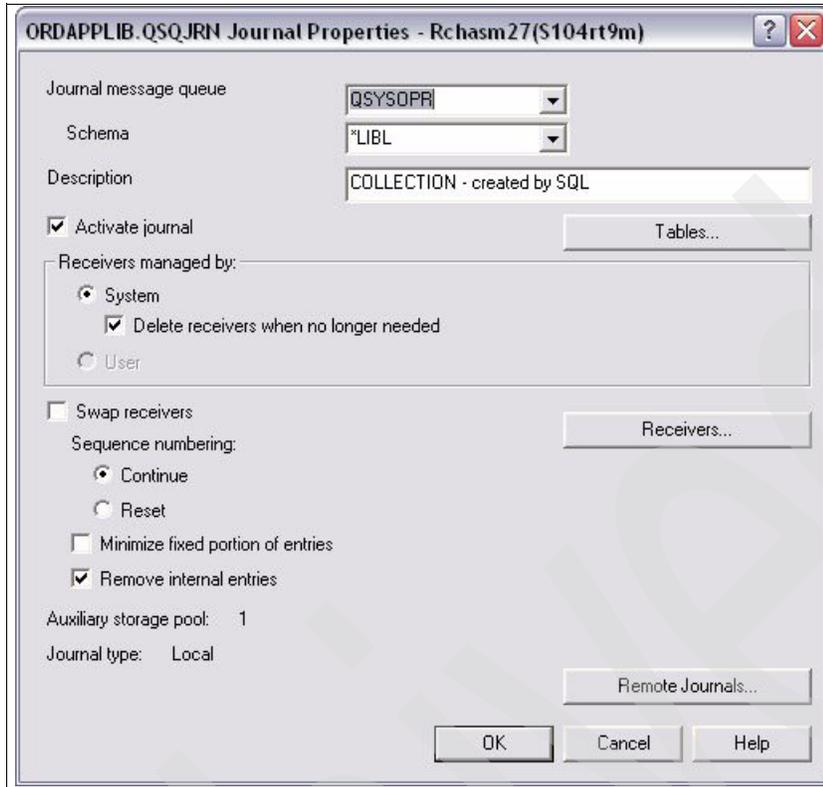


Figure 4-7 Journal Properties window

It is easier to back up a journal receiver when it is detached. Also, the receiver must be detached before it can be deleted. So you can see how this option makes it easier to manage journal receivers. Managing journal performance is a topic outside of the scope of this book, but you can reference the redbook *Striving for Optimal Journal Performance*, SG24-6486, for more information on this.

Since DB2 UDB for iSeries has no concept of table spaces and automatically stripes and balances DB2 objects across disks, the journal and journal receiver objects are going to be the only schema objects that require space management from an administrator's perspective. The only other space administration task is making sure that there is enough disk space available on the system. For more information on this topic refer to *Managing DB2 UDB for iSeries Schemas and Journals*, written by Kent Milligan, and found at:

<http://www7b.boulder.ibm.com/dmdd/library/techarticle/0305milligan/0305milligan.html>

**Note:** The journaling function can be manually disabled for each table if journal management is already in place.

### 4.1.7 Create all existing DDS logical files over the new SQL tables

After creating the new schema, we use the generated and reviewed SQL scripts to create the database objects such as:

- ▶ Tables
- ▶ Indexes

After having created the tables, we have to create DDS logical files over all the SQL tables in order to be accessed by the existing HLL programs. After creating the DDS logical files over

the new SQL database ensure that all LF access paths are implicitly sharing the SQL index access paths.

Let us illustrate this step with one example. Let us suppose that we have:

- ▶ A DDS physical file named DDS\_FILE
- ▶ A DDS logical file named DDS\_LF
- ▶ An RPG program named DDS\_LF\_RPG

The following is the source and record format information.

The physical file DDS\_FILE source code is shown in Example 4-5.

*Example 4-5 Physical file DDS\_FILE*

---

A	R	DDS_FILER			
A		FIELD1	9B	0	
			Record Length	Format Identifier	Level
Format	Fields				
DDS_FILER	1		4	<b>35FCE265C68AE</b>	

---

The logical file DDS\_LF source code is shown in Example 4-6.

*Example 4-6 Logical file DDS\_LF*

---

A	R	DDS_FILER			PFILE(DDS_FILE)
			Record Length	Format Identifier	Level
Format	Fields				
DDS_FILER	1		4	<b>35FCE265C68AE</b>	

---

The source code of the DDS\_LF\_RPG program is shown in Example 4-7.

*Example 4-7 Program code for DDS\_LF\_RPG*

---

```

FDDS_LF  if  e          disk
*-----*
* Record Format ID Check                               *
*-----*
/FR
  READ DDS_LF;
  RETURN;
/END-FR

```

---

If we execute a DSPPGMREF to the DDS\_LF\_RPG program we will get the result shown in Example 4-8.

*Example 4-8 DSPPGMREF of program DDS\_LF\_RPG*

---

Number of record formats . . . . . :	1	
Record Format	Format Level Identifier	Field Count
DDS_FILER	<b>35FCE265C68AE</b>	1

---

Note that all the format level identifiers are in sync.

The next step is to reverse engineer the DDS PF into SQL. The new SQL table will have a different name. This is shown in Example 4-9.

*Example 4-9 Reverse engineer of the DDS physical file*

---

```

CREATE TABLE DDL_TABLE (

```

```

FIELD1 INTEGER NOT NULL DEFAULT 0 ) ;
      Record Format Level
Format   Fields Length Identifier
DDL_TABLE      1     4 34EDC035C85A6

```

---

The DDL table format ID does not match the DDS PF format ID. This is expected. The next step is to convert the original DDS PF source to a LF referencing the SQL table, as shown in Example 4-10. (Note that this logical file must contain the original column definitions.)

*Example 4-10 Logical referencing the SQL table*

```

A      R DDS_FILER      PFILE(DDL_TABLE)
A      FIELD1          9 0B

```

---

After creating the DDS LF DDS\_FILE, the format ID remains the same as the original DDS PF DDS\_FILE; however, it is now based on the new SQL DDL table, as shown in Example 4-11.

*Example 4-11 Format IDs in sync*

```

Based on file . . . . . : DDL_TABLE
      Member . . . . . : DDL_TABLE
Record Format List
      Record Format Level
Format   Fields Length Identifier
DDS_FILER      1     4 35FCE265C68AE
DDS_FILER      1     4 35FCE265C68AE Original file

```

---

The next step is to change DDS LF DDS\_LF to share the format of DDS LF DDS\_FILE, as shown in Example 4-12.

*Example 4-12 Logical file sharing format of physical file*

```

A      R DDS_FILER      PFILE(DDL_TABLE)
A      FORMAT(DDS_FILE)

```

---

After recreating the DDS LF DDS\_LF, the format ID remains the same as shown below.

*Example 4-13 Format IDs in sync*

```

Based on file . . . . . : DDL_TABLE
      Member . . . . . : DDL_TABLE
Record Format List
      Record Format Level
Format   Fields Length Identifier
DDS_FILER      1     4 35FCE265C68AE
DDS_FILER      1     4 35FCE265C68AE Original file

```

---

The RPG program still contains the original format level ID and does not need recompiling, nor would any programs that reference DDS PF DDS\_FILE. At this point you are now taking advantage of database enhancements made available to SQL (in essence, faster reads and larger access path sizes).

Figure 4-8 on page 46 shows how we have moved from an existing environment to a new environment.

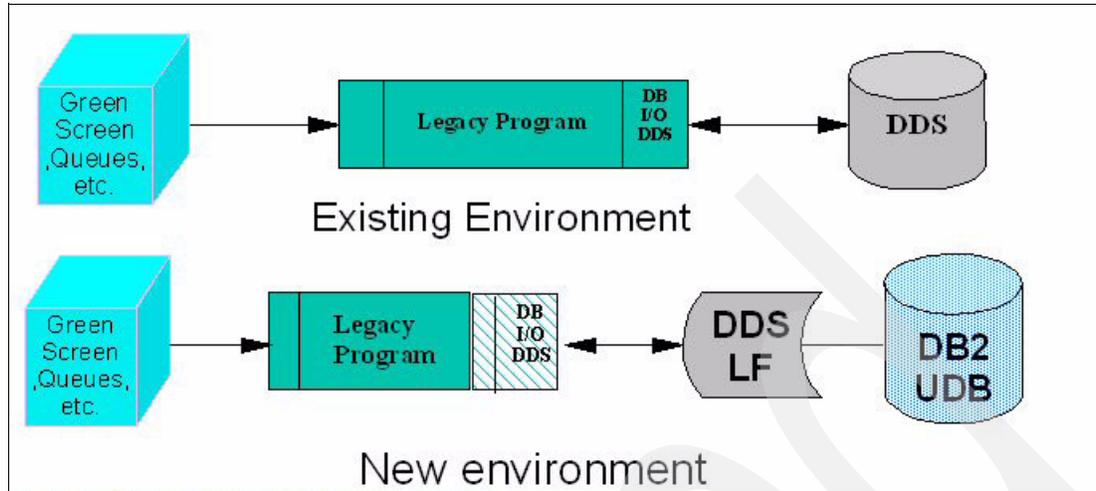


Figure 4-8 Stage 1 - Reverse engineering

#### 4.1.8 Migrate data and test existing programs

Now that we have created the schema, tables, and indexes, it is time to migrate the data into the tables. An automated test tool is beneficial at this stage. Converting an entire DDS database to SQL DDL and creating the new tables can be done in a few days. Migration of the data and testing of the programs is time consuming.

You need to create a set of file conversion programs. A conversion program reads data from an existing DDS file and writes it to a new SQL table. It may be a simple CL program, as below:

```
CPYF FROMFILE(APILIB/CSTMR) TOFILE(NEWSHEMA/CSTMR) MBROPT(*REPLACE)
```

**Note:** This example can be used if each new column attribute is identical to the existing field in DDS.

In this step it is a good opportunity to do some cleansing of the data. Some of the things to check are:

- ▶ Ensure that there are no invalid characters in the DDS numeric data. SQL checks at insert time versus read time for DDS.
- ▶ If the DDS database contains numeric fields representing dates then verify that the dates are valid.

To validate the data in the records, you would probably need to create HLL programs to check the validity of the fields, such as valid date data and so on. These efforts save time down the road.

Use the test tool to establish standard test scripts that will touch each and every file. As each file is converted, the test script can be rerun to ensure that conversion did not introduce new invalid data.

By doing all the previous steps we have:

- ▶ Converted all the DDS-created files into SQL-created objects (tables, indexes, views, and alias).
- ▶ In some cases we have solved some issues by creating logical files (for example, multiformat logical files).

- ▶ Migrated the data from the DDS objects to the SQL testing environment.
- ▶ Tested that all the application programs are running correctly.

As a last step, we may need to move this new environment (that has been a testing environment) to a production environment. In the next section we explain considerations regarding this movement.

### **Moving a schema from testing to production environment**

The preferred method for moving a schema to a different schema or system (for example, moving from a testing to a production environment) is re-running the original SQL creation script. Note that since the table objects are created in the new schema, they will be automatically journaled to the new journal.

Many times, however, recreating all of the DB2 objects is not feasible since the source objects contain a large amount of data. An alternative method to use in this case is to save and restore the schema and then manually reset the journal information afterwards. If I have a schema named ABC with two tables, DEPARTMENT and EMPLOYEE, then here are the steps that would need to be followed to move schema ABC into schema XYZ using the save and restore method:

1. Use the CL command `SAVLIB TEST ACCPTH(*YES)`.

Remember that the OS/400 container for a schema is a library. The `ACCPTH(*YES)` option saves the actual index tree if any indexes exist in the schema. That will eliminate indexes having to be rebuilt on the restore operation.

2. Create the production schema. Use the SQL statement `CREATE SCHEMA PROD`.

This will create the target schema object with auto-created journal and system catalog views.

3. Use the CL command to restore the library: `RSTLIB TEST OPTION(*NEW) RSTLIB(PROD)`.

The `*NEW` option will only restore the TEST objects that do not already exist in the PROD schema. This type of restore essentially restores everything but the objects automatically created by DB2 UDB (journal, journal receiver, and catalog views).

4. Find a the list of DB2 objects currently journaled in TEST schema.

Since the OS/400 journal CL commands only accept the short DB2 object identifiers, the short name for a DB2 UDB for iSeries object will need to be recorded in this step.

5. End journaling for all of the objects in TEST schema. Use the CL command `ENDJRNP *ALL TEST/QSQJRN`.

DB2 objects in the new schema, PROD, are currently associated with the original schema. Eliminate this association by ending journaling for all tables. If schema TEST does not exist on the system where schema PROD was restored, then the restore operation will end the journal association with the original schema. Thus, the `ENDJRNP *ALL` command is not needed when schemas TEST and PROD reside on different systems.

If other objects in the schema such as indexes have been explicitly journaled by the user, then journaling on those objects would have to be ended and restarted. For example, journaled indexes would be stopped and restarted with `ENDJRNP` and `STRJRNP` CL commands.

6. For each table in PROD schema you have to execute the following CL command:

```
STRJRNP PROD/DEPT PROD/QSQJRN IMAGES(*BOTH) OMTJRNE(*OPNCLO)
STRJRNP PROD/EMPLOYEE PROD/QSQJRN IMAGES(*BOTH) OMTJRNE(*OPNCLO)
```

The STRJRNPf currently does not support an \*ALL option like the ENDJRNPf command. So the command needs to be executed for each DB2 table object to associate these tables with XYZ journal and journal receiver objects.

**Note:** If the journal object was altered away from its default settings with the CHGJRN CL command or iSeries Navigator, those customizations would need to be performed on the journal object in the PROD schema.

7. Save the newly configured schema to backup media, so that you do not have to do these configuration steps again.

## 4.2 Comparing the SQL objects and the DDS files

When we create a table using SQL, under the covers a physical file is created. The same is true when we create an SQL index or a SQL view—a logical file is created under the covers. But there are some differences between SQL objects and DDS-created files that we need to understand. In this section we address some of the differences between DDS-created objects and SQL.

The new SQL objects reside in a schema. In 4.1.6, “Creating the new DB2 schema on the iSeries server” on page 40, we covered the differences between a library and a schema.

### 4.2.1 SQL tables compared with physical files

There are some advantages of SQL tables over physical files. They are:

- ▶ The constraint definitions can be included in the object source.
- ▶ SQL does faster reads than HLL reads to a physical file. The main reason is that a cursor reading an SQL table does not have the extra data cleansing code like a DDS PF reading.
- ▶ We can use longer and more descriptive column names.
- ▶ The data modeling tools have been made to support SQL.
- ▶ Automatically journaling if the tables are created in a schema, as we have stated before.

There are some disadvantages of SQL tables. These are:

- ▶ Slower writes, because the cursor created for SQL tables has more data validation code than a cursor used for writing into a DDS PF.
- ▶ No Distributed Data Management (DDM) support, but SQL can utilize Distributed Relational Database Architecture (DRDA®) connection by using SQL CONNECT statement.
- ▶ Multiple member files. An SQL table does not support multi-member files, as we have stated before. You can use an SQL ALIAS statement to create an alias for each physical file member, as the example:

```
CREATE ALIAS MYSCHEMA/JANSALES FOR SALES(JANUARY)
```

### 4.2.2 SQL indexes compared with keyed logical files

SQL indexes and keyed logical files both cause the same internal OS/400 object to be built. The main difference is that an SQL index can be either a binary radix tree structure or an advanced bit mapped index structure known as an Encoded Vector Index.

The advantages of SQL indexes are:

- ▶ Encoded Vector Index - An IBM solution for bitmap indexes. Realizing the limitations of traditional bitmap indexes technology, IBM Research set out to find a better solution. The result is Encoded Vector Indexes (EVIs), a patented indexing technology from IBM that has been available for several years in the iSeries. DB2 UDB for iSeries is the first member of the IBM DB2 family to provide EVIs.
- ▶ SQL indexes are created with a 64 K logical page size (since V4R2). Keyed logical files are primarily created with a logical page size of 8 K. This change to 64 K was made to improve the performance of queries that scan lots of key values in an index, since it brings more key values into memory. This attribute cannot be specified during object creation or subsequently changed. The logical page size is determined by the interface used to create the index. Regardless of the logical page size, the overall object sizes of a SQL index and a keyed logical file tend to be equivalent. The larger logical page size can result in more efficient index scans and index maintenance. This is a key benefit in a query environment. Indexes with larger logical page sizes can have an impact on the I/O performance within environments that have smaller, less than optimal memory pools.

The disadvantages of SQL indexes are:

- ▶ Single key lookups using an index may or may not be as efficient, because it takes longer to read a 64-K page than an 8-K page.
- ▶ SQL indexes do not have a way of supporting Select/Omit filtering options as logical files do. An SQL index cannot define a join operation like a logical file does.

### 4.2.3 SQL views compared with logical files

SQL views are equivalent to non-keyed logical files. SQL views have a strong advantage in that they have more functionality and flexibility to offer in terms of the processing and data manipulations that can be performed within the view definition versus a logical file.

The advantages of SQL views are:

- ▶ SQL views have more flexibility in terms of selecting and processing data.
  - CASE expressions and date/time functions
- ▶ The grouping and join processing offered by the SQL view is far superior to anything available on a logical file definition.
- ▶ The native program can open the views as logical files to enhance native functionality.

The disadvantages of SQL views are:

- ▶ SQL views cannot be keyed/ordered.
- ▶ An SQL view may not be able to replace multi-format logical files.

### 4.2.4 SQL data types

In defining the columns of a table with SQL:

- ▶ SQL supports more data types, such as large object (LOB), datalink columns, and user-defined types (UDTs).
- ▶ SQL supports column names up to 30 characters in length; and table, view, and index names up to 128 bytes in length. Longer SQL identifiers allow your object names and definitions to be more self-describing from a documentation point of view.

## 4.3 SQL system catalogs: Definitions

SQL cannot only be used to manipulate data but also to define databases, that is, creating, modifying, or dropping database tables or entire databases. There are five distinct object types involved with SQL databases:

- ▶ Schemas
- ▶ Aliases
- ▶ Tables
- ▶ Indexes
- ▶ Views

We have seen that a schema is a repository that contains SQL objects and that a schema corresponds to a library in the iSeries. In fact, each schema is of type \*LIB, that is, a library.

Creating a schema by itself would not make any sense. It is the starting point, a container destined to be filled with tables, indexes, etc., all the objects that make up your database and are at the heart of most applications.

But where is all the information about the database itself, the so-called metadata? Well, that is where the term *catalog* comes into the picture. Catalogs are automatically created when a schema is created, and they contain all the relevant information about the databases. Each modification of a table in a SQL schema (that is, creating, renaming, dropping, moving, etc.) a table updates the catalog files for that schema.

**Note:** Metadata is information about information. They serve to describe data, and their use is not limited to the field of SQL or information technology. Most, if not all, SQL-based RDBMS allow the extraction of the metadata of their content. For example, metadata is very important to reverse engineer databases. Database design tools typically use metadata to display database models.

To summarize, the following can be said: The structure of the database is maintained by the DBMS in special tables that are called catalogs. The catalogs can be queried by users or tools to display information about tables, columns, referential integrity constraints, security rights, and any other information that composes a database.

### iSeries catalog tables and views

The iSeries catalog includes the following views and tables in the QSYS2 schema.

DB2 UDB for iSeries name	Corresponding ANSI/ISO name	Description Information about...
SYSCATALOGS	CATALOGS	...relational databases
SYSCHKCST	CHECK_CONSTRAINTS	... check constraints
SYSCOLUMNS	COLUMNS	... column attributes
SYSCST	TABLE_CONSTRAINTS	... all constraints
SYSCSTCOL	CONSTRAINT_COLUMN_USAGE	... the columns referenced in a constraint
SYSCSTDEP	CONSTRAINT_TABLE_USAGE	... constraint dependencies on tables
SYSFUNCS	ROUTINES	... user defined functions

DB2 UDB for iSeries name	Corresponding ANSI/ISO name	Description Information about...
SYSINDEXES		... indexes
SYSJARCONTENTS		... jars for Java routines
SYSJAROBJECTS		... jars for Java routines
SYSKEYCST	KEY_COLUMN_USAGE	... unique, primary, and foreign keys
SYSKEYS		... index keys
SYSPACKAGE		... packages
SYSPARMS	PARAMETERS	... routine parameters
SYSPROCS	ROUTINES	... procedures
SYSREFCST	REFERENTIAL_CONSTRAINTS	... referential constraints
SYSROUTINES	ROUTINES	... functions and procedures
SYSROUTINEDEP	ROUTINE_TABLE_USAGE	... function and procedure dependencies
SYSSEQUENCES		... sequences
SYSTABLEDEP		... materialized query table dependencies
SYSTABLES	TABLES	... tables and views
SYSTRIGCOL	TRIGGER_COLUMN_USAGE	... columns used in a trigger
SYSTRIGDEP	TRIGGER_TABLE_USAGE	... objects used in a trigger
SYSTRIGGERS	TRIGGERS	... triggers
SYSTRIGUPD	TRIGGERED_UPDATE_COLUMNS	... columns in the WHEN clause of a trigger
SYSTYPES	USER_DEFINED_TYPES	... built-in data types and distinct types
SYSVIEWDEP	VIEW_TABLE_USAGE	... view dependencies on tables
SYSVIEWS	VIEWS	... definition of a view

### ODBC and JDBC catalog

The catalog includes the following views and tables in the SYSIBM library.

View name	Description
SQLCOLPRIVILEGES	Information about privileges granted on columns
SQLCOLUMNS	Information about column attributes
SQLFOREIGNKEYS	Information about foreign keys
SQLPRIMARYKEYS	Information about primary keys
SQLPROCEDURECOLS	Information about procedure parameters

View name	Description
SQLPROCEDURES	Information about procedures
SQLSCHEMAS	Information about schemas
SQLSPECIALCOLUMNS	Information about columns of a table that can be used to uniquely identify a row
SQLSTATISTICS	Statistical information about tables
SQLTABLEPRIVILEGES	Information about privileges granted on tables
SQLTABLES	Information about tables
SQLTYPEINFO	Information about the types of tables
SQLUDTS	Information about built-in data types and distinct types

### ANS and ISO catalog views

There are two versions of some of the ANS and ISO catalog views. These are not reprinted here. Refer to the *DB2 UDB for iSeries SQL Reference V5R3* manual that can be found in the information center at:

<http://publib.boulder.ibm.com/infocenter/series/v5r3/ic2924/info/db2/rbafzmst.pdf>

#### 4.3.1 SQL system catalogs: Example

In “Establishing a list of all DDS files to be converted” on page 31 we showed two examples of the use of the SQL catalog tables. Here we show another example of the use of these catalogs.

The marketing department of a company wants to increase the size of the customer number column CUSTNO and they want to find out in which tables the column is used. This is important to know to be able to do an impact analysis of this change. This example further assumes that there is not one single reference file for field definitions, but that the field may be contained in more than one file. The field might be named either CUSTNO or CUSTNR.

Using the SQL system catalogs we would execute the following SQL statement to find out how many files have the Customer Number field.

*Example 4-14 Querying the SQL system catalogs - Example 1*

---

```
SELECT COUNT(*) FROM QSYS2/SYSCOLUMNS
WHERE COLUMN_NAME LIKE 'CUST%'
```

---

Now, let us find out which are the files that need to be changed.

*Example 4-15 Querying the SQL system catalogs - Example 2*

---

```
SELECT TABLE_NAME FROM QSYS2/SYSCOLUMNS
WHERE COLUMN_NAME LIKE 'CUST%' ORDER BY TABLE_NAME
```

---

This will run the SQL statement and display the desired result, which is a list of all files that contain a column starting with CUST, sorted by table name.

## 4.4 Partitioned tables

DB2 UDB for iSeries V5R3 supports partitioned tables using SQL. Partitioning allows for the data to be stored in more than one member, but the table appears as one object for data manipulation operations such as select, insert, update, and delete. A partition is the equivalent of a database file member. Each partition can be saved, restored, exported from, imported to, dropped, or reorganized independently of the other partitions. You must have DB2 Multisystem (5722-SS1 option 27) installed on your iSeries server in order to take advantage of partitioned tables support.

There are two types of partitioning: *Hash partitioning* and *range partitioning*. You specify the type of partitioning with the PARTITION BY clause in the CREATE TABLE statement. In our example, we create a partitioned table PAYROLL in library PRODLIB with partitioning key EMPNUM in four partitions.

- ▶ Hash partitioning places rows at random intervals across a user-specified number of partitions and key columns.

```
CREATE TABLE PRODLIB.PAYROLL(EMPNUM INT, FIRSTNAME CHAR(15), LASTNAME CHAR(15),  
    SALARY INT)  
    PARTITION BY HASH(EMPNUM) INTO 4 PARTITIONS
```

- ▶ Range partitioning divides the table based on user-specified ranges of column values.

```
CREATE TABLE PRODLIB.PAYROLL  
    (EMPNUM INT, FIRSTNAME CHAR(15), LASTNAME CHAR(15), SALARY INT)  
    PARTITION BY RANGE(EMPNUM)  
    STARTING FROM (MINVALUE) ENDING AT (500) INCLUSIVE,  
    STARTING FROM (501) ENDING AT (1000) INCLUSIVE,  
    STARTING FROM (1001) ENDING AT (MAXVALUE)
```

However, as of the beginning of V5R3, the partitioned tables support cannot take advantage of the query optimizer for leveraging the performance advantages. The improvement will come in the future. The partitioned tables should really only be used in V5R3 if you have a table that is approaching the single size table limit of 4.2 billion rows or 1.7 terabytes of storage.

For more information on partitioned tables refer to the whitepaper *Table partitioning strategies for DB2 UDB for iSeries*, which can be found at:

[http://www-1.ibm.com/servers/enable/site/education/abstracts/2c52\\_abs.html](http://www-1.ibm.com/servers/enable/site/education/abstracts/2c52_abs.html)

Archived

# Data access

In our journey of database modernization we have already modernized our database definitions. We have also populated our new database objects and have tested the existing programs. Our next step is to modernize the data access.

In this section we explain:

- ▶ How to create I/O modules to access the new SQL objects
- ▶ How to start moving the business rules into the database
- ▶ How to take advantage of Embedded SQL to replace native I/O
- ▶ How to leverage and exploit the advantages of triggers, stored procedures, and user defined functions

Archived

## Creating I/O modules to access SQL objects

When modernizing your database the first step is to convert the DDS-based physical files to SQL-based tables. In this conversion we want to guarantee that the existing programs keep working with no changes. This was obtained in our first phase or stage of the methodology.

In this chapter, we move forward in the methodology and we propose the creation of SQL views and the use of service programs to replace the native I/O access. We cover the steps required in the second phase or stage proposed in our methodology in “Methodology for the modernization” on page 25.

In this chapter we cover the following topics:

- ▶ How you can benefit from the power of SQL views in RPG programs
- ▶ Creation of service programs to replace the I/O access

## 5.1 Introduction

The main goal of this stage is to minimize the impact of change on the business. This is achieved in two ways:

- ▶ By de-coupling the database access from the application program.
- ▶ By utilizing SQL views as the only way to access the data. Adding new columns to the database has no impact on existing views, thus eliminating the need to recreate the views and supporting programs. These views can be accessed via service programs or directly through ODBC or JDBC SQL statements.

The process involves a phased approach to replace native I/O operations with SQL data access methods. The strategy of using I/O modules is to limit the SQL optimization knowledge to the database programming group. This will allow the application programmers to focus on solutions to business requirements without a need to understand the complexities of database optimization.

The I/O module masks the complexity of the database from the application programmer. For example, an HLL program may be performing several read operations to multiple files to fill a subfile. This could be replaced by a single call to an I/O module that performs a single SQL fetch operation to a join view and returns a single host array (multiple occurrence data structure in RPG) to the caller.

In addition, the I/O module allows the database programmer to take advantage of database functions (that is, date and time data types, variable length fields, identity columns, etc.), thus eliminating many common HLL programming requirements. This includes programming required to format date and time data, formatting address lines, etc. Figure 5-1 illustrates the objective of this second stage.

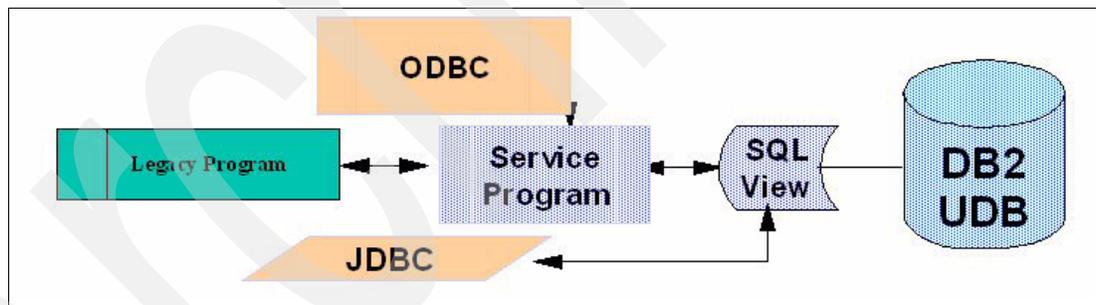


Figure 5-1 Stage 2 - I/O modules to access SQL objects

The following is an overview of the steps required in this stage:

1. Establish naming conventions.
2. Create SQL views based on business requirements.
3. Create service programs to access data from the SQL views.
4. Convert selected legacy programs to use service programs.

Let us start explaining the different steps of this stage.

## 5.2 Establish naming conventions

One of the biggest issues in naming objects on the iSeries is the 10-character limitation of the object name itself. Regardless of where the source resides you are still limited to 10 characters for the program and module objects. Because of the multitude of objects that can

be used to build an ILE program, some up-front planning is required to establish naming conventions.

The following is a list of some or all of the parts making up the I/O process:

- ▶ SQL table
- ▶ SQL view
- ▶ Service program object
- ▶ I/O module source and object
- ▶ Service program interface (prototype)
- ▶ Wrapper program for stored procedures(source and object)
- ▶ Binding directory object
- ▶ Binding source language
- ▶ Stored procedure to access the I/O module

The following are some suggested guidelines for establishing naming conventions:

1. Avoid using the object type as part of the object name. For example, do not use PGM, MOD, etc. as part of the name.
2. Establish standard abbreviations for the different database functions. There are basically four: Read, Write, Update, and Delete. Use these abbreviations to prefix the I/O module. For example, the I/O module that contains the procedures for reading the customer master table may be named GETCUSTMST. Keep in mind that you are limited to 10 characters.
3. Minimize calls to stored procedures by creating join and summary views and then creating a single procedure to access these views. For example, CustOrderSummaryByName is a view that groups on the customer name column and joins the customer master table to the customer Order Header table.
4. Use long names for SQL stored procedures. GetCustomerName says exactly what the stored procedure will do. Use the SPECIFIC clause in the CREATE PROCEDURE statement to control the short name.
5. Use the same name for both external stored procedures and HLL ILE procedure names.
6. Keep application abbreviations to two or three characters.
7. Use binding directories to link modules to programs and/or service programs.

**Note:** These suggestions are not intended to replace existing standards and conventions.

## 5.3 Create SQL views based on business requirements

Let us start by looking at the differences and highlighting the benefits of the use of views. SQL views are equivalent to non-keyed logical files. SQL views have a strong advantage in that they have more functionality and flexibility to offer in terms of the processing and data manipulations that can be performed within the view definition versus a logical file. The SQL views can be used to make the I/O access in the applications more easier.

The advantages of SQL views are:

- ▶ SQL views have more flexibility in terms of selecting and processing data.
  - CASE expressions and date/time functions
- ▶ The grouping and join processing offered by SQL views is far superior than anything available on a logical file definition.
- ▶ The native program can open the views as a logical files to enhance native functionality.

- ▶ Can be directly accessed via ODBC/JDBC. This is very useful for new/Web/java applications that access database objects on the iSeries server using ODBC or JDBC.
- ▶ Join views can mask the complexity of the database to the end users.
- ▶ Take advantage of the long name support. It can tell end users what they will get. For example, the CustOrderSummaryByName view name could mean that the view joins the customer master to the Order Header and grouped on name.
- ▶ SQL views can be used as well to solve some of the issues encountered in the first phase of the process such as replacing multi-format logical files.

The following example illustrates a joined view from the ORDERHDR table and the CUSTOMER table. The view presents a summary of the order amount grouped by the customer name.

```
CREATE VIEW ITS04710.CUSTORDERSUMMARYBYNAME (
  CUSTOMER_NAME, TOTAL_ORDER_AMT)
AS
SELECT CUSTOMER_NAME, SUM(ORDER_TOTAL) FROM ITS04710.ORDERHDR O, ITS04710.CUSTOMER C
WHERE O.CUSTOMER_NUMBER = C.CUSTOMER_NUMBER
GROUP BY CUSTOMER_NAME ;
```

To use views in RPG might be very useful, because they are more powerful than (join) logical files. In SQL views you can all use what is possible in a SQL select statement, with the exception of ordering rows.

### Accessing SQL views with native I/O

The following example is based on the Order Header File. The order total per year, based on the order date, has to be calculated and displayed.

To solve this problem a logical file keyed with Order Date is used. Example 5-1 shows the DDS described logical file ORDHDRL1.

*Example 5-1 DDS description for the keyed logical file ORDHDRL1*

---

A	R	ORDHDRF	PF	FILE(ORDHDR)
A	K	ORDHTE		
A	K	ORHNBR		

---

The following RPG snippet (Example 5-2) shows how the total per year can be calculated and displayed:

- ▶ The keyed logical file is read.
- ▶ The year portion must be extracted from the order date.
- ▶ The total for all orders in the same year must be added into a extra field.
- ▶ After having read all records with the same order year, the result is displayed, and the extra field is cleared.

*Example 5-2 RPG program to calculate and display the order totals per year*

---

F0rdHdrL1	IF	E	K	DISK	
*-----					
D	FirstRec	S		N	First Record
D	CompYear	S		4P 0	Compare Year
D	OrdYear	S		4P 0	Order Year
D	TotYear	S		+2 like(0rHTot)	Total/Year

```

D DspText          S          50A
*-----
/Free
  SetLL *Start OrdHdrL1;
  DoU %EOF(OrdHdrL1);

  Read OrdHdrF;
  If %EOF;
    If FirstRec = *On;
      Dsply DspText;
    EndIf;
    Leave;
  EndIf;

  OrdYear = %SubDt(OrdHdte: *Years); //Order Year

  If FirstRec = *Off;                //First Record
    CompYear = OrdYear;
    FirstRec = *On;
  EndIf;

  If OrdYear <> CompYear;            //Next Year
    Dsply DspText;
    Clear TotYear;
    CompYear = OrdYear;
  EndIf;

  TotYear += OrdHtot;                //Total per year
  DspText = %Char(CompYear) + ' ' + %Char(TotYear);

  EndDo;

  Return;
/End-Free

```

---

While logical files can only be used with native I/O, SQL views can be used with all databases and programming languages.

Example 5-3 shows how a SQL view can be created that summarizes all orders with the same order year.

*Example 5-3 SQL view to summarize order totals per year*

---

```

-- Create View OrdTotYrF (= format name)
Create View  ITS04710/OrdTotYrF
  (Order_Year   for OrdYear,
   Order_Total  for TotYear)
as Select    Year(Order_Date), Sum(Order_Total)
  from      Order_Header
  Group By  Year(Order_Date);

-- Rename View to Annual_Order_Total and System-Name to OrdTotYr
Rename table ITS04710/OrdTotYrF
  to Annual_Order_Total
  for System Name OrdTotYr;

```

---

Example 5-4 on page 62 shows how the RPG code from the previous example can be reduced by using the new SQL view instead of logical files:

- ▶ The SQL view is sequentially read.

- The total of the year is displayed.

*Example 5-4 Calculate and display the order totals per year by using a SQL view*

---

```

FOrdTotYr  IF   E           DISK
*-----*
D DspText      S           50A
*-----*
/Free
  SetLL *Start OrdTotYr;
  DoU %EOF(OrdTotYr);

  Read OrdTotYrF;
  If %EOF;
    leave;
  EndIf;

  DspText = %Char(OrdYear) + ' ' + %Char(TotYear);
  Dsply DspText;

EndDo;

Return;
/End-Free

```

---

**Note:** SQL views are never sorted in a predefined sequence. The query optimizer determines the access path that will be used to access the data.

SQL views can only be used within native I/O, when no predefined sequence is necessary. Alternatively, embedded SQL can be used, where an order-by sequence can be specified.

This is basically an iterative process. Begin by creating a view for each program and avoid creating views over all the columns of the physical files. Also keep in mind that views do not contain access paths, and thus do not add system maintenance overhead. Review the column requirements for each program and reuse the views as needed and appropriate.

## 5.4 Create service programs to access data from the SQL views

Once we have defined and created the required SQL views, the next step is to externalize the I/O modules. Externalizing the I/O means that all the native I/O operations (for example, CHAIN, READ, and WRITE) and other database operations are converted or coded into separate routines and programs that require I/O make requests to these routines to perform the operation on their behalf.

Externalizing I/O operations provides one way of helping to ensure that your applications can adapt quickly and relatively painlessly to changing business needs. Instead of coding a native READ, CHAIN, etc. at each point in the program where database access is required, you invoke a routine to perform the I/O for you.

An application may contain more than one service program. A *service program* is an Integrated Language Environment® (ILE) object that provides a means of packaging externally supported callable routines (functions or procedures) into a separate object.

There are basically four main database operations candidates to be replaced by I/O service programs. They are:

- Read (GET)

- ▶ Insert (PUT)
- ▶ Update
- ▶ Delete

Figure 5-2 illustrates the different modules that can be created.

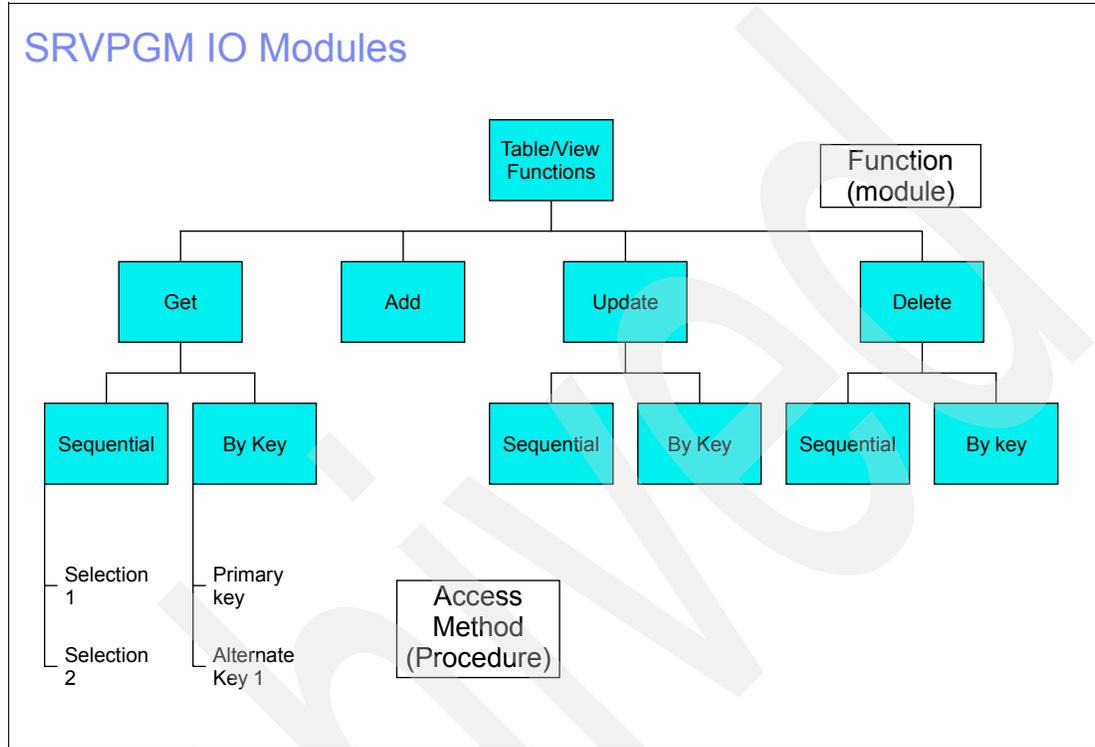


Figure 5-2 Service programs for I/O modules

## 5.5 Convert legacy programs to use service programs

This step involves replacing the HLL I/O operations (READ, READE, CHAIN, etc.) with a prototype call to the corresponding service program. The prototype is a definition of the call interface. It is used to call the program or procedure correctly. The proposed prototype call contains the following three parameters: Argument structure, record structure, and status structure.

- ▶ The argument is a data structure that corresponds to the key list used in keyed access operations. In addition, it may contain data items such as the number of rows requested and/or returned for blocked FETCH or INSERT requests.
- ▶ The record structure parameter contains one or more rows returned from read operations. For insert operations this parameter would contain one or more rows to be written. It is not required for delete operations.
- ▶ The status structure parameter contains an indication of the result of the operation. This would correspond to exceptions such as record not found, end of file reached, etc. The structure also contains SQL-related information (that is, SQL status codes, message text, etc.)

There is no need to replace every program. The candidates for SQL rewrite are:

- ▶ The programs that use OPNQRYF. Because the Query utility and OPNQRY do not use new SQL query engine (SQE), it will use the Classic Query Engine (CQE).
- ▶ The long-running batch programs that do massive reads or massive reads and writes.
- ▶ Online transaction processing (OLTP) programs that create subfile lists.
- ▶ OLTP programs that do a large numbers of inserts. It may be order entry programs that write several item records to complete a transaction.
- ▶ The programs that would benefit from join operations.
- ▶ The programs that do the record existence checking.



## Moving business rules to the database

The objective of this stage in the proposed methodology is to take advantage of DB2 data integrity functions and advanced functions that eliminate corresponding programming techniques and may replace application functions with equivalent database function.

By moving business rules into your database, you are assured that those requirements are enforced across all transactions and, more importantly, all interfaces. In contrast, business rules implemented in an application are enforced only when that application is used to change your database. Relying on application-enforced business rules opens up serious data integrity issues when data corrections are made using tools like SQL and new Web-based applications.

In this chapter we will explore additional DB2 UDB for iSeries features that will help us move the business rules into the database.

## 6.1 Database normalization

Normalization is the process of removing redundant data from your tables in order to improve storage efficiency, data integrity, and scalability. A table in a relational database is said to be in a certain normal form if it satisfies certain constraints. Edgar Codd's original definition defined three such forms, but there are now other forms accepted. Each normal form represents a stronger condition than the previous one, which means that a higher level of normalization cannot be achieved until the previous levels have been achieved.

The *First Normal Form* (or 1NF) involves removal of redundant data from horizontal rows. We want to ensure that there is no duplication of data in a given row, and that every column stores the least amount of information possible (making the field atomic). For example, normalization eliminates repeating groups by putting each into a separate table and connecting them with a primary key-foreign key relationship.

The *Second Normal Form* (or 2NF) deals with redundancy of data in vertical columns.

The *Third Normal Form* deals with looking for data in the tables that is not fully dependant on the primary key, but dependant on another value in the table. This is an ideal form for OLTP environments.

It is not within the scope of this book to explain the normalization process. The reality is that many of the iSeries customers have not taken the time and effort to normalize their databases. In this stage of the modernization process it would be a good idea to take some time and revisit the database design.

The steps involved in this stage are:

1. Eliminate unnecessary columns from the SQL tables.

In the process of normalization some unnecessary columns will be eliminated or moved to other tables. Many tables contain columns that were intended for some purpose, however, over the course of business these columns are no longer used or were never used at all. This is the opportunity to identify and remove these columns.

2. Establish a data dictionary.

A data dictionary was actually started in stage 1 when we were defining standard abbreviations for table names. The following is a list of some of the contents of the data dictionary:

- Object naming conventions
- Column naming conventions
- Function naming conventions
- Application naming conventions
- Standard abbreviations
- All existing database columns
- Object relationships
- Business rules

3. Establish data domains.

This is the process of grouping columns with like attributes into classes or domains. We can implement the data dictionary with established domains using Field Reference files, since the SQL CREATE TABLE statement can now reference this file.

4. Create a logical database model.
5. Implement the model.

## 6.2 Referential integrity

Referential integrity constraints (also known as RI constraints or referential constraints) implement business rules at the table level. An RI constraint would be used in DB2 UDB for iSeries to make sure that each new order references a customer that exists in the customer table. A referential constraint is defined for the order table to keep the child-parent (detail-master) data relationship in synch between the order and customer tables. With a referential integrity constraint in place between the two tables, DB2 UDB for iSeries guarantees that each order always refers to a valid customer.

Some of the benefits of using referential integrity are:

- ▶ It reduces application programming requirements by allowing the database to perform existence checks and cascaded delete functions.
- ▶ It prevents data corruption from sources outside the application such as ODBC, JDBC, DFU, SQL, etc.

The support of referential integrity in DB2 UDB for iSeries has been around for many years. The reality is that many customers have their referential integrity coded in their application code. In this phase it is important that combined with the normalization process of the database, it is a good time to implement referential integrity using the database. For a more detailed description of how to implement referential integrity in DB2 UDB for iSeries refer to the redbook *Advanced Functions and Administration on DB2 Universal Database for iSeries - SG24-4249-03*.

## 6.3 Constraints

DB2 UDB provides several ways to control what data can be stored in a column. These features are called *constraints* or *rules* that the database management enforces on a data column or set of columns.

DB2 UDB provides three types of constraints:

- ▶ **Primary and unique key constraints**  
Use these to prevent duplicate entries on a column from being entered into the database. Creating a primary key constraint on a table also provides equivalent functionality with a uniquely keyed physical file object.
- ▶ **Referential Integrity constraints**  
Use these to define relationships between tables and ensure that these relationships remain valid. For example, an RI constraint would prevent an order from being inserted whenever that order does not reference a valid customer number.
- ▶ **Check constraints**  
Use these to ensure that column data does not violate rules defined for the column or only a certain set of values are allowed into a column. The check constraint example here enforces that the order quantity amount is always greater than zero and less than 1000.

An example of creating a table and specifying the constraints is:

```
CREATE TABLE orders(  
  ordnum INTEGER PRIMARY KEY,  
  ordqty INTEGER CHECK(ordqty>0 AND ordqty<999),  
  ordamt DECIMAL(7,2),  
  part_id CHAR(4),  
  CONSTRAINT ordpart FOREIGN KEY(part_id) REFERENCES parts(PartID)
```

ON DELETE RESTRICT ON UPDATE RESTRICT )

Database constraints are beneficial due to the following reasons:

- ▶ They centralize the definition of business rules.
- ▶ Easier code re-use and better modularity.
- ▶ Improved data integrity.
- ▶ Improved query performance. SQE query optimizer is constraint aware.

The new SQE optimizer, starting with V5R3, also has the ability to analyze the constraint definitions to see if they can be employed to make data retrieval more intelligent and faster. For example, if a query is executed searching for all orders with a quantity greater than 1000, the optimizer would be able to return an empty result set without looking at any of the data since it knows that the check constraint definition prohibited any quantity value greater than 1000.

For a more detailed description of how to implement and define constraints in DB2 UDB for iSeries refer to the redbook *Advanced Functions and Administration on DB2 Universal Database for iSeries* - SG24-4249-03.

## 6.4 Constraint coexistence considerations

Before you begin using constraints to enforce business rules, you have to consider the impact on existing applications. This is true because the same business rules that constraints enforce generally already exist in application code. For example, a referential constraint might require that all orders have valid customer numbers. To enforce this business rule using RPG, before creating a new order you chain to the customer master file to verify that the customer number exists. If it does not, you display an error message and the order is not added.

To understand how constraints work with other application design options, you may find it helpful to consider how you currently apply duplicate key rules. It is common practice in iSeries shops to define duplicate key rules when a physical file is created. These rules ensure that no matter what application modifies the file, or what errors or malicious code that application contains, the database will never be corrupted with duplicate keys. This does not mean that your applications cease to check for duplicate keys, but it does mean that the database is protected even if your applications are bypassed or incorrect. Thus the duplicate key rule, which is a primary key or unique constraint in SQL terms, ensures that the file contains valid data regardless of how it is updated.

If you apply the same logic you currently use for imposing duplicate key rules to referential and check constraints, it is much easier to see how you can begin using these database features. We recommend that you begin defining referential and check constraints for business rules that are well defined and consistently enforced by your applications. Keep in mind that a constraint will always be enforced, so you want to avoid imposing constraints if the existing data contains violations, or if the applications do not conform to the constraint rules.

Once you have defined a constraint, the next question is how to deal with constraint violations in your applications. If you have an application that checks a business rule that is also enforced by a constraint, it is often best to leave that application code intact and accept the slight performance penalty incurred by checking twice—once in the constraint and once in the application code.

If you are writing new applications, consider checking for constraint violations instead of using data validation techniques, such as chaining to a master file. However, you may find that

there are situations where the existing methods of checking for errors make more sense than checking for constraint violations. For example, if you are already chaining to the customer master file to retrieve the customer name, it makes sense to handle invalid customer numbers at the same time, using the same methodology as you currently use.

Another issue to consider is how easily you can determine which constraint failed. SQL, RPG, and COBOL all signal constraint violations. However, if multiple constraints are assigned to a table, as is generally the case, you must retrieve the specifics concerning which constraint failed using the Receive Program Message API. In addition, constraint violations are reported as soon as the first violation is encountered. Therefore, if you need to validate an entire panel of information and report all errors to the user, checking for constraint violations in your application could be tricky.

Finally, while the duplicate key comparison works well for most constraints, some referential constraints do more than simply prevent invalid data from being stored in a table. If you define a constraint that cascades (for example, deleting all order line rows when the corresponding Order Header row is deleted), you will most likely want to remove any application code that performs the same function as the constraint.

Even if you decide never to check for constraint violations in your RPG or COBOL applications, you may still want to impose constraints. Doing so will make your business rules accessible to applications running on other platforms or written in languages such as Java. It will protect your data from corruption and it will improve application portability because constraints are a standard database capability.

## 6.5 Column-level security

You can restrict user update and read requests to specific columns of a table. There are two ways to do this:

- ▶ Create a view of the table that includes only those columns to which you want your users to have access, in the same way that it was done with logical files.
- ▶ Use the SQL GRANT/REVOKE statement to grant or revoke update authority to specific columns of an SQL table. This option is not available and possible using a logical file.

## 6.6 Column encryption

Starting with V5R3, DB2 UDB for iSeries now includes encryption and decryption column functions, so that iSeries developers do not have to write their own encryption routines. Underneath the covers, the DB2 Encrypt and Decrypt scalar functions utilize the IBM Cryptographic Access Provider 128-bit product (5722-AC3) to add another layer of security around your data.

There are specific data type and length requirements that must be met in order to use the Encryption column function. This is because the encrypted version of the data will be a binary value and longer than the original data string. The data types must be:

- ▶ BINARY, VARBINARY
- ▶ CHAR FOR BIT DATA, VARCHAR FOR BIT DATA
- ▶ BLOB

The length of an encrypted string value is the actual string length plus an extra 8 bytes (or 16, if BLOB or different CCSID values are used for the input) and must be rounded to an 8-byte

boundary. Up to 32 byte hint data can optionally be added and stored with an encrypted value.

In Example 6-1 you see how to set the encryption password with a 3-character hint and then the encryption of the employee 6-character ID value as it is inserted into a DB2 UDB table. The `decrypt_char` function on the `SELECT` statement uses the same password to return the original employee ID value of '112233' back to the application.

*Example 6-1 Encrypt and decrypt functions*

---

```
CREATE TABLE emp(id VARCHAR(19) FOR BIT DATA, name VARCHAR(50))
SET ENCRYPTION PASSWORD = 'protect' WITH HINT = 'sec'
INSERT INTO emp VALUES(ENCRYPT('112233'), 'BOB SANDERS' )
SELECT DECRYPT_CHAR(id), name FROM emp
```

---

On the iSeries server, a validation list object is a good container to safely store the encryption passwords, because the passwords can be encrypted when they are stored in the list object. Each validation list entry allows you to store an entry identifier along with an encrypted data value. The encryption password is stored in the encrypted data value and the list entry identifier could be assigned the table name or some other value that makes it easy to retrieve the encryption password for a specific column or row. A set of OS/400 APIs is provided for application programs to populate a validation list and retrieve values from the list.

### Native program access

The new column encryption functions (Encrypt and Decrypt) can be used in HLL programs in the following way:

- ▶ Encryption: Define and use an SQL before trigger to intercept the write request and then have the trigger execute the encrypt function against the sensitive columns.
- ▶ Decryption: Define an SQL view containing the decrypt function and then open the SQL view as a logical file to read unencrypted data.

## 6.7 Automatic key generation and unique identifiers

One of the simplest pieces of business logic that can be embedded into your DB2 object definitions is *key generation*. Almost all solutions have code that generates a key value for invoice or customer number and then inserts that value into a database table for storage. Why not just have DB2 UDB generate that value as it inserts the row into the table?

That is exactly the functionality that the Identity column attribute and Sequence object (new in OS/400 V5R3) can provide. Let DB2 UDB handle the key generation and locking/serialization of that value, so the programmer can concentrate on real business logic.

Using native I/O, the relative record number can be used to access exactly one selected record. SQL provides a scalar function `RRN (file)` to determine the relative record number; however, it is not possible to generate an index over the relative record number.

**Note:** When using the built-in function `RRN (file)` in SQL to get access on one specified relative record number, a table scan is always performed. SQL reads the complete table and does not even stop if the row with the appropriate record number is found.

To prevent SQL from executing a table scan, a column to hold the unique identifier must be added. Over this column an index can be built.

SQL provides three possibilities to generate unique identifiers:

- ▶ Identity column attribute
- ▶ Sequence object
- ▶ ROWID data type

### Identity column attribute

When using iSeries Navigator to create the table, select **Set as identity column** in the New Column window. The identity column must be a numeric data type such as INTEGER or DECIMAL. Start, minimal and maximal value, and increment step can be specified just like the action at overflow. Only one identity column is allowed for one table.

When a table has an identity column, the database manager can automatically generate sequential numeric values for the column as rows are inserted into the table.

**Note:** Tables containing a primary key with an identity column can be accessed by native file access. When writing a record or row through native I/O, the identity column value is generated and inserted.

Specify the identity column attribute when creating a table:

```
CREATE TABLE myschema/emp(empno INTEGER GENERATED ALWAYS AS IDENTITY
  (START WITH 10 , INCREMENT BY 10),
  name CHAR(30), dept# CHAR(4))
```

Insert a row into the table:

```
INSERT INTO myschema/emp(name,dept#) VALUES('MIKE','503A') or
INSERT INTO myschema/emp(name,dept#) VALUES(DEFAULT,'MIKE','503A')
```

### Sequence object

The sequence object allows automatic generation of values. Unlike an identity column attribute, which is bound to a specific table, a sequence object is a global and stand-alone object that can be used by any tables in the same database.

An example of creating a sequence object named ORDER\_SEQ is shown below:

```
CREATE SEQUENCE order_seq START WITH 10 INCREMENT BY 10
```

When inserting a row, the sequence number must be determined through NEXT VALUE FOR SEQUENCE. For example, we insert a row to the ORDERS table using a value from the sequence object:

```
INSERT INTO orders(ordnum,custnum) VALUES( NEXT VALUE FOR order_seq, 123 )
```

Because the sequence is an independent object and not directly tied to a particular table or column, it can be used with multiple tables and columns. Because of its independence from tables, a sequence can be easily changed through the SQL statement ALTER SEQUENCE. The ALTER SEQUENCE statement only generates or updates the sequence object, but it does not change any data.

### ROWID data type

A ROWID is a value that uniquely identifies a row in a table. A column or a host variable can have a ROWID data type. A ROWID column enables queries to be written that navigate directly to a row in the table. Each value in a ROWID column must be unique. The database manager maintains the values permanently, even across table reorganizations. When a row is inserted into the table, the database manager generates a value for the ROWID column

unless one is supplied. If a value is supplied, it must be a valid row ID value that was previously generated by either DB2 UDB for OS/390® and z/OS® or DB2 UDB for iSeries.

The internal representation of a ROWID value is transparent to the user. The value is never subject to CCSID conversion because it is considered to contain BIT data. ROWID columns contain values of the ROWID data type, which returns a 40-byte VARCHAR value that is not regularly ascending or descending.

A table can have only one ROWID. A row ID value can only be assigned to a column, parameter, or host variable with a ROWID data type. For the value of the ROWID column, the column must be defined as GENERATED BY DEFAULT or OVERRIDING SYSTEM VALUE must be specified. A unique constraint is implicitly added to every table that has a ROWID column that guarantees that every ROWID value is unique. A ROWID operand cannot be directly compared to any data type. To compare the bit representation of a ROWID, first cast the ROWID to a character string.

**Note:** In RPG, there is no data type that directly matches with the ROWID data type, but by using the keyword SQLTYPE in the Definition specifications, host variables can be defined to hold the ROWID.

The following example shows the definition of a host variable with the SQL Data type ROWID:

```
D MyRowId          S          SQLTYPE(ROWID)
```

## 6.8 Accessing non-relational data

While SQL was designed originally for relational database objects, it has been enhanced to also support non-relational objects. Examples of some non-relational objects would be image files, iSeries multi-format files, and S/36 files. These are all objects that SQL applications cannot access directly.

### 6.8.1 User defined table functions for accessing non-relational data

In V5R2, user defined table functions (UDTFs) were added and one of their advantages is they make it easier to access non-relational object. A UDTF can be written entirely in SQL or implemented as an external UDTF by using one of the high-level programming languages supported by OS/400.

If you have an existing OS/400 program that knows how to extract data out of a non-relational object (such as an IFS stream file, data queue, or S/36 file), the program can be registered as an external UDTF. Now, SQL programmers can have access to the data in these non-relational objects by just invoking the UDTF. A UDTF can be referenced anywhere on an SQL statement that a table reference is allowed.

Typically, SQL reads a S/36 record as a text field. In our example, we demonstrate manipulating S/36 data records by using UDF and UDTF. The record layout of the S/36 file, named S36EMP, is shown in Table 6-1.

Table 6-1 S/36 record layout

Field name	From	To
EMPLOYEE NO.	1	6
FIRST NAME	7	18

Field name	From	To
LAST NAME	19	33
BIRTH DATE	34	39

The 6-character birth date field is stored in the format year/month/day (YYMMDD). We create a UDF, named DMY, to convert this field into a date column with standard \*DMY format, as shown in Example 6-2.

*Example 6-2 User defined function to return \*DMY date*

---

```

CREATE FUNCTION SAMPLEDB01.DMY (DATEIN CHAR(6) )
  RETURNS DATE
  LANGUAGE SQL
  SPECIFIC SAMPLEDB01.DMY
  NOT DETERMINISTIC
  READS SQL DATA
  CALLED ON NULL INPUT
  NO EXTERNAL ACTION
  DISALLOW PARALLEL
  NOT FENCED
  BEGIN
    DECLARE DMY DATE ;
    DECLARE D CHAR ( 3 ) ;
    DECLARE M CHAR ( 3 ) ;
    DECLARE DM CHAR ( 6 ) ;
    SET D = CONCAT ( SUBSTR ( DATEIN , 5 , 2 ) , '/' ) ;
    SET M = CONCAT ( SUBSTR ( DATEIN , 3 , 2 ) , '/' ) ;
    SET DM = CONCAT ( D , M ) ;
    SET DMY = CAST ( CONCAT ( DM , SUBSTR ( DATEIN , 1 , 2 ) ) AS DATE ) ;
    RETURN DMY ;
  END ;

```

---

Example 6-3 shows you how to create a UDTF, named S36UDTF, which returns rows of a result from the S/36 data file. The user defined function, DMY, is also used in the SQL statement to convert the date field.

*Example 6-3 User-defined table function to return S/36 data as SQL table*

---

```

CREATE FUNCTION SAMPLEDB01.S36UDTF (EMPNO VARCHAR(6) )
  RETURNS TABLE (
    EMP_NO CHAR(6) ,
    FIRST_NAME CHAR(20) ,
    LAST_NAME CHAR(20) ,
    BIRTH_DATE DATE )
  LANGUAGE SQL
  SPECIFIC SAMPLEDB01.S36UDTF
  NOT DETERMINISTIC
  READS SQL DATA
  CALLED ON NULL INPUT
  NO EXTERNAL ACTION
  DISALLOW PARALLEL
  NOT FENCED
  BEGIN
  RETURN
    SELECT SUBSTR ( F00001 , 1 , 6 ) AS EMP_NO ,
           SUBSTR ( F00001 , 7 , 12 ) AS FIRST_NAME ,
           SUBSTR ( F00001 , 19 , 15 ) AS LAST_NAME ,
           SAMPLEDB01 . DMY( SUBSTR ( F00001 , 34 , 6 ) ) AS BIRTH_DATE

```

```

FROM SAMPLEDB01 . S36EMP
WHERE SUBSTR ( F00001 , 1 , 6 ) = EMPNO ;
END ;

```

Figure 6-1 shows how to execute the SQL statement that invokes the UDTF from the SQL Scripts window.

```
SELECT * FROM TABLE(S36UDTF('000010')) AS X;
```

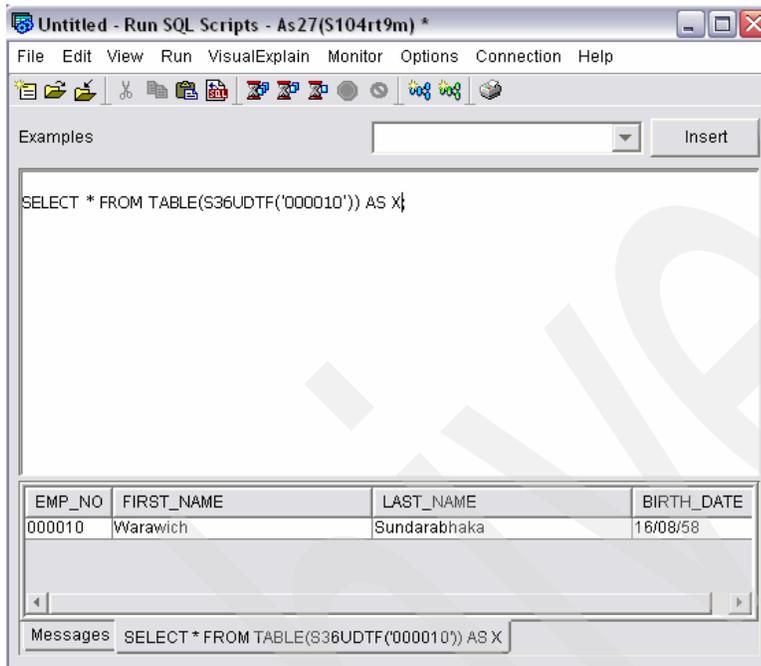


Figure 6-1 Invoking an UDTF

## 6.8.2 Datalink

The datalink data type provides a method of linking a row in a DB2 table with non-relational objects in the form of Uniform Resource Locators (URLs) that are associated with that row of data. For example, a row in the EMPLOYEE table might want a datalink to store the reference to the IFS file containing a photograph of an employee, as shown in Figure 6-2.

	PHOTO_FORMAT	PICTURE	DL_PICTURE
1	bitmap	424DAAA0000000000000...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200130.bmp">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200130.bmp</a>
2	gif	474946383961D200C900...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200130.gif">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200130.gif</a>
3	bitmap	424D7618010000000000...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200140.bmp">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200140.bmp</a>
4	gif	47494638396102011001...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200140.gif">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200140.gif</a>
5	bitmap	424DDE1E010000000000...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200150.bmp">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200150.bmp</a>
6	gif	47494638396102010E01...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200150.gif">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200150.gif</a>
7	bitmap	424D36F8000000000000...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200190.bmp">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200190.bmp</a>
8	gif	474946383961F1000001...	<a href="http://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200190.gif">HTTP://RCHASM27.ITSORCH.IBM.COM/QIBM/ProdData/OS400/SQL/Samples/db200190.gif</a>

Figure 6-2 Example of datalink data

The idea of a datalink is that the actual data stored in the column is only a pointer to the object. This object can be anything—an image file, a voice recording, a text file, and so on.

This means that a row in a table can be used to contain information about the object in traditional data types, and the object itself can be referenced using the datalink data type.

Using datalink also gives control over the objects while they are in *linked* status. A datalink column can be created such that the referenced object cannot be deleted, moved, or renamed while there is a row in the SQL table that references that object. This object is considered linked. Once the row containing that reference is deleted, the object is unlinked, but not deleted.

The maximum length of a datalink must be in the range of 1 through 32717. If FOR MIXED DATA or a mixed data CCSID is specified, the range is 4 through 32717. The specified length must be sufficient to contain both the largest expected URL and any datalink comment. If the length specification is omitted, a length of 200 is assumed.

A DATALINK value is an encapsulated value with a set of built-in scalar functions. The DLVALUE function creates a DATALINK value. The following functions can be used to extract attributes from a DATALINK value.

- ▶ DLCOMMENT
- ▶ DLLINKTYPE
- ▶ DLURLCOMPLETE
- ▶ DLURLPATH
- ▶ DLURLPATHONLY
- ▶ DLURLSCHEME
- ▶ DLURLSERVER

A datalink cannot be part of any index. Therefore, it cannot be included as a column of a primary key, foreign key, or unique constraint.

**Note:** It is not possible to create a host variable with an equivalent data type for DATALINK in RPG, neither through RPG data types nor by using the keyword SQLTYPE in the Definition specifications.

The datalink scalar functions, however, can be used in embedded SQL.

For more information on datalinks refer to the redbook *DB2 UDB for AS/400 Object Relational Support*, SG24-5409.

### 6.8.3 Large Object Support

Modern types of data have different attributes from traditional business data. For example, an application may need to store a graphical image, which is displayed on the PC, along with the other data types in the database. DB2 UDB for iSeries server has the capability to manage these types of data.

The VARCHAR, VARGRAPHIC, and VARBINARY data types have a limit of 32 K bytes of storage. While this may be sufficient for small to medium size text data, applications often need to store large text documents. They may also need to store a wide variety of additional data types such as audio, video, drawings, mixed text and graphics, and images. There are three data types to store these data objects as strings of up to two gigabytes in size.

- ▶ The character large object (CLOB) data type can store up to two gigabytes of variable-length character string. This data type is appropriate for storing text-oriented information where the amount of information can grow beyond the limits of a regular VARCHAR data type (upper limit of 32 K bytes). Code page conversion of the information is supported.

- ▶ The double byte character large object (DBCLOB) data type can store up to 1 gigabyte of variable-length double-byte character string. This data type is appropriate for storing text-oriented information where double-byte character sets are used. Again, code page conversion of the information is supported.
- ▶ The binary large object (BLOB) data type can store up to 2 gigabytes of variable-length binary string. A binary string is made up of bytes with no associated code page. This data type can store binary data larger than VARBINARY (32 K limit). This data type is good for storing image, voice, graphical, and other types of business or application-specific data.

Each table may have a large amount of associated LOB data. Although a single row containing one or more LOB values cannot exceed 3.5 gigabytes, a table may contain nearly 256 gigabytes of LOB data.

You can refer to and manipulate LOBs using host variables just like any other data type. However, host variables use the program's storage, which may not be large enough to hold LOB values. Other means are necessary to manipulate these large values. Locators are useful to identify and manipulate a large object value at the database server and for extracting pieces of the LOB value. File reference variables are useful for physically moving a large object value (or a large part of it) to and from the client.

### Large object locators

LOB locators use a small, easily managed value to refer to a much larger value. Specifically, a LOB locator is a 4-byte value stored in a host variable that a program uses to refer to a LOB value held in the database system. Using a LOB locator, a program can manipulate the LOB value as if the LOB value was stored in a regular host variable. When you use the LOB locator, there is no need to transport the LOB value from the server to the application (and possibly back again).

The LOB locator is associated with a LOB value, not a row or physical storage location in the database. Therefore, after selecting a LOB value in a locator, you cannot perform an operation on the original rows or tables that have any effect on the value referenced by the locator. The value associated with the locator is valid until the unit of work ends, or the locator is explicitly freed, whichever comes first. The FREE LOCATOR statement releases a locator from its associated value. In a similar way, a commit or rollback operation frees all LOB locators associated with the transaction.

When selecting a LOB value, you have three options:

- ▶ Select the entire LOB value in a host variable. The entire LOB value is copied into the host variable.
- ▶ Select the LOB value in a LOB locator. The LOB value remains on the server; it is not copied to the host variable.
- ▶ Select the entire LOB value in a file reference variable. The LOB value is moved to an Integrated File System (IFS) file.

For more information on datalinks refer to the redbook *DB2 UDB for AS/400 Object Relational Support*, SG24-5409.

## Embedded SQL

In our journey of database modernization we can use one of the powerful features of the iSeries, which is Embedded SQL. Embedded SQL is the capability to code SQL statements in programs written in RPG, COBOL, and C. Embedded SQL can be used to help us to modernize our data access to the database.

Some other reasons for using embedded SQL in RPG programs on the iSeries are:

- ▶ Programmers with SQL knowledge can understand RPG programs without learning native file operations.
- ▶ The same or a similar SQL code can be embedded in different programming languages.
- ▶ SQL provides a variety of scalar functions that does not exist in RPG but easily can be exploited.
- ▶ Take advantage of SQL scalar functions, user defined functions (UDF), and user defined table functions (UDTF), which can be used in SQL statements.
- ▶ Stored procedures can be called by using an SQL CALL statement.
- ▶ SQL can simplify the program logic when multiple rows are included in an operation, such as UPDATE or DELETE, or multiple joins are included in a single SQL statement.
- ▶ SQL provides much more powerful possibilities to join tables, like LEFT OUTER JOIN or EXCEPTION JOIN.
- ▶ A couple of column functions allows you to easily calculate totals, averages, minimums, and maximums.
- ▶ SQL allows you to merge data from several tables by using the UNION statement.
- ▶ SQL provides additional isolation levels and the SAVEPOINT statement that allows a partial ROLLBACK.

In this chapter we cover some considerations when using Embedded SQL, specially in RPG programs.

## 7.1 How to get started

Source code that contains an embedded SQL statement must first be processed by an SQL preprocessor. Its job is to replace SQL statements with calls to corresponding SQL function programs. This preprocessor is a part of the IBM licensed product DB2 Query Manager and SQL Development Kit for iSeries (5769-ST1), which must be available during the application development. DB2 SQL Development Kit is part of the Enterprise Edition. The runtime support is included in the operating system.

Source members with embedded SQL have a particular source type:

- ▶ SQLxxx for non-ILE sources, including embedded SQL, where xxx is the appropriate programming language. The member type for RPG/400® with embedded SQL is SQLRPG.
- ▶ SQLxxxLE for any ILE language including embedded SQL. The member type for ILE RPG with embedded SQL is SQLRPGLE.

In this chapter we focus exclusively on ILE RPG.

To embed SQL statements in your source code you have to consider the following rules:

- ▶ Enter your SQL statements on the C specification. SQL statements can only be used in classical RPG style. When you are coding in RPG free format, you have to end free format coding using the compiler directive/End-Free and restart if after the end of your SQL statement.
- ▶ Start your SQL statements using the delimiter /EXEC SQL in positions 7 through 15, with the slash (/) in position 7.
- ▶ You can start entering your SQL statements on the same line as the starting delimiter or on the new line.
- ▶ In the continuation line put C in position 6 and add a plus sign (+) in position 7 to continue your statements on any subsequent lines.
- ▶ SQL code can be inserted between position 8 and 80.
- ▶ Comments can be added elsewhere in the SQL statement, either through an asterisk (\*) in position 7 for the whole row or through an asterisk followed by a slash (/), and an ending asterisk followed by a slash (\*) for shorter comments.
- ▶ Use the ending delimiter /END-EXEC in positions 7 through 15, with the slash (/) in position 7, to signal the end of your SQL statements.
- ▶ Between /EXEC SQL and /END-EXEC only one SQL statement can be inserted. It is not possible to enter several SQL statements delimited by a semi colon (;) like in source files for RUNSQLSTM. However, you can insert multiple SQL statements starting with /EXEC SQL and ending with /END-EXEC each.

Example 7-1 shows an embedded SQL statement to delete rows in the Order Header table without deleting rows in the Order Details table.

*Example 7-1 Deleting Order Header without corresponding Order Details with SQL*

```
C/EXEC SQL
C+ Delete from Order_Header a
C+     where a.Order_Number in
C+         (Select  b.Order_Number
C+              from  Order_Header b
C+              exception join Order_Detail c
C+              on   b.Order_Number = c.Order_Number)
```

## 7.2 Creating a SQLRPG - Program/service program/module

Creating Integrated Language Environment (ILE) objects is always a two-step process:

1. The language compiler converts the source code and creates module objects.

RPG modules are created by the CL command CRTRPGMOD.

A module is sometimes called a compilation unit since it comes from the compilation of one source member. Modules are not executable. They must be built to either program or service program objects.

2. The binding process creates programs or service programs by binding the module objects.

A program is created by the CL command CRTPGM, while to create a service program the CL command CRTSRVPGM must be used. In this way it is possible to combine modules that are written in different programming languages into a single program or service program.

Modules can be bound by copy or by reference to the program objects. Bind by copy means that the module is physically copied into the program object. Every time a module is changed all programs that contain this module must be rebound. When using bind by reference, a module is bound into a service program and only the service program's signature is bound to the program object.

After a module is bound to a program or service program, the module object is no longer needed and can be deleted.

When using CRTBNDRPG, a module is generated in the QTEMP library and then bound to a program with the same name. If the compilation succeeds, the module will be deleted.

To compile an ILE Object with embedded SQL, an additional preceding step, the precompilation, is necessary.

The SQL precompile creates an output source file member. By default, the precompile process creates a temporary source file QSQLTxxxx in QTEMP, or you can specify the output source file as a permanent file name on the precompile command. If the precompile process uses the QTEMP library, the system automatically deletes the file when the job completes. A member with the same name as the program name is added to the output source file.

This member contains the following items:

- ▶ The embedded SQL statements are commented out and replaced by calls to the SQL run-time support.
- ▶ Parsed and syntax-checked SQL statement.
- ▶ SQL Communications Area (SQLCA) is added in the Definition specifications; if not, SET OPTION SQLCA = \*NO is specified.

By default, the precompiler calls the host language compiler. When creating a program CRTBNDRPG is used, while for modules and service programs CRTRPGMOD is used.

## 7.3 Compile command CRTSQLRPGI

If you want to generate RPG programs or modules, you have to use different compile commands. CRTRPGMOD creates a module object, while CRTBNDRPG creates a program object. To compile SQL program objects, there is only one single command, CRTSQLxxx (where xxx specifies the programming language.)

The object type that is created can be determined by the compile option OBJTYPE.

- ▶ Object type \*MODULE generates a module object.
- ▶ Object type \*PGM generates a program object.
- ▶ Object type \*SRVPGM generates a service program object.

**Note:** There is only one single command to generate programs, service programs, and modules with embedded SQL in RPG.

The object type can be specified by option OBJTYPE (Compile type).

### 7.3.1 Missing compile options in the SQL compile command

There are some compile options that are important for RPG compilation, but not supported by the SQL command CRTSQLRPGI:

- ▶ Activation group.

The CRTSQLRPGI command does not contain an option to determine the activation group.

- Programs are always compiled with the default activation group.
- Service programs are always compiled with activation group \*CALLER.

If you want to use different activation groups, especially for programs, you either have to use the keyword ACTGRP in the Control Specifications or you have to create the module with CRTSQLRPGI and then bind it with CRTPGM or CRTSRVPGM.

- ▶ Binding directories.

The CRTSQLRPGI command does not contain an option to specify binding directories, so it is not possible to bind several modules or service programs to the program object.

If you have to bind several modules, you either have to use the keyword BNDDIR in the control specifications or execute a two-step process by using CRTSQLRPGI to create the module first and then bind it with CRTPGM or CRTSRVPGM.

- ▶ Allow NULL values.

The CRTSQLRPGI command does not contain an option to allow NULL values.

If you use both embedded SQL and native I/O to access data, and the tables contain NULL values, you have to specify the keyword ALWNULL in the control specification.

Example 7-2 shows the control specification to implement compiler options in the control specifications (H-Specs).

*Example 7-2 Control specifications for compiler options*

---

```
H ActGrp('MYACTGRP')
H BndDir('QC2LE': 'QUSAPIBD': 'MYBNDDIR')
H AlwNull(*UsrCt1)
```

---

**Note:** Because there are no options to set activation group, binding directory, or allow null values in the SQL command, adequate keywords have to be entered in the control specification.

Compiler options in the control specifications are not overwritten by the compile command.

### 7.3.2 Important compile options for SQL statements

Additionally, the CRTSQLRPGI command has several options that are important for executing the SQL statements.

► Commit (COMMIT)

Specifies the isolation level to be used. The default value is \*CHG (the isolation level of uncommitted read). Only tables, views, and packages referred to in SQL statements are affected.

If your tables are not journaled, you have to change the value to \*NONE.

**Note:** If you want to use commitment control for native I/O in RPG, you have to specify the keyword COMMIT in the file definitions.

► Date format (DATFMT) and date separator character (DATSEP)

The default value for date format is \*JOB. Most the job date format only supports a 2-digit year, so the valid date range is between 1940-01-01 and 2039-12-31. This can cause problems in RPG, when dates out of this range are used. We recommend using a 4-digit year date format, like \*ISO.

**Note:** For fields that are defined in the Definition specifications or that are embedded in files that are defined in the File specifications, the date and time format and separators are determined by the keywords DATFMT and TIMFMT used in the definition or control specifications.

► Time format (TIMFMT) and time separator character (TIMSEP)

The default value for time format is \*HMS. Be careful when changing to \*USA, because RPG replaces the second portion of the date for AM or PM.

► Close SQL cursor (CLOSQLCSR)

This specifies when SQL cursors are implicitly closed, SQL prepared statements are implicitly discarded, and LOCK TABLE locks are released. SQL cursors are explicitly closed when you issue the CLOSE, COMMIT, or ROLLBACK (without HOLD) SQL statements.

The open data paths (ODP) are only closed when the cursors are implicitly closed. The ODPs can be reused as long as cursors are not implicitly closed. Because there is no overhead, by determining or creating the data path, repeated calls are much faster.

The default value is \*ENDACTGRP, which means that the cursors are implicitly closed at the end of the activation group.

► SQL path (SQLPATH)

This specifies the path to be used to find procedures, functions, and user-defined types in static SQL statements.

The default value is \*NAMING, which means that the path used depends on the naming convention specified for the Precompiler options (OPTION) parameter.

- For \*SYS naming, the path used is \*LIBL, the current library list at runtime.
- For \*SQL naming, the path used is "QSYS", "QSYS2", "userid", where "userid" is the value of the USER special register. If a schema name is specified for the Default Collection (DFTRDBCOL) parameter, the schema name takes the place of userid.

Alternatively, up to 268 libraries can be listed.

### 7.3.3 SET OPTION statement

The SET OPTION statement establishes the processing options to be used for SQL statements.

In RPG, compile options can be set in the control specifications by using keywords like ACTGRP, BNDDIR, ALWNULL, DATFMT, and TIMFMT. SQL does not consider the control specifications, but with the SET OPTION statement provides an equivalent.

The SET OPTION statement can be embedded elsewhere in the control specification. Only one SET OPTION statement is allowed per source member. Even if a source member consists of several independent procedures, the SET OPTION statement can only be embedded once, but it is valid for all procedures.

In SQL stored procedures, triggers, and User defined Functions, SET OPTION must be used to set the compiler options.

Example 7-3 shows the control specifications and a SET OPTION statement embedded in the same RPG source.

*Example 7-3 Coexistence of compiler options in H-Specification and SQL SET OPTION statement*

---

```
H ActGrp('MYACTGRP')
H BndDir('QC2LE': 'QUSAPIBD': 'MYBNDDIR')
H AlwNull(*UsrCtl)
H DatFmt(*Eur)
*-----
C/EXEC SQL
C+ Set Option Commit      = *NONE,
C+      C1oSQLCsr        = *ENDACTGRP,
C+      DatFmt           = *ISO,
C+      TimFmt           = *ISO
C/End-Exec
```

---

## 7.4 Error handling - SQLCA (SQL communications area)

Error handling is a key element of a good application design. In this section we cover the differences between and considerations for RPG and SQL.

RPG has several methods for dealing with errors. These are:

- ▶ Using the (E) extender in operation codes
- ▶ Using a monitor group
- ▶ Defining a \*PSSR routine
- ▶ Registering a Condition handler program

In SQL we have to work differently. In SQL, there are two variables used by the DBMS to return feedback that we must be familiar with: SQLCODE and SQLSTATE. SQLCODE is the original way in which DB2 reports error and warning conditions. Each DBMS provider developed its own error code structure, making it difficult to build portable code that manages error conditions. But in SQL92, the error conditions were standardized for all of us. That standardized error condition code is called SQLSTATE; now we have a platform-independent error code structure.

When DB2 UDB for iSeries encounters an error, the SQLCODE returned is negative, and the first two digits of the SQLSTATE are different from '00', '01', and '02'. If SQL encounters a warning, but it is a valid condition while processing the SQL statement, the SQLCODE is a positive number, and the first two digits of the SQLSTATE are '01' (warning condition) or '02' (no data condition). When the SQL statement is processed successfully, the SQLCODE returned is 0, and SQLSTATE is '00000'.

An SQL communication area (SQLCA) is a set of variables that may be updated at the end of the execution of every SQL statement. A program that contains executable SQL statements may provide one, but no more than one SQLCA (unless a stand-alone SQLCODE or a stand-alone SQLSTATE variable is used instead), except in Java, where the SQLCA is not applicable. Instead of using an SQLCA, the GET DIAGNOSTICS statement can be used in all languages to return codes and other information about the previous SQL statement.

The SQL precompiler automatically places the SQLCA in the Definition specifications of the ILE RPG for iSeries program prior to the first Calculation specification, unless a SET OPTION SQLCA = \*NO statement is found. Therefore, it is not necessary to code INCLUDE SQLCA in the source program.

If a SET OPTION SQLCA = \*NO statement is found, the SQL precompiler automatically places SQLCODE and SQLSTATE variables in the Definition specification. They are defined as shown in Example 7-4 when the SQLCA is not included.

*Example 7-4 Defining SQLCODE and SQLSTATE*

---

D SQLCODE	S	10I 0
D SQLSTATE	S	5A

---

The SQLCA source statements for ILE RPG for iSeries are as shown in Example 7-5.

*Example 7-5 SQLCA SQL communication area*

---

D SQLCA	DS	
D SQLCAID		8A INZ(X'0000000000000000')
D SQLAID		8A OVERLAY(SQLCAID)
D SQLCABC		10I 0
D SQLABC		9B 0 OVERLAY(SQLCABC)
D SQLCODE		10I 0
D SQLCOD		9B 0 OVERLAY(SQLCODE)
D SQLERRML		5I 0
D SQLERL		4B 0 OVERLAY(SQLERRML)
D SQLERRMC		70A
D SQLERM		70A OVERLAY(SQLERRMC)
D SQLERRP		8A
D SQLERP		8A OVERLAY(SQLERRP)
D SQLERR		24A
D SQLER1		9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER2		9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER3		9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER4		9B 0 OVERLAY(SQLERR:*NEXT)
D SQLER5		9B 0 OVERLAY(SQLERR:*NEXT)

D	SQLER6	9B 0	OVERLAY (SQLERR: *NEXT)
D	SQLERRR	10I 0	DIM(6) OVERLAY (SQLERR)
D	SQLWRN	11A	
D	SQLWN0	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN1	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN2	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN3	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN4	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN5	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN6	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN7	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN8	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWN9	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWNA	1A	OVERLAY (SQLWRN: *NEXT)
D	SQLWARN	1A	DIM(11) OVERLAY (SQLWRN)
D	SQLSTATE	5A	
D	SQLSTT	5A	OVERLAY (SQLSTATE)

**Note:** In Release V5R3M0 the SQLCA was redesigned and enhanced:

- ▶ Additionally to the short RPG field names like SQLCOD, longer field names, that are used in other languages like COBOL, are supported now.
- ▶ The numeric data type for the new fields with longer names is changed from binary data to integer.

To write a more standardized code we recommend using the longer field names.

Table 7-1 shows the enhancements of the SQLCA.

Table 7-1 SQLCA overview enhancements

Original Fields		Additional Fields	
Name	Definiton	Name	Definition
SQLAID	8A	SQLCAID	8A
SQLABC	9B 0	SQLCABC	10I 0
SQLCOD	9B 0	SQLCODE	10I 0
SQLERL	4B 0	SQLERRML	5I 0
SQLERM	70A	SQLERRMC	70A
SQLERP	8A	SQLERRP	8A
SQLERR	24A	SQLERRD	10I 0 Dim(6)
SQLWRN	11A	SQLWARN	1A Dim(11)
SQLSTT	5A	SQLSTATE	5A

The SQLCODE (SQLCOD) and SQLSTATE (SQLSTT) values are set by the database manager after each SQL statement is executed. If the SQLCA is used, a program should check either the SQLCODE or SQLSTATE value to determine whether the last SQL statement was successful.

## 7.4.1 SQLCODE

An SQLCODE is a return code. The return code is sent by the database manager after completion of each SQL statement.

Each SQLCODE that is recognized by a DB2 UDB for iSeries server has a corresponding message in the message file QSQLMSG. The message identifier for any SQLCODE is

constructed by appending the absolute value (5 digits) of the SQLCODE to SQ and changing the third character to 'L' if the first character of the SQLCODE is '0'. For example, if the SQLCODE is 30070, the message identifier is SQ30070. If the SQLCODE is -0204, the message identifier is SQL0204. Lastly, if the SQLCODE is a three-digit positive number, a zero is added before the first digit. For example, if the SQLCODE is 551, the message identifier is SQL0551.

If the error message text contains variables, the appropriate variable texts are returned in the field SQLERRMC (SQLERM). To get the complete message text, you only have to use Application Programming Interface (API) QMHRTVM (Retrieve Message) or CL command RTVMSG (Retrieve Message).

**Note:** If you use a cursor to read your rows, do not use SQLCODE = \*Zeros to detect if a row was returned. In some cases SQL warnings are returned (SQLCODE between 1 and 99), but the row is nevertheless retrieved. It is better to use SQLCODE <> 100 or SQLSTATE <> '02000' instead.

## 7.4.2 SQLSTATE

SQLSTATE provides application programs with common return codes for success, warning, and error conditions found among the DB2 Universal Database products. SQLSTATE values are particularly useful when handling errors in distributed SQL applications. SQLSTATE values are consistent with the SQLSTATE specifications contained in the SQL 1999 standard.

In SQL functions, SQL procedures, SQL triggers, and embedded applications other than Java, SQLSTATE values are returned in the following way:

- ▶ The last five bytes of the SQLCA
- ▶ A stand-alone SQLSTATE variable
- ▶ The GET DIAGNOSTICS statement

SQLSTATE values are comprised of a two-character class code value, followed by a three-character subclass code value.

The class code of an SQLSTATE value indicates whether the SQL statement was executed successfully (class codes 00 and 01) or unsuccessfully (all other class codes).

SQLSTATE is related to SQLCODE. Every SQLSTATE has one or more SQLCODEs associated with it. An SQLSTATE can refer to more than one SQLCODE.

Table 7-2 compares SQLCODE and SQLSTATE.

Table 7-2 Comparing SQLCODE and SQLSTATE

	SQLCODE / SQLCOD	SQLSTATE / SQLSTT
Error	< *Zeros	Position 1-2 <> '01', '02', '00'
Not Found	= 100	'02000'
Warning	between 1 and 99	Position 1-2 = '01'

**Note:** SQLCODE is the original way in which DB2 reports error and warning conditions, but the SQL standard standardizes the SQLSTATE; that is why SQLSTATE should be preferred.

For more information on error handling in SQL refer to the redbook *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503.

## 7.5 Host variables

Sometimes you like to use different values in your SQL statement instead of literal values, to make your application more flexible and to process different rows in a table.

SQL allows you to embed such variables called host variables. A host variable in an SQL statement must be identified by a preceding colon (:).

We can differentiate between:

- ▶ Single field host variable
- ▶ Host structure
- ▶ Host structure array

### 7.5.1 Single field host variable

A host variable is a single field in your program that is specified in an SQL statement, usually as the source or target for the value of a column. Every field defined in your source code can be used as host variable. It makes no difference if the fields are defined in the File, Definitions, or Calculations Specifications. The host variable and column must be data type compatible. For more information about type compatibility look at 9.1, “Comparing RPG and SQL data types” on page 172.

**Note:** Array elements cannot be used as host variables.

Host variables are commonly used in SQL statements in these ways:

- ▶ In a *WHERE clause* of a select statement
- ▶ As a *receiving area for column values*
- ▶ As a listed column in a SELECT statement
- ▶ As a value in *other clauses of an SQL statement*:
  - In the SET clause of an UPDATE statement
  - In the VALUES clause of an INSERT statement
  - As parameters in the SQL CALL statement

#### Using host variables in a WHERE clause

You can use a host variable to specify a value in the predicate of a search condition, or to replace a literal value in an expression.

Example 7-6 shows how the delivery date in the Order Header table can be updated with the current date for a range of order numbers ('00005'-'00010').

*Example 7-6 Using host variables in a WHERE clause*

---

```
D StartOrderNo   S           5A
D EndOrderNo     S           5A
*-----*
C                eval      StartOrderNo = '00005'
C                eval      EndOrderNo   = '00010'

C/EXEC SQL
C+ Update Order_Header
C+ set Order_Delivery = Current Date
C+ where Order_Number between :StartOrderNo and :EndOrderNo
C/End-Exec
C                Return
```

---

## Using Host variables as a receiving area for column values

Host variables can be used in an SELECT ... INTO or FETCH clause, to retrieve values from the return table.

In Example 7-7 the customer number and the order total for order number '00005' are returned into the host variables Customer and Total.

### Example 7-7 Using host variables as a receiving area

---

```
D OrderNo      S          5A
D Customer     S          5A
D Total        S          11P 2
*-----*
C              eval      OrderNo   = '00005'

C/EXEC SQL
C+ Select Customer_Number, Order_Total
C+   into :Customer, :Total
C+   from Order_Header
C+   where Order_Number = :OrderNo
C/End-Exec

C              Return
```

---

## Using host variables as a value in a SELECT clause

When specifying a list of items in the SELECT clause, you are not restricted to the column names of tables and views.

Example 7-8 shows how the total amount can be raised by using a host variable.

### Example 7-8 Using host variables as a value in a select clause

---

```
D OrderNo      S          5A
D Raise        S          11P 2
D Total        S          11P 2
D NewTotal     S          11P 2
*-----*
C              eval      OrderNo   = '00005'
C              eval      Raise     = 100

C/EXEC SQL
C+ Select Customer_Number, Order_Total, Order_Total + :Raise
C+   into :Customer, :Total, :NewTotal
C+   from Order_Header
C+   where Order_Number = :OrderNo
C/End-Exec

C              Return
```

---

## Using host variables in the SET clause of an SQL statement

Example 7-9 shows how the order total can be raised by using a host variable.

### Example 7-9 Using host variables in a SET clause of an SQL statement

---

```
D OrderNo      S          5A
D Raise        S          5P 2
*-----*
```

```

C          eval      OrderNo      = '00005'
C          eval      Raise        = 200

C/EXEC SQL
C+ Update Order_Header
C+ set   Order_Total = Order_Total + :Raise
C+ where Order_Number = :OrderNo
C/End-Exec

C          Return

```

---

### Using host variables in the VALUES clause of an INSERT statement

Example 7-10 shows how a row can be inserted by using host variables in the VALUES clause.

*Example 7-10 Using host variables in the values clause of an insert statement*

```

D OrderNo      S          5A
D Customer     S          5A
*-----*
C          eval      OrderNo      = '10010'
C          eval      Customer     = '00010'

C/EXEC SQL
C+ Insert into Order_Header
C+          (Order_Number,
C+          Customer_Number)
C+ values (:OrderNo,
C+        :Customer)
C/End-Exec

C          Return

```

---

### Using host variables as parameters in the CALL statement

When calling a stored procedure, parameters can be passed as host variables.

Example 7-11 shows how a stored procedure can be called using host variables as parameters.

*Example 7-11 Using host variables as parameters in the CALL statement*

```

D OrderNo      S          5A
D Customer     S          5A
*-----*
C          eval      OrderNo      = '10010'
C          eval      Customer     = '00010'

C/EXEC SQL
C+ CALL calcTotals(:OrderNo, :Customer)
C/End-Exec

C          Return

```

---

## 7.5.2 Host structure

A host structure is a group of host variables used as the source or target for a set of selected values (for example, the set of values for the columns of a row). A host structure is defined as an internally or externally described data structure in your source code.

Host variables are commonly used in SQL statements as a receiving area for column values.

Host structures can be used in an SELECT ... INTO or FETCH clause. The INTO clause names one or more host variables that you want to contain column values returned by SQL.

**Note:** When using host variables for each variable a separate pointer must be returned. When using host structures only one pointer is returned. That could be a performance gain.

Example 7-12 shows how an external data structure can be used to receive the complete row.

*Example 7-12 Receiving SQL data in a external data structure*

---

D DsOrdHdr	E DS	ExtName(OrdHdr)
*-----		

```
C/EXEC SQL
C+ Select *
C+   into :DSOrdHdr
C+   from Order_Header
C+   where Order_Number = '00020'
C/End-Exec

C                               Return
```

---

## 7.5.3 Host structure array

A host structure array is defined as a multi occurrence data structure or an array data structure. Both types of data structures can be used on the SQL FETCH or INSERT statement when processing multiple rows. The following list of items must be considered when using a data structure with multiple row blocking support.

- ▶ All subfields must be valid host variables.
- ▶ All subfields must be contiguous. The first FROM position must be 1, and there cannot be overlaps in the TO and FROM positions.
- ▶ If the date and time format and separator of date and time subfields within the host structure are not the same as the DATFMT, DATSEP, TIMFMT, and TIMSEP parameters on the CRTSQLRPGI command (or in the SET OPTION statement), then the host structure array is not usable.

Blocked FETCH and blocked INSERT are the only SQL statements that allow an array data structure. A host variable reference with a subscript like MyStructure(index).MySubfield is not supported by SQL.

**Note:** To use blocked processing brings performance advantages because only one single pointer must be returned for a group of rows.

Host variables are commonly used in SQL statements in these ways:

- ▶ As a *receiving area for column values* to receive multiple rows in a single fetch

Example 7-13 shows how a number of rows can be read into an array data structure.

*Example 7-13 Receiving SQL data into a host structure array*

```

D DSOrderHeader E DS                               ExtName(OrdHdr) Qualified Dim(3)
-----
D OrderNo      S                               5A
D Elements     S                               3U 0 inz(%Elem(DSOrderHeader))
D Index        S                               3U 0
*-----
C/EXEC SQL
C+ Declare CsrOrdH Cursor for
C+ Select *
C+   from Order_Header
C+  where Order_Number between '00005' and '00050'
C+ for read only
C+ optimize for 100 rows
C/End-Exec

C/EXEC SQL Open CsrOrdH
C/END-EXEC

C/EXEC SQL
C+ Fetch next from CsrOrdH
C+   for :Elements rows
C+   into :DSOrderHeader
C/END-EXEC

C/EXEC SQL Close CsrOrdH
C/END-EXEC

/Free
  For Index = 1 to %Elem(DSOrderHeader);
    Dsply DSOrderHeader(Index).OrHNbr;
  EndFor;

  Return;
/End-Free

```

- ▶ To *insert multiple rows* in a table

Example 7-14 shows how to insert multiple rows by using an array data structure.

*Example 7-14 Insert multiple rows in a table*

```

D DSOrderHeader E DS                               ExtName(OrdHdr) qualified Dim(3)
-----
D OrderNo      S                               5A  inz('10040')
D Elements     S                               3U 0 inz(%Elem(DSOrderHeader))
D Index        S                               3U 0
*-----
/Free
  cclear DSOrderHeader;
  For Index = 1 to Elements;
    OrderNo = %EditC(%Dec(%Int(OrderNo) + 10: 5: 0): 'X');
    DSOrderHeader(Index).OrHNbr = OrderNo;
    DSOrderHeader(Index).CusNbr = '00010';
    DSOrderHeader(Index).OrHDte = %Date();
  EndFor;
/End-Free

```

```

        DSOrderHeader(Index).OrHD1y = %Date() + %Days(7);
    EndFor;
/End-Free

C/EXEC SQL
C+ Insert into Order_Header
C+      :Elements Rows
C+      values (:DSOrderHeader)
C/End-Exec

C          Return

```

---

## 7.5.4 Naming considerations for host variables

Any valid ILE RPG for iSeries variable name can be used for a host variable with the following restrictions.

- ▶ Do not use host variable names or external entry names that begin with the characters 'SQ', 'SQL', 'RDI', or 'DSN'. These names are reserved for the database manager.
- ▶ The length of host variable names is limited to 64, except in RPG field names can be defined with up to 4 096 characters.
- ▶ The names of host variables should be unique within the source code of the member. If the same name is used for more than one variable and the data types are different, the data type of the host variable is unpredictable. This must be considered when using local field definitions in RPG procedures. You do not have to define your host variables as global variables, but the names should be unique in your source.

However, if a data structure has the QUALIFIED keyword, then the subfields of that data structure can have the same name as a subfield in a different data structure or as a stand-alone variable. The subfield of a data structure with the QUALIFIED keyword must be referenced using the data structure name to qualify the subfield name.

## 7.6 Exploiting SQL scalar functions in RPG

To embed SQL statements in RPG means not only to get data access, but you easily can use SQL scalar functions to change a host variable.

Currently RPG provides about 75 built-in functions, while SQL has about 120 scalar functions. There are some domains where SQL delivers other, additional or better functions than RPG does and vice versa. The following list contains functions that may be useful, but are not available in RPG:

- ▶ String functions
  - UPPER/LOWER: To convert a string in either upper or lower case
  - HEX: Returns the hexadecimal representation of a string
  - REPEAT: Returns a string composed of expression repeated integer times
  - REPLACE: To replace all occurrences of a search string with a new string
  - LEFT / RIGHT: Returns the left- or rightmost characters of a string
  - SOUNDEX: To compare a string on a phonetical base
  - DIFFERENCE: Returns a value from 0 to 4 representing the difference between the sounds of two strings based on applying the SOUNDEX function to the strings
  - SPACE: Returns a number of SBCS \*BLANKS

- TRIM: To remove any desired leading and/or trailing character
- ▶ Mathematical functions
  - ANTILOG/LOG10: Returns the (anti-)logarithm (base 10) of a number
  - EXP: Returns a value that is the base of the natural logarithm (e) raised to a power specified by the argument
  - LN: Returns the natural logarithm of a number
  - SQRT: Returns the square root of a number
  - CEILING/CEIL: Or CEILING function returns the smallest integer value that is greater than or equal to expression
  - FLOOR: Returns the largest integer value less than or equal to expression
- ▶ Trigonometric functions
  - ACOS/COS: Returns the (arc) cosine of the argument as an angle expressed in radians
  - COSH: Returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians
  - ASIN/SIN: Returns the (arc) sine of the argument as an angle expressed in radians
  - SINH: Returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians
  - ATAN/TAN: Returns the (arc) tangent of the argument as an angle expressed in radians
  - ATANH/TANH: Returns the hyperbolic (arc) tangent of a number, in radians
  - ATAN2: Returns the arc tangent of x and y coordinates as an angle expressed in radians
  - COT: Returns the cotangent of the argument, where the argument is an angle expressed in radians
  - PI: Returns the value of 3.141592653589793
  - RADIANS: Returns the number of radians for an argument that is expressed in degrees
  - DEGREES: Returns the number of degrees of the argument, which is an angle expressed in radians
- ▶ Date and time functions
  - DAYNAME: Returns a mixed-case character string containing the name of the day.
  - MONTHNAME: Returns a mixed-case character string containing the name of the month.
  - DAYOFWEEK: Returns an integer between 1 and 7 that represents the day of the week, where 1 is Sunday and 7 is Saturday.
  - DAYOFWEEK\_ISO: Returns an integer between 1 and 7 that represents the day of the week, where 1 is Monday and 7 is Sunday.
  - DAYOFYEAR: Returns an integer between 1 and 366 that represents the day of the year where 1 is January 1.
  - JULIAN\_DAY: Returns an integer value representing a number of days from January 1, 4713 B.C. (the start of the Julian date calendar) to the specified date.

- MIDNIGHT\_SECONDS: Returns an integer value that is greater than or equal to 0 and less than or equal to 86 400 representing the number of seconds between midnight and the time value specified in the argument.
  - QUARTER: Returns an integer between 1 and 4 that represents the quarter of the year.
  - TIMESTAMPDIFF: Returns an estimated number of intervals, which can be years, quarters, months, weeks, days, hours, minutes, seconds, or mircoseconds.
  - WEEK: Returns an integer between 1 and 54 that represents the week of the year. The week starts with Sunday, and January 1 is always in the first week.
  - WEEK\_ISO: Returns an integer between 1 and 53 that represents the week of the year. The week starts with Monday. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4.
- Other functions
- MAX()/MIN(): Returns the maximum/minimum value in a set of values

**Note:** While the SQL scalar function REPLACE replaces all occurrences, the RPG-Function %REPLACE only replaces the first one. The RPG-Function %REPLACE is an equivalent to the SQL scalar function INSERT.

For more information about SQL scalar functions look at *iSeriesDB2 Universal Database for iSeries SQL Reference* manual that can be found in information center at:

<http://publib.boulder.ibm.com/infocenter/iseriess/v5r3/ic2924/info/db2/rbafzmst.pdf>

## SQL set statement

The SQL set statement can be compared with the RPG Operations Code EVAL. It can be used to change the value of a host variable.

Example 7-15 shows how the scalar function replace can be used to remove characters in an string.

*Example 7-15 Using the scalar function REPLACE to remove characters from a string*

---

```
D MyText          S          50A  inz('ABC-XYZ-1234-567890-A')
*-----
C/EXEC SQL
C+ Set :MyText = Replace(:MyText, '-', '')
C/End-Exec

C   MyText          Dsply
C                                     Return
```

---

Example 7-16 shows how the scalar function trim can be used to remove leading asterisks (\*).

*Example 7-16 Using the scalar function TRIM to remove leading characters*

---

```
MyText2          S          50A  inz('*****123.45')
*-----
C/EXEC SQL
C+ Set :MyText2 = Trim(leading '*' from :MyText2)
C/End-Exec

C   MyText2          Dsply
```

## 7.7 Static SQL without cursor

Because the use of SQL-described tables and views in native I/O has some restrictions (that is, they cannot be sorted), we must look for an alternate method.

With static and dynamic SQL you can embed SQL statements into your source code.

In static SQL, the statement is determined at compile time. All SQL scalar functions can be used in the embedded SQL statements. You can integrate host variables, which are set at runtime. The syntax is checked by the precompiler and then the SQL statements are replaced by adequate function calls.

Static SQL is commonly used in these ways:

- ▶ To return one single row from a select statement into host variables
- ▶ To insert, update, or delete several rows using one single SQL statement
- ▶ For other actions such as:
  - To declare global temporary tables
  - To create and drop temporary aliases
  - To grant temporary privileges
  - To set path or set schema

### 7.7.1 Static SQL returning a single row

If the result of an SQL statement will be only one row, it can be directly returned into host variables in one of the following manners:

- ▶ SELECT ... INTO
- ▶ SET HostVariable = SELECT ...

#### SELECT ... INTO

The SELECT INTO statement produces a result table consisting of at most one row, and assigns the values in that row to host variables.

- ▶ If a single row is returned, SQLCODE is set to 0 or a value between 1 and 99, and the appropriate values are assigned to the host variables.
- ▶ If the table is empty, the statement assigns +100 to SQLCODE and '02000' to SQLSTATE. If you do not use indicator variables to detect NULL values, the host variables are *not* updated; otherwise NULL is returned.
- ▶ If the result consists of more than one row, SQLCODE -811 is returned, but the host variables are updated with the results from the first row.

Example 7-17 shows how the total amount of an order is calculated and returned within a SELECT ... INTO statement.

*Example 7-17 Using SELECT ... INTO to retrieve summary values*

```
D TotalDetail      S              11P 2
*-----
C/EXEC SQL
C+ Select Sum(OrderDt1_Total)
C+   into :TotalDetail
C+   from Order_Detail
```

```
C+ where Order_Number = '00020'
C/End-Exec
```

```
C TotalDetail Dsply
```

```
C Return
```

---

### SET :HostVariable = (SELECT ... )

It is possible to fill host variables directly through a select statement. This can only be used when the result consists only of one single record.

In contrast to the SELECT ... INTO statement, SQLCODE and SQLSTATE cannot be used to check if a record is found. If no record is found, NULL values are returned by default. You either have to use indicator variables to detect NULL values or an SQL scalar function like COALESCE that converts the NULL value into a default.

If the result consists of more than one row, SQLCODE -811 is returned, but in contrast to the SELECT ... INTO statement, the host variables are not updated.

The following example shows how the total amount of an order is calculated and returned within a SET-Statement.

*Example 7-18 Using the SET statement to retrieve summary values*

---

```
D TotalDetail S 11P 2
*-----
C/EXEC SQL
C+ Set :TotalDetail = (Select Sum(OrderDtl_Total)
C+ from Order_Detail
C+ where Order_Number = '00020')
C/End-Exec

C TotalDetail Dsply
C Return
```

---

## 7.7.2 Processing non-Select statements with static SQL without cursor

A 100 percent normalized database is a utopia. In most databases you will find a certain degree of denormalization, which leads to some redundancies in the tables. Consequently, you have to sometimes update several rows with the same value.

There are also other situations where you have to insert and delete a couple of rows. For example, if you have to reorganize your tables. You write rows to history tables and delete the original rows after.

In Example 7-20 on page 96, written with native I/O, all Order Header and corresponding Order Detail rows with the order date of the previous year must be saved in history files and deleted after.

The logical file over the Order Header file is described in Example 5-1 on page 60. The Order Header history file ORDHDRH is created via CRTDUPOBJ from the Order Header File (ORDHDR) described in Example 4-2 on page 36.

Example 7-19 on page 96 shows the DDS definition of the Order Detail file ORDDTL.

*Example 7-19 DDS definition for the Order Detail file ORDDTL*

---

```

A                               UNIQUE
A      R ORDDTLF
A      ORHNBR      5      COLHDG('ORDER NUMBER  ')
A      PRDNBR      5      COLHDG('PRODUCT NUMBER ')
A      ORDQTY      5P 0   COLHDG('ORDER DTL QUANTITY')
A      ORDTOT      9P 2   COLHDG('ORDERDTL TOTAL  ')
*
A      K ORHNBR
A      K PRDNBR

```

---

The Order Detail history file ORDDTLH is created via CRTDUPOBJ from the Order Detail File ORDDTL.

*Example 7-20 Write history files for Order Header and Order Detail with native I/O*

---

```

FOrdHdrL1 UF E      K DISK  Rename(OrdHdrF: OrdHdrF1)
FOrdDt1   UF E      K DISK
FOrdHdrH  0 E      DISK    Rename(OrdHdrF: OrdHdrHF)
FOrdDt1H  0 E      DISK    Rename(OrdDt1F : OrdDt1HF)
*-----
D PrevYear      S      4P 0      Previous Year

D KeyOrdHdrL1   DS      likeRec(OrdHdrF1: *Key)      Key for ORDHRL1
D KeyOrdDt1     DS      likeRec(OrdDt1F: *Key)      Key for ORDDTL
*-----
/Free
//Order Header
PrevYear = %SubDt(%Date(): *Years) - 1;
SetLL (%Date%Char(PrevYear) + '-01-01') OrdHdrF1;

DoU %EOF(OrdHdrL1);
Read OrdHdrF1;
If %EOF or %SubDt(OrdHdrF1: *Years) > PrevYear;
  leave;
EndIf;

// Order Detail
KeyOrdDt1.OrHNbr = OrHNbr;
clear KeyOrdDt1.PrdNbr;
SetLL %KDS(KeyOrdDt1) OrdDt1F;

DoU %EOF(OrdDt1);
ReadE %KDS(KeyOrdDt1: 1) OrdDt1HF;
If %EOF;
  leave;
EndIf;

Write OrdDt1HF; //Write Order Detail History Record
Delete OrdDt1F; //Delete Order Detail Record
EndDo;

Write OrdHdrHF; //Write Order Header History Record
Delete OrdHdrF1; //Delete Order Header Record

EndDo;
Return;
/End-Free

```

---

Example 7-21 shows how this can be done in four single SQL statements.

*Example 7-21 Write history files for Order Header and Order Detail with embedded SQL*

---

```

D PrevYear          S          4P 0          Previous Year
*-----
C          eval          PrevYear = %SubDt(%Date(): *Years) - 1

* Insert into Order Detail History File
C/EXEC SQL
C+ Insert into ORDDTLH
C+      Select d.*
C+      from ORDHDR h join ORDDTL d
C+      on      h.ORHNBR      = d.ORHNBR
C+      and year(OrHDte) = :PrevYear
C/END-EXEC

* Delete from Order Detail File
C/EXEC SQL
C+ Delete from ORDDTL d
C+      Where d.OrHNbr in (select  h.OrHNbr
C+                          from ORDHDR h
C+                          where year(h.OrHDte) = :PrevYear)
C/END-EXEC

* Insert into Order Header History File
C/EXEC SQL
C+ Insert into ORDHDRH
C+      Select *
C+      from ORDHDR
C+      where year(OrHDte) = :PrevYear
C/END-EXEC

* Delete from Order Header File
C/EXEC SQL
C+ Delete from ORDHDR h
C+      Where year(h.OrHDte) = :PrevYear
C/END-EXEC
C          Return

```

---

In static SQL you can embed almost all SQL statements that can be executed in interactive SQL or through iSeries Navigator.

The following example shows how a summary table over the Order Header, Order Detail, and stock tables is created containing the accumulated amounts by customer. Before creating the new summary table, an existing one will be deleted.

*Example 7-22 Creating a summary table with embedded SQL*

---

```

C/EXEC SQL Drop Table ITS04710/Summary
C/END-EXEC
C          If          SQLCODE = *Zeros or SQLCODE = -204

C/EXEC SQL
C+ create table ITS04710/Summary
C+      as (select year(current_date)-1 as fiscal_year, customer_number,
C+              sum(orderDt1_Quantity * Product_price) as amount
C+      from      Order_header h
C+      join      Order_detail d
C+      on        h.Order_Number = d.Order_Number

```

```

C+           and year(Order_Date) = year(current date) - 1
C+           join Stock s
C+           on      d.product_number = s.product_number
C+           group by customer_number)
C+   with data
C/END-EXEC

C           EndIf

C           Return

```

---

**Note:** Host variables can neither be used in the DROP TABLE nor in the CREATE TABLE statement. If a variable creation is needed, you have to use dynamic SQL.

## 7.8 Using a cursor

When SQL runs a select statement, the resulting rows comprise the result table. A cursor provides a way to access a result table. It is used within an SQL program to maintain a position in the result table. SQL uses a cursor to work with the rows in the result table and to make them available to your program. Your program can have several cursors, although each must have a unique name. Even if your source code consists of several independent procedures, the cursor name must be unique in your source member.

Using a cursor can be compared with native I/O, single record access.

Statements related to using a cursor include the following:

- ▶ A DECLARE CURSOR statement to define and name the cursor and specify the rows to be retrieved with the embedded select statement.

When moving from RPG native I/O to embedded SQL the declare cursor statement replaces the File specification.

**Note:** In your source, the DECLARE statement must always be positioned prior to the according OPEN, FETCH, and CLOSE statements. This is independent from the order in which these statements are executed. To put the DECLARE statement into the Initialization Subroutine (\*INZSR), that is coded at the end of the source, will cause a compile error.

Ascending or descending sequence in SQL is determined through a ORDER BY clause in the DECLARE statement, like it is fixed through the according logical file in the File specification.

**Note:** If a predefined sequence is not absolutely necessary, do not use a ORDER BY clause to let the optimizer evaluate all the options to find the optimal access path.

- ▶ OPEN statement to open the cursor for use within the program. The cursor must be opened before any rows can be retrieved.

The SQL OPEN statement can be compared with a user-controlled open of the table and an additional SETLL statement to position the pointer before the first row.

- ▶ A FETCH statement to retrieve rows from the cursor's result table or to position the cursor on another row.

The SQL FETCH statement is the equivalent to all CHAIN, READ, READE, READP, and READPE operations in RPG. By using a scrollable cursor, it is also possible to move the cursor forwards and backwards in your result table.

**Note:** Contrary to RPG, more than one row can be received in one FETCH statement, by using host structure arrays. The next FETCH will receive the next or previous block or rows.

- ▶ CLOSE statement to close the cursor for use within the program.

**Note:** When using a serial cursor, an OPEN without a preceding CLOSE will not reposition the cursor on the top of the result table. To be sure that the cursor is really closed, execute a CLOSE statement before your OPEN statement.

## 7.8.1 The DECLARE statement

The declare statement is used to define the cursor name and the associated SELECT statement.

An example of the DECLARE statement is shown below, which is important for embedded static SQL. All parts written in parenthesis can be omitted. For more information look at *iSeries DB2 Universal Database for iSeries SQL Reference*.

```
DECLARE Cursor Name (DYNAMIC (SCROLL)) CURSOR (WITH HOLD)
  FOR Select Statement
    (FOR READ ONLY/FOR FETCH ONLY)
    (FOR UPDATE (OF Column1, Column2,.....ColumnN))
    (OPTIMIZE FOR n ROWS)
    (WITH Isolation Level)
```

- ▶ Cursor name

Any name can be specified for the cursor, but a cursor name must be unique in the source member where it is defined.

**Note:** Even if the source member contains several independent procedures, the cursor name must be unique.

- ▶ NO SCROLL or (DYNAMIC) SCROLL

This specifies whether the cursor is scrollable or not scrollable. If neither NO SCROLL nor SCROLL is specified a serial cursor is defined.

- NO SCROLL

Specifies that a serial cursor is defined.

In Example 7-23 a serial cursor is declared by omitting the SCROLL or NO SCROLL keyword.

*Example 7-23 Declaring a serial cursor*

```
C/EXEC SQL
C+ Declare CsrOrdH Cursor for
C+ Select Customer_Number, sum(Order_Total)
C+   from Order_Header
C+   where   Order_Number between :FirstOrderNo and :LastOrderNo
C+          and Year(Order_Date) = :PrevYear
C+   group by Customer_Number
```

```
C+ order by sum(Order_Total) desc
C/End-Exec
```

---

#### – SCROLL

Specifies that the cursor is scrollable. The cursor may or may not have immediate sensitivity to inserts, updates, and deletes done by other activation groups.

If DYNAMIC is not specified, the cursor is read-only, which means that the SELECT statement cannot contain a FOR UPDATE clause.

**Note:** If DYNAMIC is not specified and an UPDATE is performed in your program, the SQL precompiler will accept it. But when running your program, you will get SQLCODE -510 (Cursor for table read only) and the rows will not be changed.

#### • DYNAMIC SCROLL

Specifies that the cursor is updatable if the result table is updatable, and that the cursor will usually have immediate sensitivity to inserts, updates, and deletes done by other application processes.

In Example 7-24 a scroll cursor is defined. All columns can be changed by a SQL UPDATE statement.

#### *Example 7-24 Declaring a scroll cursor*

---

```
C/EXEC SQL
C+ Declare CsrOrdH DYNAMIC SCROLL Cursor for
C+ Select Order_Total
C+   from Order_Header
C+   where   Order_Number between :FirstOrderNo and :LastOrderNo
C+         and Year(Order_Date) = :PrevYear
C/End-Exec
```

---

#### ► WITHOUT HOLD or WITH HOLD

This specifies whether the cursor should be prevented from being closed as a consequence of a commit or rollback operation. If neither WITHOUT HOLD nor WITH HOLD is specified, the cursor is closed when a commit or rollback operation is performed.

#### – WITHOUT HOLD

This does not prevent the cursor from being closed as a consequence of a commit or rollback operation. This is the default.

This must be considered when moving from native I/O to SQL data access. Let us assume that in a program two cursors are defined—the first to read all Order Header rows with a specified delivery date, and the second to update all Order Detail records assigned to the Order Header. You either have to update all Order Details for an order or none. You have to execute the commit after an order is completely updated. If an error occurs or not all detail rows are updated a rollback is performed. If the cursor is defined WITHOUT HOLD, you cannot continue with the next order, because the cursor will be closed through the executed COMMIT or ROLLBACK. To prevent this situation, you have to define the cursor to access the Order Header table WITH HOLD.

**Note:** The cursor is only closed when the SQL statements COMMIT or ROLLBACK are executed. The cursor remains open when the RPG Operation Codes COMMIT or ROLBK are used instead.

In the Example 7-23 on page 99 and Example 7-24 on page 100 cursors are declared without hold.

– WITH HOLD

This prevents the cursor from being closed as a consequence of a SQL commit or rollback operation.

When WITH HOLD is specified, a commit operation commits all the changes in the current unit of work, and releases all locks except those that are required to maintain the cursor position.

In Example 7-25 a serial cursor WITH HOLD is defined.

*Example 7-25 Declaring a serial cursor with WITH HOLD clause*

---

```
C/EXEC SQL
C+ Declare CsrOrdH Cursor WITH HOLD for
C+ Select Order_Number, Order_Total
C+   from Order_Header
C+   where   Order_Number between :FirstOrderNo and :LastOrderNo
C+         and Year(Order_Date)  = :PrevYear
C/End-Exec
```

---

► SELECT STATEMENT

Specifies the SELECT statement of the cursor that can contain references to host variables.

If dynamic SQL is used, the statement name that is defined by the PREPARE statement must be used.

– FOR READ ONLY clause

The FOR READ ONLY or FOR FETCH ONLY clause indicates that the result table is read-only and therefore the cursor cannot be used for positioned UPDATE and DELETE statements.

Some result tables are read-only by nature (for example, a table based on a read-only view or when tables are joined). FOR READ ONLY can still be specified for such tables, but the specification has no effect.

For result tables in which updates and deletes are allowed, specifying FOR READ ONLY can possibly improve the performance of FETCH operations by allowing the database manager to do blocking and avoid exclusive locks.

Example 7-26 shows a cursor that is read only.

*Example 7-26 Declaring a READ ONLY cursor*

---

```
C/EXEC SQL
C+ Declare CsrOrdH Cursor WITH HOLD for
C+ Select Order_Number, Order_Total
C+   from Order_Header
C+   where   Order_Number between :FirstOrderNo and :LastOrderNo
C+         and Year(Order_Date)  = :PrevYear
C+ For Read Only
C/End-Exec
```

---

– FOR UPDATE OF Clause

If the select statement is not read only, rows can be updated without specifying the FOR UPDATE clause.

The FOR UPDATE OF clause identifies the columns that can be updated in a subsequent positioned UPDATE statement. Each column name must be unqualified

and must identify a column of the table or view identified in the first FROM clause of the fullselect.

If the FOR UPDATE OF clause is specified and other columns than listed are updated, a negative SQLCODE, -503 (Column cannot be updated), is returned and no update is performed. The column names specified in the FOR UPDATE OF clause must not be selected in the SELECT statement.

The FOR UPDATE OF clause can be compared with the RPG built-in function %FIELDS that can be added to the UPDATE statement in RPG.

If the FOR UPDATE clause is specified without column names or not specified, all updatable columns of the table or view identified in the first FROM clause of the fullselect are included.

The FOR UPDATE clause must not be specified if the result table of the fullselect is read-only or the FOR READ ONLY clause is used.

In Example 7-27 a serial cursor with an additional FOR UPDATE OF clause is defined.

*Example 7-27 Declaring a serial cursor with specified FOR UPDATE OF clause*

```
C/EXEC SQL
C+ Declare CsrOrdH Cursor WITH HOLD for
C+ Select Order_Total
C+   from Order_Header
C+   where   Order_Number between :FirstOrderNo and :LastOrderNo
C+         and Year(Order_Date) = :PrevYear
C+ For Update of Order_Delivery, Order_Total
C/End-Exec
```

**Note:** In RPG you define in the File Specifications if a file is read as an update or input file. If a file is defined as an update file, the row is locked as soon as it is read.

If you do not use commitment control, a record can be unlocked by:

- ▶ Reading the next record
- ▶ Updating or deleting the record
- ▶ Using the operation code UNLOCK
- ▶ Specifying a (N)-extender in your read or chain statement

If you work with commitment control, a record is not unlocked by the operations code UNLOCK or the (N)-extender. The record will be unlocked as soon a COMMIT or ROLBK (Rollback) operation is executed.

In SQL a row is locked with the FETCH statement, if a UPDATE or DELETE statement with WHERE CURRENT OF CursorName is used in your source code. If no UPDATE or DELETE is executed, the row will not be locked with the FETCH statement, even if the cursor is updatable.

– FOR OPTIMIZE OF clause

The optimize clause tells the database manager to assume that the program does not intend to retrieve more than integer rows from the result table.

Without this clause, or with the keyword ALL, the database manager assumes that all rows of the result table are to be retrieved. Optimizing for integer rows can improve performance.

The database manager will optimize the query based on the specified number of rows. The clause does not change the result table or the order in which the rows are fetched.

Any number of rows can be fetched, but performance can possibly degrade after integer fetches.

The value of integer must be a positive integer (not zero).

In Example 7-28 a scroll cursor is defined with the OPTIMIZE FOR clause in the SELECT statement.

*Example 7-28 Declaring a scroll cursor with OPTIMIZE for clause*

---

```
C/EXEC SQL
C+ Declare CsrOrdH SCROLL Cursor for
C+ Select Order_Total
C+   from Order_Header
C+  where   Order_Number between :FirstOrderNo and :LastOrderNo
C+         and Year(Order_Date)  = :PrevYear
C+ Optimize for 100 rows
C/End-Exec
```

---

– WITH Isolation Level

The WITH Isolation Level clause allows us to override the isolation level that is determined at compile time for this statement.

The isolation level can be set to:

- RR - Repeatable Read
- RS - Read Stability
- CS - Cursor Stability

The KEEP LOCKS clause specifies that any read locks acquired will be held for a longer duration. Normally, read locks are released when the next row is read. If the isolation clause is associated with a cursor, the locks will be held until the cursor is closed or until a COMMIT or ROLLBACK statement is executed. Otherwise, the locks will be held until the completion of the SQL statement. The KEEP LOCKS clause is only allowed on an SQL SELECT, SELECT INTO, or DECLARE CURSOR statement. It is not allowed on updatable cursors.

- UR - Uncommitted Read
- NC - No Commit

In Example 7-29 a serial cursor with the for update clause and with isolation level RS = Read Stability is defined.

*Example 7-29 Declaring a serial cursor with specified isolation level*

---

```
C/EXEC SQL
C+ Declare CsrOrdH Cursor WITH HOLD for
C+ Select Order_Number, Order_Total
C+   from Order_Header
C+  where   Order_Number between :FirstOrderNo and :LastOrderNo
C+         and Year(Order_Date)  = :PrevYear
C+ For Update of Order_Delivery, Order_Total
C+ With RS
C/End-Exec
```

---

## 7.8.2 The OPEN statement

The open statement is used to open a cursor and position it before the first row. The cursor name defined in your declare statement must be declared in the open statement.

Example 7-30 on page 104 shows the DECLARE statement of a serial cursor and the appropriate OPEN statement.

*Example 7-30 Declaring and opening a serial cursor*

---

```
C/EXEC SQL
C+ Declare CsrOrdH Cursor WITH HOLD for
C+ Select Order_Number, Order_Total
C+   from Order_Header
C+   where   Order_Number between :FirstOrderNo and :LastOrderNo
C+          and Year(Order_Date) = :PrevYear
C+ For Update of Order_Delivery, Order_Total
C+ With RS
C/End-Exec

C/EXEC SQL Open CsrOrdH
C/END-EXEC
```

---

### 7.8.3 The FETCH statement

The FETCH statement positions a cursor on a row of the result table. It can return zero, one, or multiple rows, and it assigns the values of the rows returned to host variables.

We can differentiate between:

- ▶ A Single row FETCH
- ▶ A Multiple row FETCH

#### Single row FETCH

Below you will see the part of the FETCH statement that is important for retrieving a single row at a time from a serial cursor. The parameters in parenthesis are optional. For more information look at *iSeries DB2 Universal Database for iSeries SQL Reference*.

```
FETCH (Next FROM) Cursor Name
      INTO :Host Variable (:Indicator)
```

- ▶ NEXT FROM

If no scrolling option is specified, then the NEXT FROM keyword is optional.

When using a scroll cursor you can move forwards and backwards through the result table. Therefore you must specify one of the following options to position your cursor:

<b>NEXT</b>	Positions the cursor on the next row of the result table relative to the current cursor position
<b>PRIOR</b>	Positions the cursor on the previous row of the result table relative to the current cursor position
<b>FIRST</b>	Positions the cursor on the first row of the result table
<b>LAST</b>	Positions the cursor on the last row of the result table
<b>BEFORE</b>	Positions the cursor before the first row of the result table
<b>AFTER</b>	Positions the cursor after the last row of the result table
<b>CURRENT</b>	Does not reposition the cursor
<b>RELATIVE <i>integer</i></b>	Positions the cursor on a record relative to the current record depending on the value of the integer
<b>RELATIVE <i>:Host Variable</i></b>	Positions the cursor on a record relative to the current record depending on the value of the host variable

- ▶ Cursor name

The cursor name must identify a cursor defined in a DECLARE CURSOR statement. When the FETCH statement is executed, the cursor must be in the open state.

► INTO host variable

This identifies one or more host structures or host variables. In the operational form of INTO, a host structure is replaced by a reference to each of its variables. The first value in the result row is assigned to the first host variable in the list, the second value to the second host variable, and so on.

Example 7-31 shows the DECLARE statement and the appropriate OPEN and FETCH statements.

*Example 7-31 Using the FETCH statement to retrieve a single row*

---

```
D CustomerNo      S          5A
D TotalCustomer  S          11P 2
*-----*
C/EXEC SQL
C+ Declare CsrOrdH Cursor
C+ for Select  Customer_Number, sum(Order_Total)
C+      from  Order_Header
C+      where Year(Order_Date) = :PrevYear
C+      group by Customer_Number
C+      order by sum(Order_Total) desc
C/End-Exec

C/EXEC SQL Open  CsrOrdH
C/END-EXEC

C/EXEC SQL
C+ Fetch next from CsrOrdH
C+      into :CustomerNo, :TotalCustomer
C/END-EXEC
```

---

### Multiple row FETCH

In the text below we show an extract of the FETCH statement that is important for retrieving multiple rows at a time from a serial cursor. The parameters in parenthesis are optional.

```
FETCH (Next FROM) Cursor Name
      FOR integer ROWS
      INTO :Host array data structure (:Indicator Array data structure)
```

To fetch multiple rows at time it must be specified how many rows you want to receive.

► FOR *k* ROWS

This evaluates the host variable or integer to an integral value *k*. If a host variable is specified, it must be a numeric host variable with zero scale and it must not include an indicator variable. The value of *k* must be in the range of 1 to 32767. The cursor is positioned on the row specified by the orientation keyword (for example, NEXT), and that row is fetched. Then the next *k*-1 rows are fetched (moving forward in the table), until the end of the cursor is reached. After the fetch operation, the cursor is positioned on the last row fetched.

The maximum value of fetched rows (32 767) matches with the maximum elements an array data structure or a multi-occurrence data structure in RPG can have.

When a multiple-row-fetch is successfully executed, three statement information items are available in the SQL Diagnostics Area (or the SQLCA):

- ROW\_COUNT (or SQLERRD(3) of the SQLCA) shows the number of rows retrieved.

- DB2\_ROW\_LENGTH (or SQLERRD(4) of the SQLCA) contains the length of the row retrieved.
- DB2\_LAST\_ROW (or SQLERRD(5) of the SQLCA) contains +100 if the last row was fetched.

► INTO host structure array

The host structure array identifies an array data structure or a multi occurrence data structure. The first structure in the array corresponds to the first row, the second structure in the array corresponds to the second row, and so on. In addition, the first value in the row corresponds to the first item in the structure, the second value in the row corresponds to the second item in the structure, and so on. The number of rows to be fetched must be less than or equal to the dimension of the host structure array.

Example 7-32 shows the DECLARE statement, the OPEN statement, and a multiple row fetch into the external data structure DSOrderHeader.

*Example 7-32 Fetching multiple rows into a host structure array*

---

```

D DSOrderHeader E DS          ExtName(OrdHdr) Qualified Dim(3)
D Elements      S             3I 0 inz(%Elem(DSOrderHeader))
*-----
C/EXEC SQL
C+ Declare CsrOrdH Cursor for
C+ Select *
C+   from Order_Header
C+   where Order_Number between '00005' and '00050'
C/End-Exec

C/EXEC SQL Open CsrOrdH
C/END-EXEC

C/EXEC SQL
C+ Fetch next from CsrOrdH
C+   for :Elements rows
C+   into :DSOrderHeader
C/END-EXEC

```

---

### The CLOSE statement

The CLOSE statement closes a cursor explicitly. If a result table was created when the cursor was opened, that table is destroyed.

The cursor name that should be closed must be added to the CLOSE statement.

Example 7-33 shows the CLOSE statement for the cursor CsrOrdH (defined in Example 7-32).

*Example 7-33 Closing a cursor*

---

```

C/EXEC SQL Close CsrOrdH
C/END-EXEC

```

---

An implicit close of the cursor is performed by the end of the activation group or the module, depending on the option CLOSQLCSR that is set either in the compile command or the SQL SET OPTION statement.

**Note:** When using serial cursors, a cursor must be closed before it can be reopened. If the cursor is OPEN, a new OPEN will not be executed.

## 7.8.4 Types of cursors

The type of cursor determines the positioning methods that can be used with the cursor.

SQL distinguishes between two types of cursor:

- ▶ Serial cursor
- ▶ Scroll cursor

### Serial cursor

A serial cursor is one defined without the `SCROLL` keyword in the `DECLARE` statement.

For a serial cursor, each row of the result table can be fetched only once per `OPEN` of the cursor. When the cursor is opened, it is positioned before the first row in the result table. When a `FETCH` is issued, the cursor is moved to the next row in the result table. That row is then the current row. If host variables are specified (with the `INTO` clause on the `FETCH` statement), SQL moves the current row's contents into your program's host variables.

This sequence is repeated each time a `FETCH` statement is issued until the end-of-data (`SQLCODE = 100` or `SQLSTATE='02000'`) is reached. When you reach the end-of-data, close the cursor.

**Note:** You cannot access any rows in the result table after you reach the end-of-data. To use a serial cursor again, you must first close the cursor and then re-issue the `OPEN` statement. You can never back up using a serial cursor. To be sure that the cursor was really closed before you open it, execute the `CLOSE` statement before the `OPEN` statement.

A serial cursor can be compared with the RPG cycle definitions `IP` (Input Primary) or `UP` (Update Primary) in the File specifications.

### Scroll cursor

For a scrollable cursor, the rows of the result table can be fetched many times. The cursor is moved through the result table based on the position option specified on the `FETCH` statement. When the cursor is opened, it is positioned before the first row in the result table. When a `FETCH` is issued, the cursor is positioned to the row in the result table that is specified by the position option. That row is then the current row. If host variables are specified (with the `INTO` clause on the `FETCH` statement), SQL moves the current row's contents into your program's host variables. Host variables cannot be specified for the `BEFORE` and `AFTER` position options.

This sequence is repeated each time a `FETCH` statement is issued. The cursor does not need to be closed when an end-of-data or beginning-of-data condition occurs. The position options enable the program to continue fetching rows from the table. The following scroll options are used to position the cursor when issuing a `FETCH` statement. These positions are relative to the current cursor location in the result table.

<b>NEXT</b>	Positions the cursor on the next row of the result table relative to the current cursor position. <code>NEXT</code> is the default if no other cursor orientation is specified.
<b>PRIOR</b>	Positions the cursor on the previous row of the result table relative to the current cursor position.
<b>FIRST</b>	Positions the cursor on the first row of the result table.
<b>LAST</b>	Positions the cursor on the last row of the result table.

<b>BEFORE</b>	Positions the cursor before the first row of the result table.
<b>AFTER</b>	Positions the cursor after the last row of the result table.
<b>CURRENT</b>	Does not reposition the cursor, but maintains the current cursor position. If the cursor has been declared as DYNAMIC SCROLL and the current row has been updated so its place within the sort order of the result table is changed, an error is returned.
<b>RELATIVE</b>	Host variable or integer is assigned to an integer value k. RELATIVE positions the cursor to the row in the result table that is either k rows after the current row if k>0, or k rows before the current row if k<0. If a host variable is specified, it must be a numeric variable with zero scale and it must not include an indicator variable.

**Note:** A scroll cursor can only be moved by a number of rows, or be positioned at the end or beginning of the result table, but it is not possible to position by key. That means there is no equivalent for a SETGT / READ or SETLL / READ in SQL. To achieve this functionality you either have to DECLARE an additional cursor or you have to change the WHERE clause in your DECLARE statement, and CLOSE and OPEN the cursor again.

### 7.8.5 Updating or deleting rows using a cursor

If you want to update or delete a row that is fetched by using a cursor, you have to add WHERE CURRENT OF CursorName to your update or delete statement.

**Note:** If you do not add WHERE CURRENT OF CursorName, all rows are updated or deleted.

In Example 7-34 a serial cursor is defined to read all Order\_Header rows with a order date in the previous year. If the order date was in the months between January and March, the row is deleted; otherwise the order total is raised by 10.

*Example 7-34 Using the WHERE CURRENT OF clause to update and delete rows*

```

D PrevYear          S          4P 0
-----
D DSCsrOrdH         DS
D  OrderDate        D
D  OrderTotal       11P 2
*-----*
C          eval      PrevYear = %SubDt(%Date(): *Years) - 1

C/EXEC SQL
C+ Declare CsrOrdH Cursor WITH HOLD
C+ For Select  Order_Date, Order_Total
C+           from Order_Header
C+           where Year(Order_Date) = :PrevYear
C+ For Update of Order_Delivery, Order_Total
C/End-Exec

C/EXEC SQL Open  CsrOrdH
C/END-EXEC
C          DoU      SQLSTATE = '02000'
C/EXEC SQL
C+ Fetch next from CsrOrdH
C+           into :DSCsrOrdH
C/END-EXEC

```

```

C          if      SQLState = '02000' or SQLCode < *Zeros
C          leave
C          EndIf

C          if      %SubDt(OrderDate: *Months) <= 3
C/EXEC SQL
C+ delete from Order_Header
C+      where Current of CsrOrdH
C/END-EXEC
C          else
C          eval      OrderTotal += 10

C/EXEC SQL
C+ update Order_Header
C+      Set Order_Total = :OrderTotal,
C+      Order_Delivery = Current Date - 1 Year
C+ where Current of CsrOrdH
C/END-EXEC
C          EndIf

C          EndDo

C/EXEC SQL Close CsrOrdH
C/END-EXEC

C          Return

```

---

## 7.9 Dynamic SQL

Dynamic SQL allows you to define your SQL statements at runtime. That means you create a text string that contains the SQL statement. Before being executed the text string must be converted to an SQL statement.

Because the string is created at runtime, host variables are not necessary and cannot be used. They can be directly integrated into the string. But there are some situations where you wish to use variables. In these cases you can use parameter markers (?) that can be set in the EXECUTE or OPEN statement.

To convert the character string containing the SQL statement to an executable SQL statement one of the following steps is necessary:

- ▶ EXECUTE IMMEDIATE:
 

A string is converted to an SQL statement and executed immediately. This statement can only be used if no cursor is needed.
- ▶ PREPARE and EXECUTE:
 

A string is converted and later executed. Variables can be embedded as parameter markers and be replaced in the EXECUTE statement. EXECUTE can only be used if no cursor is needed.
- ▶ PREPARE and DECLARE CURSOR:
 

A string is converted and the converted SQL statement is used to DECLARE a cursor. Like in static SQL, either a serial or a scroll cursor can be used.

If you use a variable SELECT list a SQL Descriptor Area (SQLDA) is required where the returned variables are described.

## 7.9.1 Defining the character string containing the SQL statement

The character field to hold the SQL command can be defined as a Single Byte Character with fixed or varying length or as a Double Byte Character with fixed or varying length. The character string must be an executable SQL statement, which means that not only the correct syntax, but also the integrated variables must be converted.

- ▶ Character strings must be embedded in single quotation marks (").

To integrate a single quotation mark into a string, you have to double it. Do not use an Hex value instead, because this can cause problems when using different EBCDIC codes.

- ▶ Date and Time fields must be converted into a character string. The character representation requires a four-digit year format. Then the SQL scalar functions DATE or TIME must be used to convert these character strings into valid dates.

## 7.9.2 The EXECUTE IMMEDIATE statement

With EXECUTE IMMEDIATE, the command string is converted and executed in a single SQL statement. It is a combination of the PREPARE and EXECUTE statements. It can be used to prepare and execute SQL statements that contain neither host variables nor parameter markers.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is not valid, it is not executed and the error condition that prevents its execution is reported in the stand-alone SQLSTATE and SQLCODE. If the SQL statement is valid, but an error occurs during its execution, that error condition is reported in the stand-alone SQLSTATE and SQLCODE. Additional information on the error can be retrieved from the SQL Diagnostics Area (or the SQLCA).

In Example 7-35, the Order Header table with order date from the previous year and the appropriate Order Detail rows are saved in history tables. The names of the history tables are dynamically built. The year of the stored order data is part of the table name. In the create table statement the table is not only built but filled with the appropriate data.

Order Header and Order Detail rows are deleted by using static SQL.

*Example 7-35 Using the EXECUTE IMMEDIATE statement*

```
D PrevYear      S          4P 0
D MySQLString   S          32740A  varying
*-----*
/Free
PrevYear      = %SubDt(%Date(): *Years) - 1;
MyFile        = 'ORDDTL' + %Char(PrevYear);

MySQLString = 'Create Table ' + %Trim(MyLib) + '/' + %Trim(MyFile) +
              ' as (Select d.* +
                    from ORDHDR h join ORDDTL d +
                    on   h.OrHNbr   = d.OrHNbr +
                    and year(ORDHTE)= '+ %Char(PrevYear)+')+
              with Data';

/End-Free
C/EXEC SQL  Execute Immediate :MySQLString
C/End-Exec

C/EXEC SQL
C+ Delete from ORDDTL d
C+       Where d.OrHNbr in (select  h.OrHNbr
C+                               from ORDHDR h
```

```

C/END-EXEC
/Free

MyFile      = 'ORDHDR' + %Char(PrevYear);

MySQLString = 'Create Table ' + %Trim(MyLib) + '/' + %Trim(MyFile) +
              ' as (Select  *  from OrdHdr +
                  where year(ORHDTE) = '+ %Char(PrevYear)+' ) +
              with Data';

/End-Free
C/EXEC SQL  Execute Immediate :MySQLString
C/End-Exec

C/EXEC SQL
C+ Delete from  ORDHDR h
C+           Where year(h.OrHDte) = :PrevYear
C/END-EXEC

C              Return

```

---

### 7.9.3 Combining the SQL statements PREPARE and EXECUTE

If a single SQL statement must be executed repeatedly, it will be better to prepare the SQL statement once, using the SQL PREPARE statement, and execute the statement several times with the SQL EXECUTE statement.

When using the EXECUTE IMMEDIATE statement instead, the PREPARE statement is performed every time.

#### **Parameter markers**

Although a statement string cannot include references to host variables, it may include parameter markers. These can be replaced by the values of host variables when the prepared statement is executed. A parameter marker is a question mark (?) that is used where a host variable could be used if the statement string were a static SQL statement.

#### **The PREPARE statement**

The PREPARE statement creates an executable form of an SQL statement from a character-string form of the statement. The character-string form is called a statement string, and the executable form is called a prepared statement.

The text below illustrates the parts of the PREPARE statement that are necessary to convert a character string into a SQL string. For more information look at *iSeries DB2 Universal Database for iSeries SQL Reference*.

```

PREPARE statement-name
FROM host variable

```

##### ► Statement name

This specifies the Name of the SQL statement. The statement name must be unique in your source member.

##### ► Host variable

This specifies the character variable that contains the SQL string.

#### **The EXECUTE statement**

The EXECUTE statement executes a prepared SQL statement without a cursor.

The text below shows the EXECUTE statement. For more information look at *iSeries DB2 Universal Database for iSeries SQL Reference*.

```
EXECUTE statement-name
    (USING HostVariable1, HostVariable2,.....HostVariableN)
```

► **Statement name**

This specifies the name of the SQL statement that is to be executed.

► **USING :HostVariable**

If parameters are used, USING and the host variables that contain the values must be listed. The host variables must be listed in the sequence they are needed.

Example 7-36 shows an example where an in INSERT statement is dynamically built once and then executed several times.

*Example 7-36 Dynamic SQL without cursor by using PREPARE and EXECUTE statements*

```
D KeyOrHDte      S          like(OrHDte)
D NextMonth     S          like(OrHDte)

D DsFile        DS
D MyFile        10A
D               3A overlay(MyFile) inz('ORH')
D FileYear      4S 0 overlay(MyFile: *Next)
D FileMonth     2S 0 overlay(MyFile: *Next)

D MyLib         S          10A inz('ITS04710')
D MySQLString   S          256A
*-----
/Free
FileYear = %SubDt(%Date(): *Years) - 1;
FileMonth = %SubDt(%Date(): *Months) - 3;
MySQLString = 'Insert into ' + %Trim(MyLib) + '/' + %Trim(MyFile) +
              ' Values(?, ?, ?, ?, ?, ?)';
/End-Free

C/EXEC SQL Prepare MyDynSQL from :MySQLString
C/End-Exec

/Free
KeyOrHDte = %Date(%Char(FileYear) + '-'
              + %EditC(FileMonth: 'X') + '-01');
NextMonth = KeyOrHDte + %Months(1);

SetLL KeyOrHDte ORDHDRF1;

DoU %EOF(ORDHDRL1);
  Read OrdHdrF1;
  If %EOF or OrhDte >= NextMonth;
    leave;
  endif;
/End-Free
C/EXEC SQL
C+ Execute MyDynSQL
c+ using :OrHNbr, :CusNbr, :OrHDte, :OrHD1y, :SrNbr, :OrHTot
C/End-Exec
/Free
  EndDo;

Return;
```

## 7.9.4 Combining the SQL statements PREPARE and DECLARE

If a cursor must be defined, but the SQL select statement cannot be defined at compile time, the character string can be built at runtime. The PREPARE statement converts the character string to an SQL string.

The DECLARE statement defines the SQL cursor using the executable SQL string instead of an SELECT statement.

### The PREPARE statement

The PREPARE statement creates an executable form of an SQL statement from a character-string form of the statement. The character-string form is called a statement string, and the executable form is called a prepared statement.

The text below shows the parts of the PREPARE statement that are necessary to convert a character string into a SQL string. For more information look at *iSeries DB2 Universal Database for iSeries SQL Reference*.

```
PREPARE statement-name  
FROM host variable
```

► Statement name

This specifies the name of the SQL statement. The statement name must be unique in your source member.

► Host variable

This specifies the character variable that contains the SQL string. The character string can contain the FOR READ/FETCH ONLY clause, the OPTIMIZE clause, the UPDATE clause, and the WITH Isolation Level clause.

### The DECLARE CURSOR statement

The DECLARE statement defines the cursor for the executable SQL statement. Like in static SQL, a serial and a scroll cursor can be created.

The text below shows the DECLARE CURSOR statement, and how it can be used with dynamic SQL.

```
DECLARE Cursor Name (DYNAMIC (SCROLL)) CURSOR (WITH HOLD)  
FOR Prepared SQL Statement
```

In Example 7-37 a SELECT statement that includes a date conversion is dynamically built. The SQL PREPARE and DECLARE statements are executed.

*Example 7-37 Using a cursor with dynamic SQL*

---

```
D StartDate      S          D  
D LastDate       S          D  
  
D Customer       S          5A  
D OrderTotal     S          11P 2  
D MyString       S          256A  varying  
  
D DspText        S          50A  
*-----  
/Free  
  FirstDate = D'2004-06-01';
```

```

LastDate = D'2004-06-30';

MyString = 'Select Customer_Number, Sum(Order_Total) +
           from Order_Header +
           where Order_Date between Date(''+%Char(FirstDate)+'') +
           and Date(''+%Char(LastDate) +''') +
           group by Customer_Number +
           order by Sum(Order_Total) desc';

/End-Free
C/EXEC SQL Prepare MySQLstm from :MyString
C/End-Exec

C/EXEC SQL
C+ Declare CsrOrdH Cursor WITH HOLD for MySQLstm
C/End-Exec

C/EXEC SQL Open CsrOrdH
C/END-EXEC
C           DoU           SQLSTATE = '02000'

C/EXEC SQL
C+ Fetch next from CsrOrdH
C+       into :Customer, :OrderTotal
C/END-EXEC

C           if           SQLState = '02000' or SQLCode < *Zeros
C           leave
C           EndIf

C           eval       DspText = Customer + ' ' + %Char(OrderTotal)
C   DspText   Dsply

C           EndDo

C           Return

```

---

## 7.9.5 The SQL descriptor area

A SELECT statement with a variable SELECT list (that is, a list of columns to be returned as part of the query) that runs dynamically requires an SQL descriptor area (SQLDA). This is because you cannot know in advance how many or what type of variables to allocate in order to receive the results of the SELECT.

An INCLUDE SQLDA statement can be specified at the end of the Definition specifications in an ILE RPG for iSeries program.

Example 7-38 shows how the INCLUDE statement can be integrated. SQL\_NUM is used in the SQLDA. It must be defined as a numeric constant and include the maximum number of selected variables.

*Example 7-38 Embedding descriptor area in RPG*

```

D SQL_NUM      C           const(5)
C/EXEC SQL Include SQLDA
C/END-EXEC

```

---

Example 7-39 on page 115 shows the SQLDA how it will be embedded by the precompiler in the RPG source member.

Example 7-39 SQL descriptor area

---

```
D*      SQL Descriptor area
D SQLDA      DS
D  SQLDAID      1      8A
D  SQLDABC      9     12B 0
D  SQLN       13     14B 0
D  SQLD       15     16B 0
D  SQL_VAR    17     80A  DIM(SQL_NUM)
D
D           19     20B 0
D           21     32A
D           33     48*
D           49     64*
D           65     66B 0
D           67     96A
D*
D SQLVAR      DS
D  SQLTYPE     1      2B 0
D  SQLLEN      3      4B 0
D  SQLRES      5      16A
D  SQLDATA    17     32*
D  SQLIND     33     48*
D  SQLNAMELEN 49     50B 0
D  SQLNAME    51     80A
D* End of SQLDA
D* Extended SQLDA
D SQLVAR2     DS
D  SQLLONGL   1      4B 0
D  SQLRSVDL   5     32A
D  SQLDATAL  33     48*
D  SQLTNAMEN 49     50B 0
D  SQLTNAME   51     80A
D* End of Extended SQLDA
```

---

Archived

## Externalizing data access

In our journey of database modernization we have:

- ▶ Reversed engineered the database definition and now the database has been defined by SQL
- ▶ Created I/O modules to access DB2 data
- ▶ Normalized and moved the business rules to the database

In the final stage of the proposed methodology we are proposing the use of stored procedures, triggers, and user defined functions to complement and enhance the database access.

In this chapter we discuss how triggers, stored procedures, and user defined functions can assist you in moving more of the business logic into the database.

## 8.1 Trigger programs

Triggers are user-written programs that are directly linked with a table or physical file. They are automatically activated by the database manager when a data change is performed, regardless of which interface caused that data change. It does not matter if a row is changed by native I/O, by interactive SQL, by UPDDTA, by SQLJ, etc., the trigger is activated. The triggers are mainly intended for monitoring database changes and taking appropriate actions, and for enforcing complex business rules.

### The purposes of using triggers

The purposes are:

- ▶ Enforcing business rules, regardless of how complex they are

After an order is completely delivered, you may want to copy the Order Header and the appropriate Order Detail rows into a history table and delete the original rows.

Another example may be if you enter a order, you want to check if the desired quantity of the product number is on stock. If so, your program will automatically reserve the requested quantity. Hence, if you complete your order and all quantities are available, the order will be automatically shipped. If new products are restocked, it can be checked if there are open orders and the missing quantities can be automatically reserved and even eventually shipped, if the order is completed.
- ▶ Data validation and audit trail

You may need to ensure that, whenever a sales representative enters an order, a sales representative is actually assigned to that particular customer. You want to also keep track of the violation attempts. Again, a trigger can be activated on the order table to perform the validation and track the violators in a separate table.
- ▶ Integrating existing applications and advanced technologies

If you get an order entry from a client and you are out of stock for some of the ordered products, a trigger can automatically create and send a fax or e-mail to the client, to announce that the products can actually not be delivered. It could also send a purchase request to the supplier.
- ▶ Preserving data consistency across different database tables

In this case, triggers can complement referential integrity and check constraint support, since they can provide a much wider and more powerful range of data validation and business actions to be performed when data changes in your database.

With check constraints you can only compare columns from the same table. With triggers you can compare different tables. For example, if you enter an order, you can check whether the product is out of stock. All stocks are restored in a separate table.

### The benefits of using triggers

The benefits are:

- ▶ Application independence

DB2 Universal Database for iSeries activates the trigger program, regardless of the interface you are using to access the data. Rules implemented by triggers are enforced consistently by the system rather than by a single application.

Because the triggers are integrated into the database, no additional calls in programs or procedures are required.
- ▶ Easy maintenance

If you must change the business rules in your database environment, you only need to update or rewrite the triggers. No change is needed to the applications (they transparently comply with the new rules).

- ▶ Code reusability

Functions implemented at the database level are automatically available to all applications using that database. You do not need to replicate those functions throughout the different applications.

- ▶ Easier client/server application development

Client/server applications take advantage of triggers. In a client/server environment, triggers may provide a way to split the application logic between the client and the server system. In addition, client applications do not need specific code to activate the logic at the server side. Application performance may also benefit from this implementation by reducing data traffic across communication lines.

A maximum of 300 triggers can be added to one single table. The trigger program to be called can be the same for each trigger or it can be a different program for each trigger. If there is more than one trigger for a single event, activation time, or column defined, the triggers are executed in the sequence they are created, which means that the last created trigger is executed last. This must be considered if conflicting triggers are defined.

The activation of trigger programs is determined by the associated:

- ▶ Trigger time
- ▶ Trigger event

There are two types of triggers:

- ▶ External triggers
- ▶ SQL triggers

Trigger programs can activate additional trigger programs by executing database changes through integrated inserts, updates, and deletes to other files (trigger cascading).

If a trigger event occurs, the database manager calls either QDBPUT (for input triggers) or QDBUDR (for update or delete triggers). These programs start the proper trigger programs. The QDBPUT and QDBUDR programs and the triggers are integrated in the call stack. After having executed the trigger programs the control goes back to the running application program. Because the triggers are embedded in the call stack, they can run under the same commitment level, assuming that the activation group of the trigger programs is \*CALLER or the commitment control (STRCMTCTL) is started commitment definition scope (CMTSCOPE) \*JOB.

Trigger programs cannot return any parameter values to their caller. If a failure occurs, an escape message must be sent to the caller. In SQL triggers a SIGNAL statement will do the job. By sending an escape message, all call stack entries between the program where the message is sent to and the sender are ended and removed from the call stack.

### 8.1.1 Activation time of trigger programs

Triggers are called by the database manager, when a data change is performed depending on the trigger time and the trigger event, that are defined with the trigger.

We can differentiate between two trigger times:

- ▶ \*BEFORE

A BEFORE trigger is activated before the row is physically written to the disc. BEFORE triggers are not used for further modifying the database because they are activated before the trigger event is applied to the database. Consequently, they are activated before integrity constraints are checked and may be violated by the trigger event.

BEFORE triggers can be used to change column values before an update or delete. Let us assume that you have to save the timestamp of the creation or last update in your table.

Example 8-1 shows a before insert trigger that inserts the current timestamp and user in a row.

*Example 8-1 Before insert trigger to update insert time and user*

---

```
CREATE TRIGGER ITS04710/MYTRGTABLE01
  BEFORE INSERT ON MYTRGTABLE
  REFERENCING NEW Ins
  FOR EACH ROW
  MODE DB2ROW
  BEGIN ATOMIC
    set Ins.InsertTime = Current_Timestamp;
    set Ins.InsertUser = User;
  END;
```

---

Another example would be to generate a serial number in a table, depending on a key.

The following example shows how a serial number depending on a key value is generated.

*Example 8-2 Before insert trigger to build a serial number depending in a key*

---

```
Create Trigger ITS04710/MyTrgTable02
  BEFORE INSERT on ITS04710/MyTrgTable
  Referencing NEW as INS
  For Each Row
  Mode DB2ROW
  Select Coalesce(Max(a.CurrNbr) + 1 , 1)
    into INS.CurrNbr
  from ITS04710/MyTrgTable a
  Where a.Text = INS.Text;
```

---

► **\*AFTER**

The trigger program is called after the change operation on the specified table.

As a part of an application, AFTER triggers always see the database in a consistent state. They can be used to execute additional actions in the database, like updating or deleting other tables, for example, writing transaction data, history tables, or summary tables. Further, they can be used to perform actions outside the database, for example, printing invoices or sending e-mails.

**Note:** AFTER triggers are run after the integrity constraints that may be violated by the triggering SQL operation have been checked.

BEFORE triggers are activated before integrity constraints are checked and may be violated by the trigger event.

Example 8-3 on page 121 shows an AFTER UPDATE trigger that automatically updates the reservations if the order quantity changes.

### Example 8-3 After update trigger

---

```
Create Trigger ITS04710/UpdateReservation01
  AFTER UPDATE of OrderDt1_Quantity on ITS04710/ORDER_
  Referencing OLD as O
              NEW as N
  For Each Row
  Mode DB2ROW
  When (O.OrderDt1_Quantity <> N.OrderDt1_Quantity)
  BEGIN ATOMIC
  Update ITS04710/Reservation R
    set Reserved_Quantity =    R.Reserved_Quantity
                              - O.OrderDt1_Quantity
                              + N.OrderDt1_Quantity
  Where    R.Order_Number = N.Order_Number
          and R.Product_Number = N.Product_Number;
END;
```

---

## 8.1.2 Trigger events

Every trigger is associated with an event. Triggers are activated when their corresponding event occurs in the database. The trigger event occurs when the specified action, either an UPDATE, INSERT, or DELETE (including those caused by actions of referential constraints), is performed on the subject table.

The database events do not include clearing, initializing, moving, applying journal changes, removing journal changes, or changing end-of-data operations.

We can differentiate between four events:

► **Insert**

An insert trigger is activated as soon a row is added into the table, for example, by using the SQL statement INSERT, or a write in native I/O or the CL command CPYF (Copy File), etc.

► **Update**

An update trigger is activated if a database row is changed by using the SQL statement UPDATE, or an update in native I/O, etc. An update trigger can also be fired as the result of a referential constraint clause ON DELETE SET NULL or ON DELETE SET DEFAULT.

While external triggers can only be defined for the whole row, an SQL trigger can be defined on either row or column level.

► **Delete**

A delete trigger is activated if a database row is deleted by using the SQL statement DELETE or a delete in native I/O, etc. A delete trigger can also be fired as result of a referential constraint clause ON DELETE CASCADE.

For example, you defined in your constraint that all Order Detail rows must be deleted if the associated header is deleted. Additionally, you put an AFTER DELETE trigger on the Order Detail table. When deleting the Order Header, this trigger will be fired.

► **Read**

A read trigger can only be defined as an external trigger. An SQL trigger cannot be activated through a READ or FETCH statement.

### 8.1.3 External triggers

External triggers are programs written in any HLL language and can be used for DDS-described physical files or SQL-defined tables. Trigger programs can contain embedded SQL statements.

External triggers are activated on row level, which means the program is called as soon as one column value in the row was changed. It is not possible to define external triggers at the column level.

Because the trigger programs are called by the database manager, independent of which interface was used, there are some recommendations for external trigger programs you have to care about.

For trigger programs two parameters are required:

- ▶ Trigger buffer
- ▶ Length of the trigger buffer

External triggers are not automatically linked to the database table, but there are two methods for registering the triggers:

- ▶ CL command `ADDPFTRG` (Add Physical File Trigger)
- ▶ Using the iSeries Navigator

#### Recommendations for external trigger programs

The recommendations are:

- ▶ Create the trigger program so that it runs under the user profile of the user who created it. In this way, users who do not have the same level of authority to the program will not encounter errors.
- ▶ Create the program with `USRPRF(*OWNER)` and `*EXCLUDE` public authority, and do not grant authorities to the trigger program to `USER(*PUBLIC)`. Avoid having the trigger program altered or replaced by other users. The database invokes the trigger program whether or not the user causing the trigger program to run has authority to the trigger program.
- ▶ Create the program as `ACTGRP(*CALLER)` if the program is running in an ILE environment. This allows the trigger program to run under the same commitment definition as the application.

**Note:** The default value for the commitment definition scope (CMTSCOPE) in the `STRCMTCTL` command is `*ACTGRP`.

If you started the commitment control with the commitment definition scope `*JOB`, you can use any activation group.

- ▶ Open the table with a commit lock level the same as the application's commit lock level. This allows the trigger program to run under the same commit lock level as the application.
- ▶ Create the program in the table's schema. When saving and restoring your schema, the triggers are correctly activated. When saving in different schemas, the triggers must be recreated or added to the table.
- ▶ Use `commit` or `rollback` in the trigger program if the trigger program runs under a different activation group than the application. If the trigger runs in the same activation group, avoid `commits` and `rollbacks` and let it perform outside the trigger program.

- ▶ Signal an exception or send an escape message if an error occurs or is detected in the trigger program. If an error message is not signalled from the trigger program, the database assumes that the trigger ran successfully. This may cause the user data to end up in an inconsistent state.

**Required parameters for external triggers**

When a trigger is activated, the system automatically provides the program with the following parameter list:

- ▶ Trigger buffer:

The trigger buffer has two logical parts:

- Static area:

- A trigger template that contains the physical file name, member name, trigger event, trigger time, commit lock level, and CCSID of the current change record and relative record number.
- Offsets and lengths of the record areas and null byte maps. This area occupies (in decimal) offset 0 through 95.

- Dynamic area:

Areas for the old record and old null byte map, new record, and new null byte map

- ▶ Trigger buffer length:

The length of the trigger buffer provided by DB2 UDB for iSeries. The trigger buffer length is a 4-byte binary value. In RPG you have to define this parameter as 9B 0 or even better as 10I 0.

By defining these parameters in your trigger programs, you can take the appropriate actions based on the kind of data change that has occurred and the characteristics of the job that fired the trigger.

**Trigger buffer**

Table 8-1 shows the static part of the trigger buffer.

Table 8-1 The trigger buffer structure

Decimal offset	Parameter	Type	Description
0	Physical file Name	char(10)	The physical file being changed.
10	Physical file library name	char(10)	The library in which the physical file resides.
20	Physical file member name	char (10)	The name of the physical file member.
30	Trigger Event	char(1)	The event that caused the trigger program to be called; the possible values can be "1" (Insert), "2" (Delete), "3" (Update), "4" (Read).
31	Trigger Time	char(1)	Can be "1" (After) or "2" (Before).
32	Commit level	char(1)	Reports the commit lock level of the interface that activated the trigger "0" (*NONE), "1" (*CHG), "2" (*CS), "3" (*ALL).
33	Reserved	char(3)	Reserved.
36	CCSID of data	binary(4)	The CCSID of the data in the new or the original records; the data is converted to the job CCSID by the database.

Decimal offset	Parameter	Type	Description
40	Relative record number	binary(4)	Relative record number of the record to be updated or deleted (*BEFORE triggers) or the relative record number of the record that was inserted, updated, deleted, or read (*AFTER triggers).
44	Reserved	char(4)	Reserved.
48	Original Record offset	binary(4)	The location of the original record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the original value of the record does not apply to the operation; for example, an insert operation.
52	Old record length	binary(4)	The maximum length is 32766 bytes.
56	Old record null map offset	binary(4)	The location of the null byte map of the original record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the original value of the record does not apply to the change operation, for example, an insert operation.
60	Old record null map length	binary(4)	The length is equal to the number of fields in the physical file.
64	New record offset	binary(4)	The location of the new record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the new value of the record does not apply to the change operation, for example, a delete operation.
68	New record length	binary(4)	The maximum length is 32766 bytes.
72	New record null map offset	binary(4)	The location of the null byte map of the new record. The offset value is from the beginning of the trigger buffer. This field is not applicable if the new value of the record does not apply to the change operation, for example, a delete operation.
76	New record null map length	binary(4)	The length is equal to the number of fields in the physical file.
80	Reserved	char(16)	Reserved.
*	Original record	char(*)	A copy of the original physical record before being updated, deleted, or read. The original record applies only to update, delete, and read operations.
*	Original record null byte map	char(*)	This structure contains the NULL value information for each field of the original record. Each byte represents one field. The possible values for each byte are: "0" (Not NULL) and "1" (NULL).
*	New record	char(*)	A copy of the record that is being inserted or updated in a physical file as a result of the change operation. The new record only applies to the insert or update operations.
*	New record null byte map	char(*)	This structure contains the NULL value information for each field of the new record. Each byte represents one field. The possible values for each byte are: "0" (Not NULL) and "1" (NULL).

If you like to use your own field names in your program, you have to define a data structure containing position 1-80.

**Note:** Binary(4) means the maximum value can be saved in 4 byte. In RPG Binary(4) must be defined as 9B 0 or better 10I 0. The integer definition can hold the complete range of binary values.

A template of the trigger buffer is saved in the library QSYSINC, file QRPGLSRC for ILE RPG programs, QRPGRSRC for RPG/400 programs, and QCBLLSRC for cobol programs, with the member name TRGBUF.

Example 8-4 is a copy of the trigger buffer saved in the QSYSINC library for ILE RPG programs.

*Example 8-4 Trigger buffer saved in QSYSINC*

---

DQDBTB	DS			
D*				Qdb Trigger Buffer
D QDBFILN02		1	10	
D*				File Name
D QDBLIBN02		11	20	
D*				Library Name
D QDBMNO0		21	30	
D*				Member Name
D QDBTE		31	31	
D*				Trigger Event
D QDBTT		32	32	
D*				Trigger Time
D QDBCLL		33	33	
D*				Commit Lock Level
D QDBRSV104		34	36	
D*				Reserved 1
D QDBDAC		37	40B 0	
D*				Data Area Ccsid
D QDBCR		41	44B 0	
D*				Current Rrn
D QDBRSV204		45	48	
D*				Reserved 2
D QDBORO		49	52B 0	
D*				Old Record Offset
D QDBORL		53	56B 0	
D*				Old Record Len
D QDBORNBM		57	60B 0	
D*				Old Record Null Byte Map
D QDBRNBM		61	64B 0	
D*				Old Record Null Byte Map Len
D QDBNRO		65	68B 0	
D*				New Record Offset
D QDBNRL		69	72B 0	
D*				New Record Len
D QDBNRNBM		73	76B 0	
D*				New Record Null Byte Map
D QDBRNML00		77	80B 0	
D*				New Record Null Byte Map Len

---

This source snippet can be easily embedded as copy member in your source code.

The following discussion refers to the most important fields in the trigger buffer, as previously marked:

► **Trigger Event (QDBTE):**

This field gives you the possibility of determining the event that called the trigger. This information is particularly valuable when a trigger is defined for different events. You may want to identify which record image to use, depending on the event that has activated the trigger.

- If the trigger event is INSERT, only the new record is available.

- If the trigger event is UPDATE, the new and the old records are available.
- If the trigger event is DELETE, only the old record is available.

► Commitment Level (QDBCLL):

We do not know which interface fired the trigger. Consequently, different commitment levels can be used. Using the SQL SET TRANSACTION statement, you can change the commitment level for your program.

If you use native I/O in your trigger program, you can define your files as user controlled open and add the keyword COMMIT(Indicator). Before opening your file you can set the indicator depending on the commitment level.

Example 8-5 shows how files can be used with conditional commit in RPG.

*Example 8-5 Setting commitment level in external triggers*

---

```

FOrdHdr  UF  E           K DISK  UsrOpn Commit(CmtLv1)
*-----*
* Data structure QDBTB = Static part of the trigger buffer
/COPY QSYSINC/QRPGLESRC,TrgBuf

D CmtLv1      S           N
D NoCommit    C           Const('0')
*-----*
/Free
  If QDBCLL = NoCommit;
    CmtLv1 = *Off;
  Else;
    CmtLv1 = *On;
  EndIf;

  Open  OrdHdr;

  *InLR = *On;
/End-Free

```

---

► Old record offset QDBORO:

In the variable part of the trigger buffer the record before and/or after the trigger event are restored. Because different files have different record lengths, but always the same parameter interface is used, the start and end point for the records are different.

Old record offset specifies the number of bytes from the begin of the trigger buffer to the start byte of the old record.

► Old record length QDBORL:

Specifies the number of bytes that are used for the old record.

► New record offset QDBNRO:

New record offset specifies the number of bytes from the begin of the trigger buffer to the start byte of the new record.

► New record length QDBNRL:

Specifies the number of bytes that are used for the new record.

There are two common techniques to use the trigger buffer offset information to correctly locate the record images.

► Using a substring function

Example 8-6 on page 127 shows an AFTER INSERT trigger.

After inserting a new Order Detail record, a new reservation for the product or an update of an existing reservation is performed. The new inserted record is moved to an external data structure, by using the built-in function %SubSt (Substring)

*Example 8-6 Retrieving old/new record in an external trigger by using built-in function %Subst()*

```

H DEBUG Option(*NoDebugIo)
*-----
* Data structure QDBTB - Static Part of the TriggerBuffer
/COPY QSYSINC/QRPGLESRC,TrgBuf
D          81  9999

D TrgBufLen      S          10I 0

D New          e ds          extname(OrderDt1) qualified

D CmtLvl        S          N
D NoCommit      C          Const('0')
*-----
C  *entry      plist
C              parm          QDBTB
C              parm          TrgBufLen
/Free
//Open file with the appropriate commitment level
If  %Open(Reserve)
    and ( QDBCLL = NoCommit and CmtLvl = *On
        or QDBCLL <> NoCommit and CmtLvl = *Off);
    Close Reserve;
EndIf;

If Not %Open(Reserve);
    Open Reserve;
EndIf;

//Move new Record into external datastructure
New = %Subst(QDBTB: QDBNRO + 1: QDBNRL);

//Create new or update existing reservation
Chain (New.OrHNbr: New.PrdNbr) ReserveF;
If Not %Found;
    clear ReserveF;
EndIf;

OrHNbr = New.OrhNbr;
PrdNbr = New.PrdNbr;
ResQty += New.OrdQty;

If %Found(Reserve);
    Update ReserveF;
else;
    write ReserveF;
EndIf;

Return;
/End-Free

```

► Using pointers in RPG

The following example shows the same program as in Example 8-7 on page 128, with two minor differences. The external data structure is based on a pointer, and instead of using a

substring function to move the new record into the external data structure, a pointer is used.

*Example 8-7 Retrieving old/new record in a system trigger by using based pointers*

---

```

H DEBUG Option(*NoDebugIo)
*-----
* Data structure QDBTB - Static Part of the TriggerBuffer
/COPY QSYSINC/QRPGLESRC,TrgBuf
D          81  9999

D TrgBufLen      S          10I 0

D CmtLvl        S          N
D NoCommit      C          Const('0')

* Externe Datenstrukturen
D New          e ds          extname(OrderDt1) qualified
D          based(PtrNewRec)
*-----
C  *entry      plist
C              parm          QDBTB
C              parm          TrgBufLen
/Free
  If  %Open(Reserve)
    and ( QDBCLL = NoCommit and CmtLvl = *On
        or QDBCLL <> NoCommit and CmtLvl = *Off);
    Close Reserve;
  EndIf;

  If Not %Open(Reserve);
    Open Reserve;
  EndIf;

PtrNewRec = %addr(QDBTB) + QDBNRO;

Chain (New.OrHNbr: New.PrdNbr) ReserveF;
If Not %Found;
  clear ReserveF;
EndIf;

OrHNbr = New.OrhNbr;
PrdNbr = New.PrdNbr;
ResQty += New.OrdQty;

If %Found(Reserve);
  Update ReserveF;
else;
  write ReserveF;
EndIf;

Return;
/End-Free

```

---

**Note:** Please do not hard code the start of your old and new record image in your data structure. The dynamic part of the trigger buffer could be changed.

There is a very interesting technique called *Softcoding the trigger buffer* that is proposed and described in Paul Conte's book *Database Design and Programming for DB2/400*. The purpose of this technique is that if the trigger buffer is softcoded, any changes to the underlying structure can be incorporated by simply recompiling the trigger program.

## Registering an external trigger

External triggers are not automatically linked with a table, but there are two methods to register the trigger.

### **CL command Add Physical File Trigger (ADDPFTRG)**

The Add Physical File Trigger (ADDPFTRG) command associates a trigger program with a physical file. Once this association is established, DB2 UDB for iSeries calls the trigger program when a change operation is performed against the physical file, a member of the physical file, and any logical file created over the physical file or views created by SQL.

To register an external trigger with ADDPFTRG, you have to specify the following options:

- ▶ Trigger event
- ▶ Trigger time
- ▶ Trigger program and library (\*LIBL is allowed, but the actual library name is resolved and stored in the file description)

All other options are optional or the default values can be used. For more information look at the existing redbook *Stored Procedures, Triggers and User Defined Functions in DB2 UDB for iSeries* at:

<http://www.redbooks.ibm.com/redbooks/pdfs/sg246503.pdf>

Example 8-8 shows registration of the trigger program in Example 8-7 on page 128.

#### *Example 8-8 Registering a system triggers using the CL command ADDPFTRG*

---

```
ADDPFTRG FILE(ITS04710/OrderDt1)
          TRGTIME(*AFTER)
          TRGEVENT(*INSERT)
          PGM(ITS04710/HSTRG04)
          TRG(InsertReservation02)
```

---

### **Registering external triggers with iSeries Navigator**

To register external triggers with iSeries Navigator, the following steps are necessary:

1. In the iSeries Navigator window, expand your **server** → **Databases**.
2. Choose the database you are working with and expand its **Schemas**.
3. Click the schema that contains the table to which you want to add the trigger.
4. Right-click the Triggers icon and Select **NEW** → **External**.

Figure 8-1 on page 130 shows the information that has to be provided in the General tab when registering an external trigger.

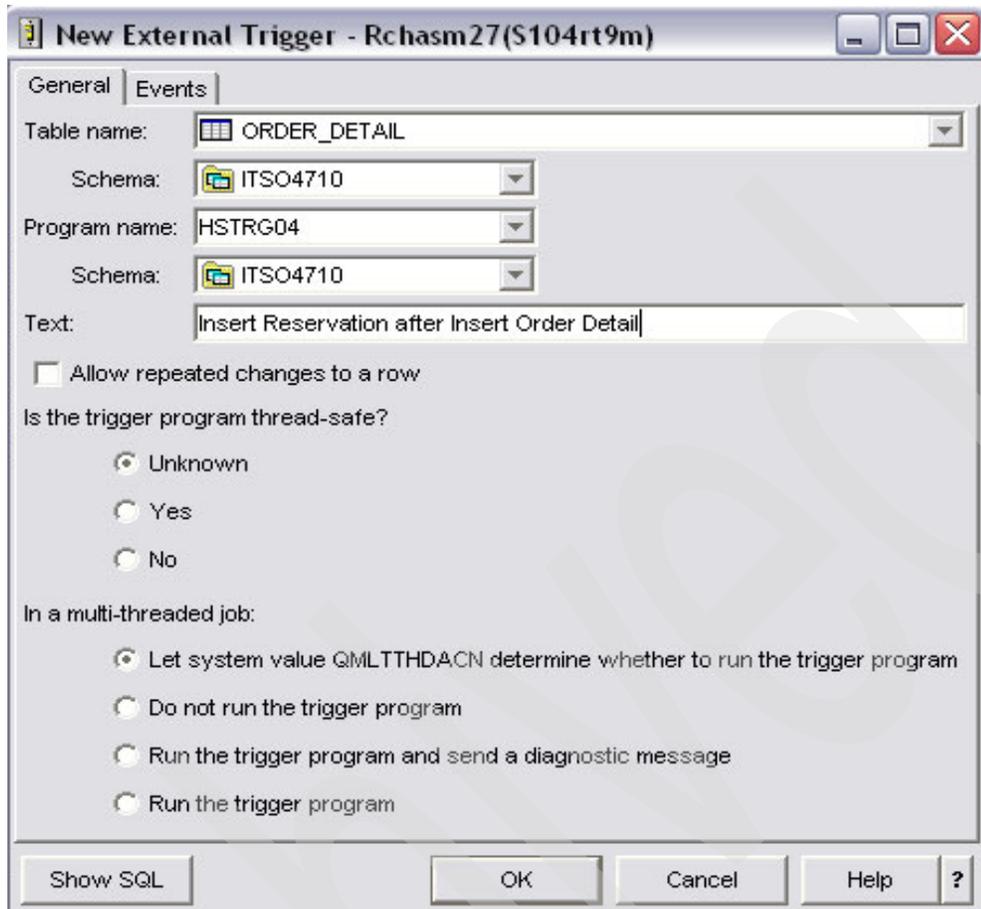


Figure 8-1 General tab for external triggers

Figure 8-2 on page 131 shows the information required in the Events tab when registering an external trigger.

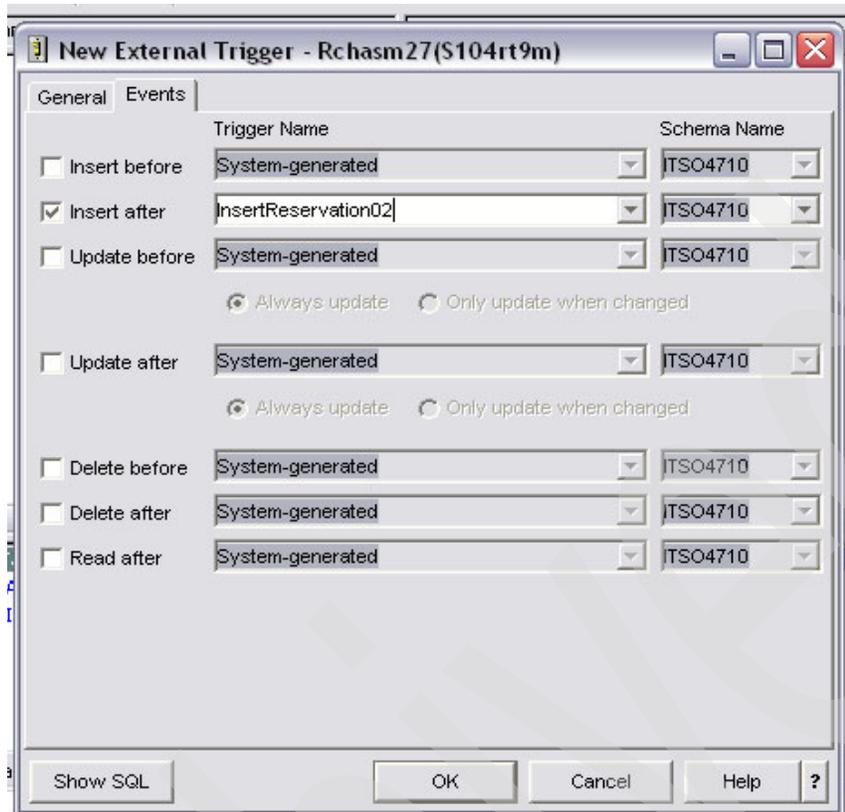


Figure 8-2 Events tab for external triggers

### 8.1.4 SQL triggers

While external triggers are written in an HLL and activated at record or row level, SQL triggers provide much more functionality. But the greatest advantage of using SQL triggers is portability. You can often use the same SQL trigger across other Relational Database Management Systems (RDBMS), because the implementation of SQL triggers is based on the SQL standard.

An SQL trigger can be generated by using the SQL statement CREATE TRIGGER.

**Note:** Contrary to external triggers, SQL triggers are automatically registered by executing the CREATE TRIGGER statement.

When a trigger is created, SQL creates a temporary source file that will contain C source code with embedded SQL statements. A program object is then created using the CRTSQLCI and CRTPGM commands. From release V5R1 on, an Internal C Compiler is shipped with the system. The internal compiler allows customers to create SQL triggers without having to purchase the C Compiler, even if the user does not have the ILE C product installed.

The program is created with activation group ACTGRP(\*CALLER). This makes sure that your program runs under the same commitment control level as the program or procedure that fired the trigger. The activation group is always considered as the Unit of Work. Besides commitment control can be started with commitment scope on job level, the default value for the commitment scope in the CL command STRCMTCTL (Start Commitment control) is \*ACTGRP (Activation Group).

The SQL options used to create the program are the options that are in effect at the time the CREATE TRIGGER statement is executed.

You can specify your compile options directly in the trigger by using the SET OPTION statement in your trigger program.

## Structure of an SQL trigger

An SQL trigger can be created by either specifying the CREATE TRIGGER SQL statement or by using iSeries Navigator.

The CREATE TRIGGER statement is one single command, and consists of two parts:

- ▶ The control information
- ▶ The SQL trigger body

The following example shows the structure of an SQL trigger.

### *Example 8-9 Structure of an SQL trigger*

---

```
CREATE TRIGGER TriggerName
  BEFORE/AFTER INSERT/UPDATE/DELETE
  ON TableName
  REFERENCING Old/New Record
  FOR EACH ROW / FOR EACH STATEMENT
  MODE DB2ROW / DB2SQL
  SET OPTION
  WHEN (search condition)

  Trigger body
```

---

### **Trigger control information**

The trigger control information is:

- ▶ Trigger name

The name, including the implicit or explicit qualifier, must not be the same as a trigger that already exists at the current server.

**Note:** QTEMP cannot be used as the trigger-name schema qualifier. Do not use names that begin with 'SQ', 'SQL', 'RDI', or 'DSN'. These names are reserved for the database manager.

Because a trigger is directly linked to the database, it is preferable to create the trigger in the same schema as the table is located. When saving and restoring your schema, the triggers are correctly activated. When located in different schemas, the triggers must be recreated or added to the table.

- ▶ Trigger time/trigger event

Trigger time BEFORE or AFTER, depending on when your trigger should be activated, must be specified.

The trigger event that fires the trigger, INSERT, UPDATE, or DELETE must be specified.

Update triggers can be defined on column level by adding OF ColumnName1, ... ColumnNameN to the UPDATE event. This means that the trigger is only be fired if changes in the listed columns are executed.

Example 8-10 on page 133 shows an AFTER UPDATE trigger that is fired when the OrderDtl\_Quantity is changed.

*Example 8-10 After update trigger at the column level*

---

```
Create Trigger ITS04710/UpdateReservation01
  AFTER UPDATE of OrderDt1_Quantity on ITS04710/ORDER_DETAIL
  Referencing OLD ROW as O
             NEW ROW as N
  For Each Row
  Mode DB2ROW
  When (O.OrderDt1_Quantity <> N.OrderDt1_Quantity)
  BEGIN ATOMIC
  Update ITS04710/Reservation R
    set Reserved_Quantity =  R.Reserved_Quantity
                          - O.OrderDt1_Quantity
                          + N.OrderDt1_Quantity
  Where   R.Order_Number = N.Order_Number
         and R.Product_Number = N.Product_Number;
END;
```

---

► Referencing

you can specify a correlation name for the triggered record image. In your trigger program you can get access on the values by specifying `CorrelationName.Column`.

The OLD ROW image is only available for update and delete triggers. The NEW ROW image is only available for insert and update triggers.

– OLD ROW AS correlation name

This specifies a correlation name that identifies the values in the row prior to the triggering SQL operation.

– NEW ROW AS correlation name

This specifies a correlation name that identifies the values in the row prior to the triggering SQL operation.

Example 8-11 shows a before insert trigger that updates the current time and user in the inserted row.

*Example 8-11 SQL trigger with referencing old/new record*

---

```
Create Trigger ITS04710/MyTrgTable01
  Before Insert on MyTrgTable
  Referencing NEW ROW as Ins
  For Each Row
  Mode DB2ROW
  BEGIN Atomic
  set Ins.InsertTime = Current_Timestamp;
  set Ins.InsertUser = User;
END;
```

---

► Granularity

– FOR EACH ROW

The trigger is fired with each row that is affected by the event. This method must be selected if values from the old and new row values must be compared. For example, you want to write a transaction file where all changes are stored.

With FOR EACH ROW Granularity, the operation sequence is:

```
Modify row 1
Call Trigger row 1
Modify row 2
Call Trigger row 2
```

```
...
Modify last row
Call Trigger last row
```

– FOR EACH STATEMENT

The trigger is fired only once, when all affected rows are handled. The trigger is even fired when no records are changed by the event.

FOR EACH STATEMENT cannot be used for BEFORE triggers, for example, if you want to write or update a summary file with the new values.

With FOR EACH STATEMENT Granularity, the operation sequence is:

```
Modify row 1
Modify row 2
...
Modify last row
Call Trigger
```

► Mode

– DB2ROW

The triggers are activated on each row operation.

DB2ROW is valid for both BEFORE and AFTER trigger.

The DB2ROW mode causes the trigger to be fired after each row change. Not surprisingly, this mode can only be specified on row-level triggers. Rows that are not yet modified by the operation appear to the trigger program to have their original values, while rows already modified show the new values.

In DB2ROW mode, the operation sequence is:

```
Modify row 1
Call trigger for row 1
Modify row 2
Call trigger for row 2
...
Modify last row
Call trigger for last row
```

FOR EACH STATEMENT cannot be specified for a MODE DB2ROW trigger.

– DB2SQL

The DB2SQL mode waits until all of the row changes are made and then calls the trigger. The DB2SQL mode triggers are activated after all of the row operations occur. The DB2SQL mode specifies that the trigger program will not run for any row until after all rows affected by the operation are modified. All rows appear to the trigger program to have their new values.

In DB2SQL mode, the operation sequence is:

```
Modify row 1
Modify row 2
...
Modify last row
Calls trigger for row 1
Calls trigger for row 2
...
Calls trigger for last row
```

Example 8-12 on page 135 shows a trigger that is only activated once per statement.

*Example 8-12 SQL trigger with FOR EACH STATEMENT clause*

---

```
Create Trigger ITS04710/InsertOrderHeaderTotal
  After Insert on ITS04710/ORDER_DETAIL
  Referencing NEW_TABLE as NewDetail
  For Each Statement
  Mode DB2SQL
  Set Option DbgView = *SOURCE
BEGIN ATOMIC
  Declare SumTotal Decimal(7, 0);
  select coalesce(sum(NewDetail.OrderDt1_Total), 0)
    into SumTotal
    from NewDetail;
  update Order_Header
    set Order_Total = Order_Total + SumTotal ;
END;
```

---

► **WHEN(SearchCondition)**

It is not only possible to define an SQL trigger at the column level, but you can also specify conditions under which circumstances a trigger is executed. Each column value that is stored in either the OLD or NEW row can be compared. For example, you define two triggers to handle the raise and the decrease of a certain value, because different actions are necessary.

Example 8-13 shows a trigger that is fired when the Order Detail quantity is changed, but only if the new quantity is higher than the old one.

*Example 8-13 Conditional trigger*

---

```
Create Trigger ITS04710/RaiseReservation01
  AFTER UPDATE of OrderDt1_Quantity on ITS04710/ORDER_DETAIL
  Referencing OLD ROW as O
             NEW ROW as N
  For Each Row
  Mode DB2ROW
  When (O.OrderDt1_Quantity < N.OrderDt1_Quantity)
BEGIN ATOMIC
  Update ITS04710/Reservation R
    set Reserved_Quantity = R.Reserved_Quantity
                          - O.OrderDt1_Quantity
                          + N.OrderDt1_Quantity
  Where   R.Order_Number = N.Order_Number
         and R.Product_Number = N.Product_Number;
END;
```

---

**Trigger body**

The trigger body contains all the executable statements. When a trigger is created, SQL creates a temporary source file that contains C source code with embedded SQL statements that are specified in the trigger body. A program object is then created using the CRTSQLCL and CRTPGM commands.

If your trigger consists only of one executable statement, you simply add it to the trigger control information.

Example 8-14 shows a BEFORE INSERT trigger, where only one single statement is executed. The next serial number for a key is calculated and inserted in the new row.

*Example 8-14 Before insert trigger with a single SQL statement*

---

```
Create Trigger ITS04710/MyTrgTable02
```

```

BEFORE INSERT on MyTrgTable
Referencing NEW as INS
For Each Row
Mode DB2ROW
Select Coalesce(Max(a.CurrNbr) + 1 , 1)
      into INS.CurrNbr
      from MyTrgTable a
      where a.Text = INS.Text;

```

---

If several SQL statements must be executed in one single trigger program, they must be embedded in a compound statement. A compound statement starts with BEGIN and ends with END.

All SQL statements inside this trigger body must be ended with a semi colon (;).

For more information about SQL Programming Language, refer to “SQL programming language” on page 163.

Example 8-15 shows a trigger where several statements are embedded in a single compound statement. The product price is determined from the stock table, and the total, multiplying quantity and price is calculated and updated in the new row.

*Example 8-15 SQL trigger with trigger body*

---

```

Create Trigger ITS04710/CalcOrderDetailTotal
  Before Insert on ITS04710/ORDER_DETAIL
  Referencing NEW as N
  For Each Row
  Mode DB2ROW

  BEGIN ATOMIC
  Declare Price Decimal(5, 0);
  Select coalesce(S.Product_Price, 0)
        into Price
        from Stock S
        Where S.Product_Number = N.Product_Number;
  Set N.OrderDt1_Total = Price * N.OrderDt1_Quantity;
  END;

```

---

### Creating SQL triggers using iSeries Navigator

To generate SQL triggers with iSeries Navigator, the following steps are necessary:

1. In the iSeries Navigator window, expand your server and select **Databases**.
2. Choose the database you are working with and expand its Schemas.
3. Click the schema that contains the table to which you want to add the trigger.
4. Right-click the Triggers icon and select **NEW** → **SQL**.

Figure 8-3 on page 137 shows the General tab for SQL triggers.

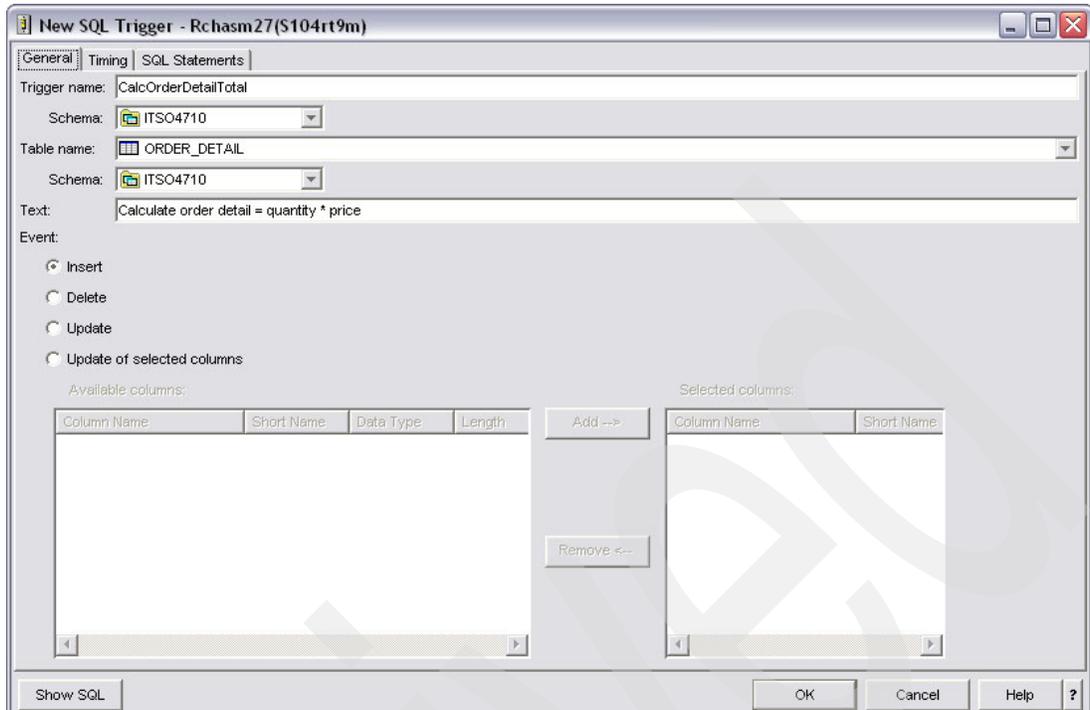


Figure 8-3 General tab for SQL triggers

Figure 8-4 shows the Timing tab for SQL triggers.

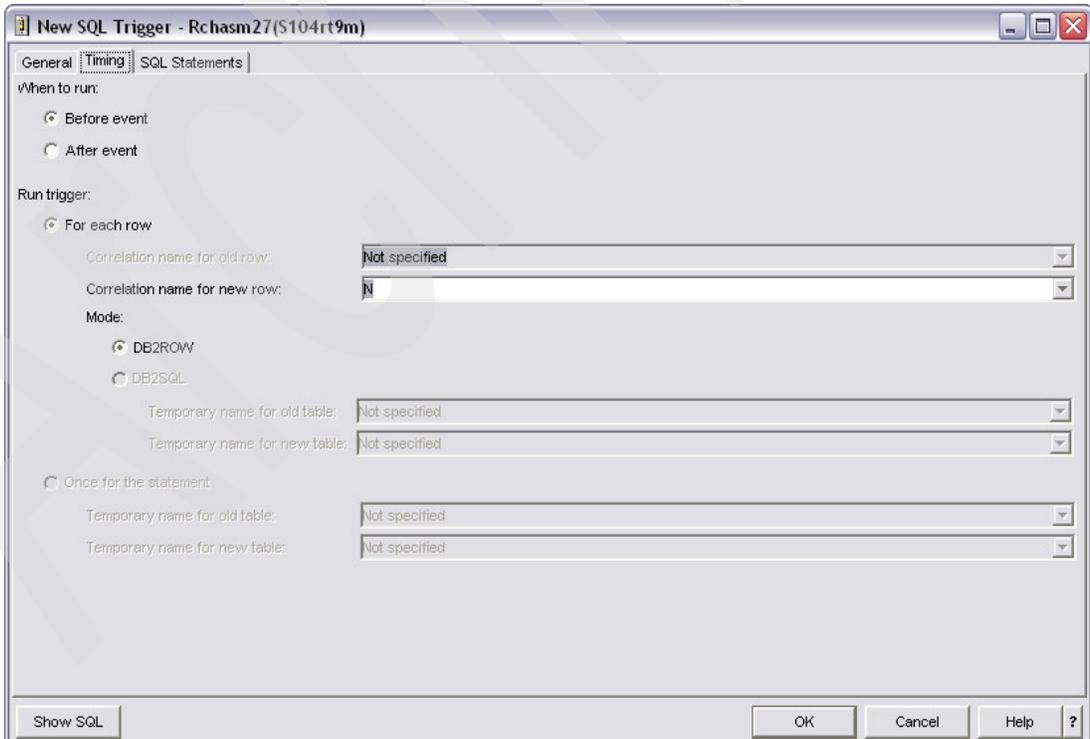


Figure 8-4 Timing tab for SQL triggers

Figure 8-5 on page 138 shows the trigger body for SQL triggers.

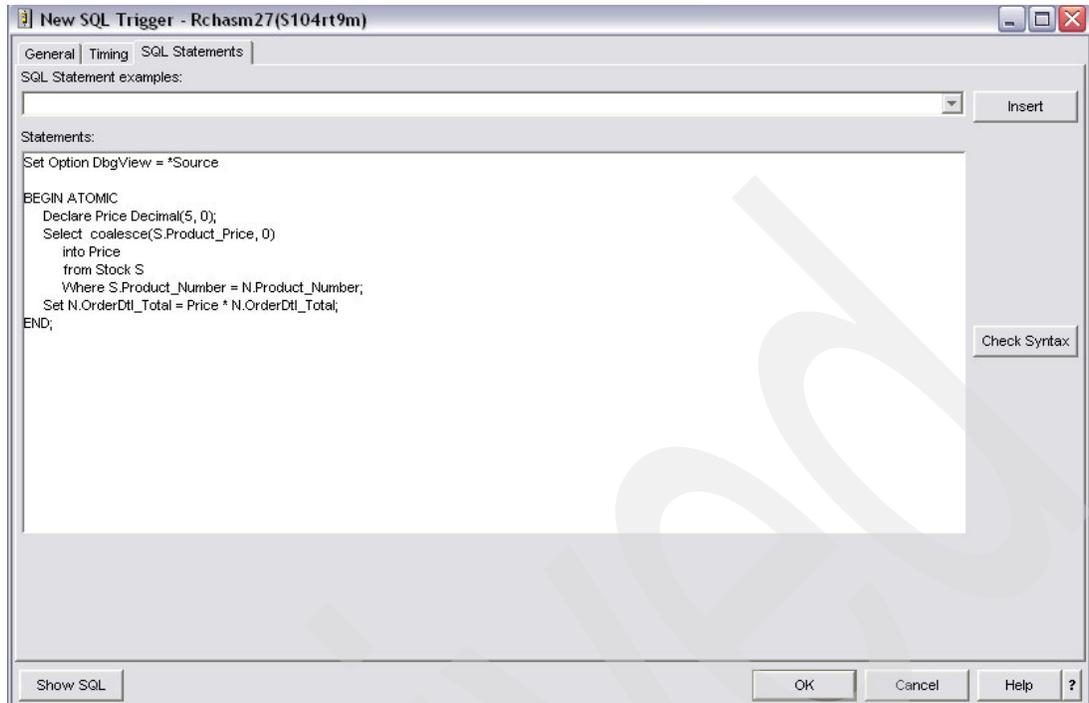


Figure 8-5 SQL trigger body

### 8.1.5 Getting information about triggers

To determine which triggers are registered for a specific table, you can use one of the following methods:

- ▶ CL command Display File description DSPFD

Using the CL command DSPFD with option TYPE = \*TRG will list all triggers and their characteristics.

The following example shows how all triggers on the Order Detail file can be listed.

```
DSPFD FILE(ORDERDTL)
      TYPE(*TRG)
```

- ▶ Querying the system tables

<b>SYSTRIGGERS</b>	Provides all information about the triggers
<b>SYSTRIGDEP</b>	Provides all information about dependencies between files used in SQL triggers
<b>SYSTRIGCOL</b>	Provides information about columns used in SQL triggers
<b>SYSTRIGUPD</b>	Provides information about updated columns used in SQL triggers.

For more information on triggers refer to the redbook *Stored Procedures, Triggers and User Defined Functions in DB2 UDB for iSeries*, SG24-6503.

## 8.2 Stored procedures

Stored procedures are programs or service programs containing a specific signature, so that they can be called from any SQL interface, such as interactive and embedded SQL, ODBC, and JDBC.

In contrast to triggers, which are directly linked to the database tables, stored procedures must be called explicitly by using the SQL CALL statement. When a stored procedure is called, it is embedded in the call stack and executed. If the stored procedure ends, either normally or abnormally, the control is returned to the caller. In this way it is possible to interchange parameter information between caller and stored procedure.

Stored procedures can be called locally (on the same system where the application runs) or remotely on a different system. They are the easiest way to perform a remote call and to distribute the execution logic of an application program.

Stored procedures offer a number of powerful advantages for distributed application development. These include the following;

- ▶ Common business functions can be encapsulated in stored procedures and made universally accessible, promoting code re-use and consistency, and providing support for object-oriented application design.
- ▶ Performance can be significantly improved for distributed applications that require several SQL calls to be made by the client against a remote database. Instead of multiple trips across the network for each of these requests, they can be combined and executed within a stored procedure so only one single call is needed. This performance improvement can also create subsequent benefits in reducing lock contention.
- ▶ Security can be enhanced, as developers are only able to work with the stored procedure input and output parameters, and are prevented from viewing or altering the underlying code that implements the business function. Stored procedures can help to control the access to database objects.

There are two types of stored procedures:

- ▶ External stored procedures
- ▶ SQL stored procedures

## 8.2.1 External stored procedures

External stored procedures are programs or service programs written in an HLL with a unique signature to be called from any SQL interface, like embedded SQL, ODBC, JDBC, etc.

Programs do not need to be registered, as long as you do not want to overload the procedures (look at “Procedure signature and overloading” on page 148). They can be called directly by the SQL interfaces using the SQL command CALL.

Programs or service programs are registered as stored procedures by using the SQL command CREATE PROCEDURE.

**Note:** Since Release V5R3M0, procedures in service programs without return value can be registered as stored procedures.

To register procedures in service programs, the option EXTERNAL NAME in the CREATE PROCEDURE statement (look at “SQL statement CREATE PROCEDURE” on page 144), must include the procedure’s entry point (procedure name).

The activation group of the program or service program is inherited to the stored procedure, which means that if your program runs in a named activation group, the stored procedure uses the same activation group.

The activation group is considered a unit of work. This is especially important with regard to the commitment control. The default value for the commitment control (CMTSCOPE) in the CL command Start Commitment Control (STRCMTCTL) is activation group (\*ACTGRP).

If your program or service program is generated with a named activation group or a new activation group, and the commitment control is started at activation group level, commit and rollback will lead to unexpected results.

Therefore we recommended creating programs or service programs that must be registered as stored procedures with the activation group \*CALLER. If you want to use named activation groups, you have to start your commitment control on job level (CMTSCOPE = \*JOB).

**Note:** Commitment control can only be started once in an job, with commitment scope either Job or Activation group. If you have to change, you first have to end commitment control using the CL command ENDCMTCTL and restart it again with a different commitment scope.

In contrast to JAVA or .Net languages, RPG cannot directly receive result sets. But nevertheless it is possible to return result sets from RPG. There are two methods to realize this:

- ▶ Define a cursor with WITH RETURN and open it before you end your program or procedure.
- ▶ In your RPG program, you can fill your data into a multi-occurrence data structure or an array data structure and then use the SQL command SET RESULT SETS to return the data structure as result set.

**Note:** If you have to receive result sets within RPG you have to use the Common Level Interface (CLI) APIs.

### Registering stored procedures using iSeries Navigator

To register external stored procedures with iSeries Navigator, the following steps are necessary:

1. In the iSeries Navigator window, expand your server, then select **Databases**.
2. Choose the database you are working with and expand its Schemas.
3. Right-click **Procedure**.
4. Select **NEW** → **External**.

Figure 8-6 on page 141 shows the General tab for external triggers.

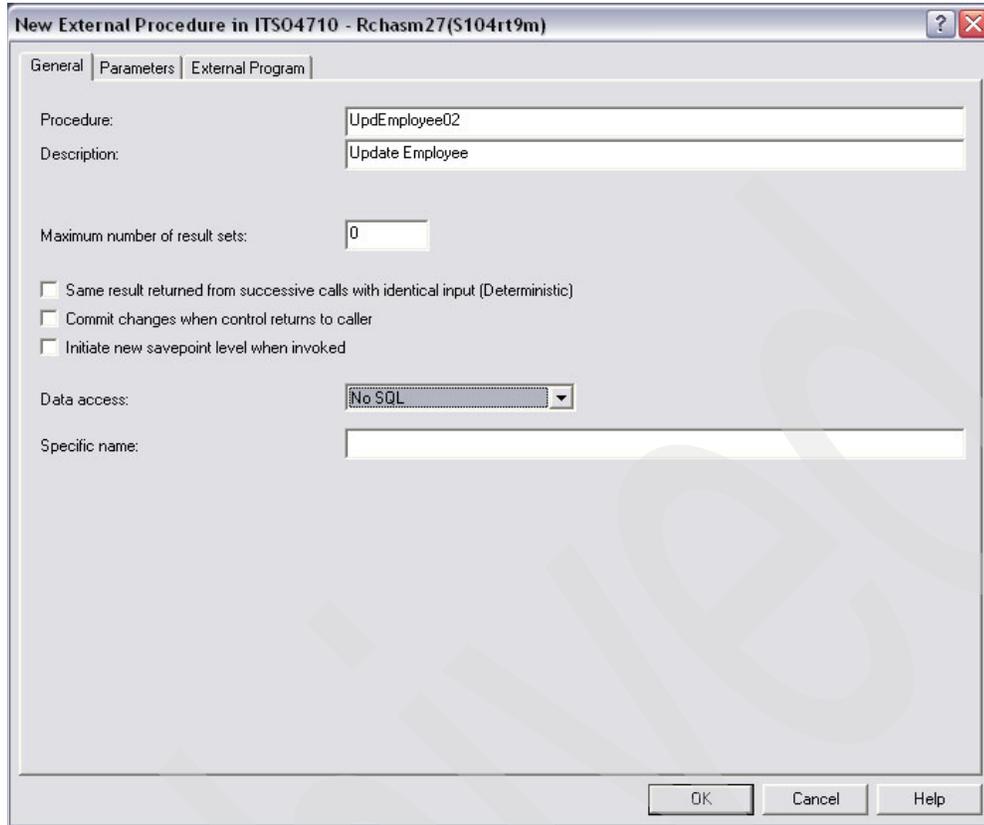


Figure 8-6 General requirements for external triggers

Figure 8-7 on page 142 shows the Parameters window for external procedures.

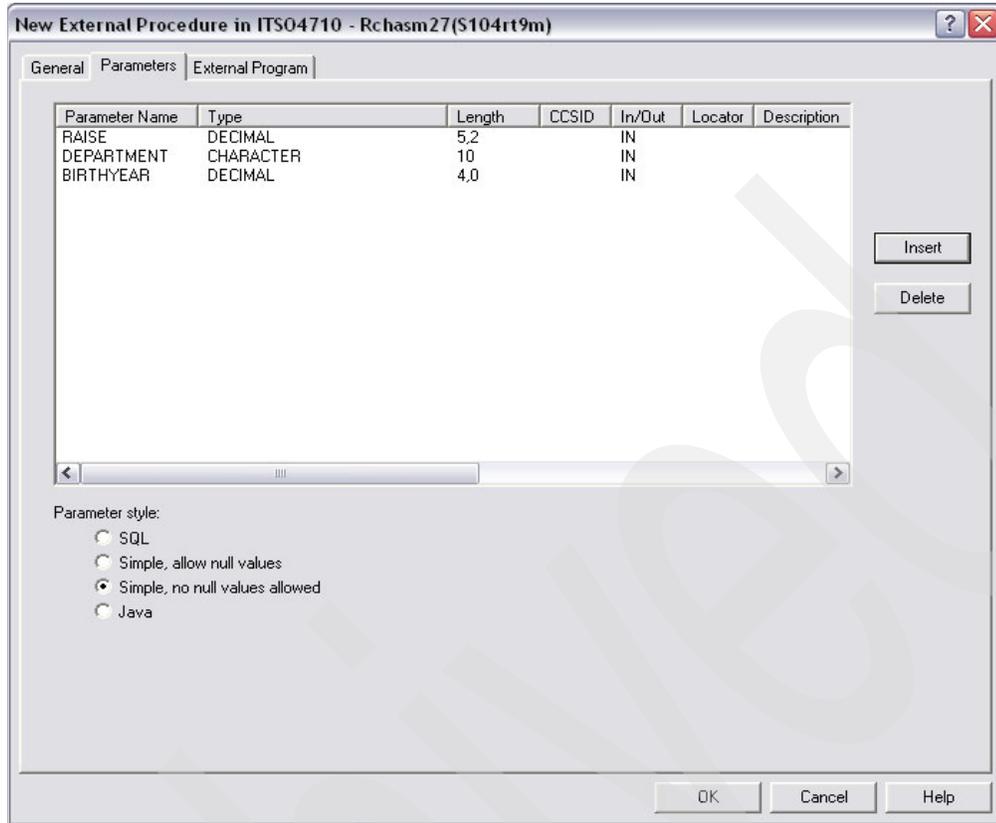


Figure 8-7 Parameter definition for external stored procedures

Figure 8-8 on page 143 shows the Program window for an external trigger.

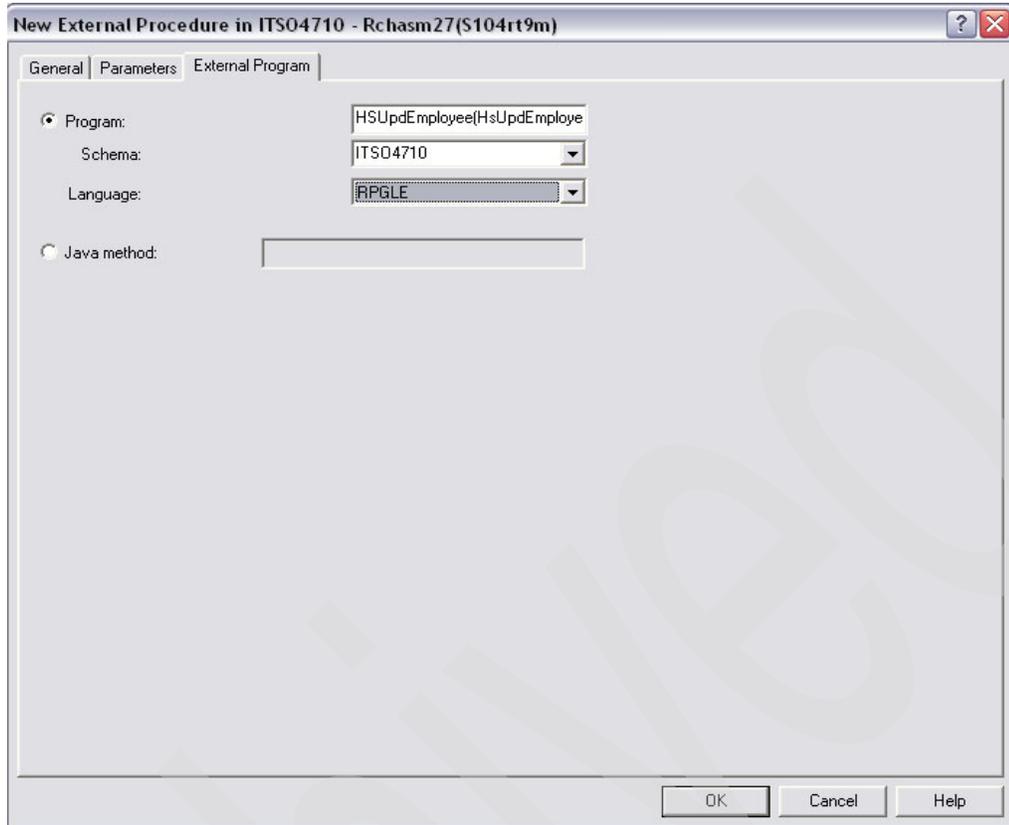


Figure 8-8 Program information for external triggers

## 8.2.2 SQL stored procedures

SQL stored procedures are generated by using the SQL command CREATE PROCEDURE, like external stored procedures. But in contrast to the external procedures that point to a program or service program, the program code is embedded in the CREATE PROCEDURE and written only by using SQL commands.

In the program body all SQL statements and scalar functions can be used. Additionally, SQL/PSM language provides a couple of control statements (for more information about SQL programming language look at “SQL programming language” on page 163).

When creating the procedure, a temporary source file is generated, containing the SQL statements converted into API calls in the C-language. From this source member a C-Program Object is created.

The activation group of SQL stored procedures is always \*CALLER.

### Creating SQL stored procedures with iSeries Navigator

To register SQL stored procedures with iSeries Navigator, the following steps are necessary:

1. In the iSeries Navigator window, expand your server and then select **Databases**.
2. Choose the database you are working with and expand its Schemas.
3. Select the Schema.
4. Right-click **Procedure**.
5. Select **NEW** → **SQL**.

The General tab and the Parameters tab are identical to the external stored procedures, but instead of external program information you have to enter your SQL statement.

Figure 8-9 shows the SQL statements for an SQL stored procedure.

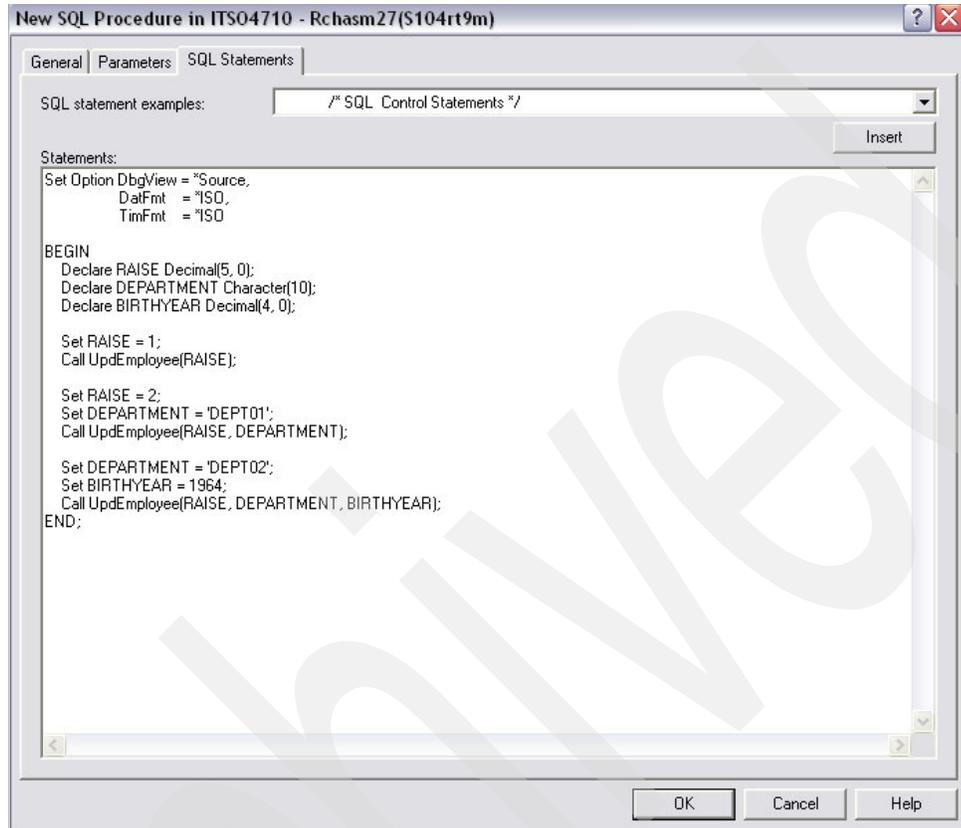


Figure 8-9 SQL stored procedure SQL statements

### 8.2.3 SQL statement CREATE PROCEDURE

The CREATE PROCEDURE statement can be used either to register programs or service programs written in an HLL as a stored procedure or to create a SQL stored procedure.

The text below shows the most important options of the create procedure statement. For more information see the SQL Reference.

```
CREATE PROCEDURE ProcedureName  
  (Parameter Declaration)  
  LANGUAGE  
  PARAMETER STYLE  
  SQL DATA  
  DYNAMIC RESULT SETS  
  EXTERNAL NAME  
  SPECIFIC  
  COMMIT ON RETURN
```

► Procedure name

This is the name that is used to call the stored procedure. It can be defined with up to 128 characters.

The combination of schema, procedure name, and number of parameters must be unique on the current server.

If you want to generate external stored procedures, it is not necessary that the procedure name and the name of the program or service program are identical. The external name of the program or procedure must be declared with the EXTERNAL NAME option.

► **Parameter declaration**

When defining a parameter, you have to determine if it is input only, output only, or both by, using one of the following modes:

- IN for input-only parameters
- OUT for output-only parameters
- INOUT for a parameter that is both input and output capable

Further, the parameter name and the data type must be specified.

Example 8-16 shows the registration of an external procedure with several input and output parameters.

*Example 8-16 Defining parameters in SQL stored procedures*

---

```
Create Procedure ITS04710/GetOrderSummary
  (IN OrderNo      Char(5) ,
   OUT OrderDate   DATE ,
   OUT DeliveryDate DATE ,
   OUT NoOfPositions INTEGER ,
   OUT NoOfProducts INTEGER ,
   OUT TotalAmount DECIMAL(11, 2) )
LANGUAGE RPGLE
SPECIFIC ITS04710/GetOrdSum
NOT DETERMINISTIC
NO SQL
CALLED ON NULL INPUT
EXTERNAL NAME 'ITS04710/GETORDSUM'
PARAMETER STYLE GENERAL ;
```

---

► **DYNAMIC RESULT SETS Number**

Must be specified when the stored procedure returns one or more result sets.

If you add WITH RETURN to your cursor declaration, it indicates that the cursor is intended for use as a result set from a stored procedure.

You must open the cursor before it can be returned to the caller.

The result sets are scrollable if a cursor is used to return a result set and the cursor is scrollable. If a cursor is used to return a result set, the result set starts with the current position. Thus, if five FETCH NEXT operations have been performed prior to returning from the procedure, the result set will start with the sixth row of the result set.

Example 8-17 shows a stored procedure that returns a single result set.

*Example 8-17 Stored procedure returning a result set*

---

```
Create Procedure ITS04710/RetResSet
  (IN OrdYear Decimal(4, 0) )
  Dynamic Result Sets 1
Language SQL
Reads SQL Data
BEGIN
  Declare C1 insensitive Scroll Cursor WITH RETURN
  For select Month(Order_Date) as ResMonth,
           Supplier_Name, Sum(Product_Price * OrderDt1_Quantity)
  from ((ORDERHDR h
        join Orderdt1 d on h.Order_Number = d.Order_Number)
        join Stock p on d.Product_Number = p.Product_Number)
```

```
        join Supplier s on p.Supplier_Number = s.Supplier_Number
Where   Year(Order_Date) = OrdYear
Group By Month(Order_Date), Supplier_Name
Order By Month(Order_Date), Supplier_Name;
```

```
OPEN C1 ;
END ;
```

**Note:** With embedded SQL you cannot access result sets. The only way to receive result sets within RPG is to use the Common Level Interface (CLI) APIs.

You cannot get direct access to result sets. RPG can return result sets through an open cursor or by using the SQL statement SET RESULT SETS.

► Language

Specifies the language in which the external program that must be registered is written. The following languages can be specified:

- C
- C++
- CL
- COBOL or COBOLLE for ILE COBOL programs
- FORTRAN
- JAVA
- PLI
- REXX
- RPG for RPG/400 programs or RPGLE for ILE RPG programs

For SQL stored procedures, SQL must be specified.

► Parameter-Style

This option is only important for external stored procedures. Depending on the parameter style, DB2 UDB for iSeries passes a number of additional parameters. You can specify the required parameter style when the procedure is created. DB2 UDB for iSeries supports the following parameter styles:

- GENERAL

Specifies that the procedure will use a parameter passing mechanism where the procedure receives the parameters specified on the CALL. No additional parameters are added.

- GENERAL WITH NULLS

Specifies that in addition to the parameters on the CALL statement as specified in GENERAL, another argument is passed to the procedure. This additional argument contains an indicator array with an element for each of the parameters of the CALL statement.

If a NULL value was passed for the corresponding argument, the indicator variable contains -1. If a valid value is passed, the indicator variable contains 0.

- SQL

Specifies that in addition to the parameters on the CALL statement as specified in GENERAL, other arguments are passed to the procedure. These are:

- A CHAR(5) output parameter for SQLSTATE.
- A VARCHAR(517) input parameter for the fully qualified procedure name.
- A VARCHAR(128) input parameter for the specific name.
- A VARCHAR(70) output parameter for the message text.

Parameter style SQL cannot be used with programming language JAVA.

- DB2SQL

The DB2SQL style is a superset of the SQL parameter style.

- DB2GENERAL

Specifies that the procedure will use a parameter passing convention that is defined for use with Java methods.

- JAVA

Specifies that the procedure will use a parameter passing convention that conforms to the Java language and SQLJ Routines specification.

- ▶ SQL DATA

Registering a stored procedure, you have to specify if SQL statements are used, by specifying one of the following options:

- NO SQL DATA

The stored procedure does not contain SQL statements. This must be used for your RPG or COBOL programs or service programs that do not contain embedded SQL.

- CONTAINS SQL DATA

This option must be used if you want to register a RPG or COBOL program or service program that contains SQL statements like SET, CALL, and COMMIT, but does not access database data.

If you create an SQL stored procedure that only executes calls to other stored procedures and set parameters, you have to specify this option, too.

- READS SQL DATA

This option must be used if you want to register a RPG or COBOL program, where you get access to database data by using the select statement, but no insert, update, or delete with SQL is performed.

If you write a SQL stored procedure that only returns result sets, you have to specify this option, too.

- MODIFIES SQL DATA

This option must be specified if you are inserting, updating, or deleting data through SQL in your stored procedure.

- ▶ EXTERNAL NAME

Specifies the program or service program that will be executed when the procedure is called by the CALL statement. The program name must identify a program or service program that exists at the application server at the time the procedure is called.

If you want to register a service program, you have to add the procedure name that is called to the external name. It must even be specified if the service program consists only of one single procedure with the same name.

Example 8-18 shows the registration of the procedure HSEMPLOYEE in the service program HSEMPLOYEE.

*Example 8-18 Registering a service program as stored procedure*

---

```
Create Procedure ITS04710/UpdEmployee
  (IN Raise Decimal(5, 2))
  Language RPGLE
  Not Deterministic
  No SQL
```

Called on NULL input  
External Name 'ITS04710/HSEMPLOYEE(HSEMPLOYEE) '  
Parameter Style General;

---

► **SPECIFIC NAME**

DB2 Universal Database for iSeries identifies each stored procedure with a specific name that, combined with the specific schema, must be unique in the system. This gains importance because multiple stored procedures with the same name but different signatures must have different specific names or signatures. If you do not provide a specific name, DB2 UDB for iSeries generates one automatically.

If you do not overload your procedure, the generated specific name will be the procedure name. When overloading, the specific name will be generated by using the system conventions, character 1-5 from the procedure name and a serial number.

► **COMMIT ON RETURN**

Specifies whether the database manager commits the transaction immediately on return from the procedure.

If COMMIT ON RETURN YES is specified, the database manager issues a commit if the procedure returns successfully. If the procedure returns with an error, a commit is not issued.

## 8.2.4 Procedure signature and overloading

DB2 Universal Database for iSeries supports the concept of procedure overloading. This means that you can have several procedures with the same name in the same schema, provided they have different signatures. The signature is generated by executing the SQL CREATE PROCEDURE statement.

The signature of a procedure is determined by the qualified name and the number of parameters in the procedure. A signature is unique in one schema.

Procedures with the same name and identical number of parameters cannot coexist in the same schema, even if the parameters have different data types. But it is possible to register the same program or service program with the same parameters in another procedure in the same library.

**Note:** You do not have to register programs as stored procedures as long as you do not want to overload procedures. Programs can be directly called by the SQL CALL statement, without being registered as stored procedures.

### Overloading ILE Programs and procedures

Procedure overloading must be used if you want to register ILE Programs with optional parameters. Optional parameters are generated by using the keyword OPTIONS(\*NoPass) in the prototype declaration.

Example 8-19 on page 149 updates the table MyEmployee. Depending on the passed parameters, different selections are performed.

If only parameter 1, RAISE is specified, all rows are updated.

If RAISE and DEPARTMENT are specified only those rows that belong to the passed department are updated.

If all three parameters are passed, all rows with the specified department and where the year of the birthday is equal to the passed year are updated.

The source is compiled into a module by using the CRTRPGMOD command and then bound into a service program by using the CRTSRVPGM command.

*Example 8-19 ILE RPG program with optional parameters*

```

H DEBUG Option(*NoDebugIO)
*-----
FMyEmployeeUF  E          K DISK
*-----
D UpdEmployee  PR          ExtProc('HSEMPLOYEE')
D  ParRaise    5P 2        const
D  ParDept     10A varying const options(*NoPass)
D  ParYear     4P 0        const options(*NoPass)

D UpdEmployee  PI
D  ParRaise    5P 2        const
D  ParDept     10A varying const options(*NoPass)
D  ParYear     4P 0        const options(*NoPass)
*-----
/free
  setLL *Zeros MyEmployer;

  DoU %EOF(MyEmployee);
    Read MyEmployer;

    If %EOF;
      leave;
    EndIf;

    If  ParRaise = *Zeros
      or (%Parms >= 2 and EmpDept <> ParDept)
      or (%Parms >= 3 and %SubDt(EmpBDay: *Y) <> ParYear);
      iter;
    Endif;

    EmpSal *= (1 + ParRaise/100);
    Update MyEmployer;

  EndDo;

  Return;
/End-Free

```

If we want to call the procedures with one, two, or three parameters, we have to register a procedure with the same name but a different number of parameters.

Example 8-20 shows the CREATE PROCEDURE statement for each of these procedures.

*Example 8-20 Overloading an external stored procedure*

```

Create Procedure ITS04710/UpdEmployee
  (IN Raise Decimal(5, 2))
  Language RPGLE
  Not Deterministic
  No SQL
  Called on NULL input

```

**External Name 'ITS04710/HSEMPLOYEE(HSEMPLOYEE)'**  
 Parameter Style General;

Create Procedure **ITS04710/UpdEmployee**  
**(IN Raise Decimal(5, 2),**  
**IN Department Char(10))**  
 Language RPGLE  
 Not Deterministic  
 No SQL  
 Called on NULL input  
**External Name 'ITS04710/HSEMPLOYEE(HSEMPLOYEE)'**  
 Parameter Style General;

Create Procedure **ITS04710/UpdEmployee**  
**(IN Raise Decimal(5, 2),**  
**IN Department Char(10),**  
**IN BirthYear Decimal(4, 0))**  
 Language RPGLE  
 Not Deterministic  
 No SQL  
 Called on NULL input  
**External Name 'ITS04710/HSEMPLOYEE(HSEMPLOYEE)'**  
 Parameter Style General;

If you list the procedures in the iSeries Navigator, you will see three procedures with the same procedure name, but different signatures. Figure 8-10 shows the procedures listed in the schema ITS04710.

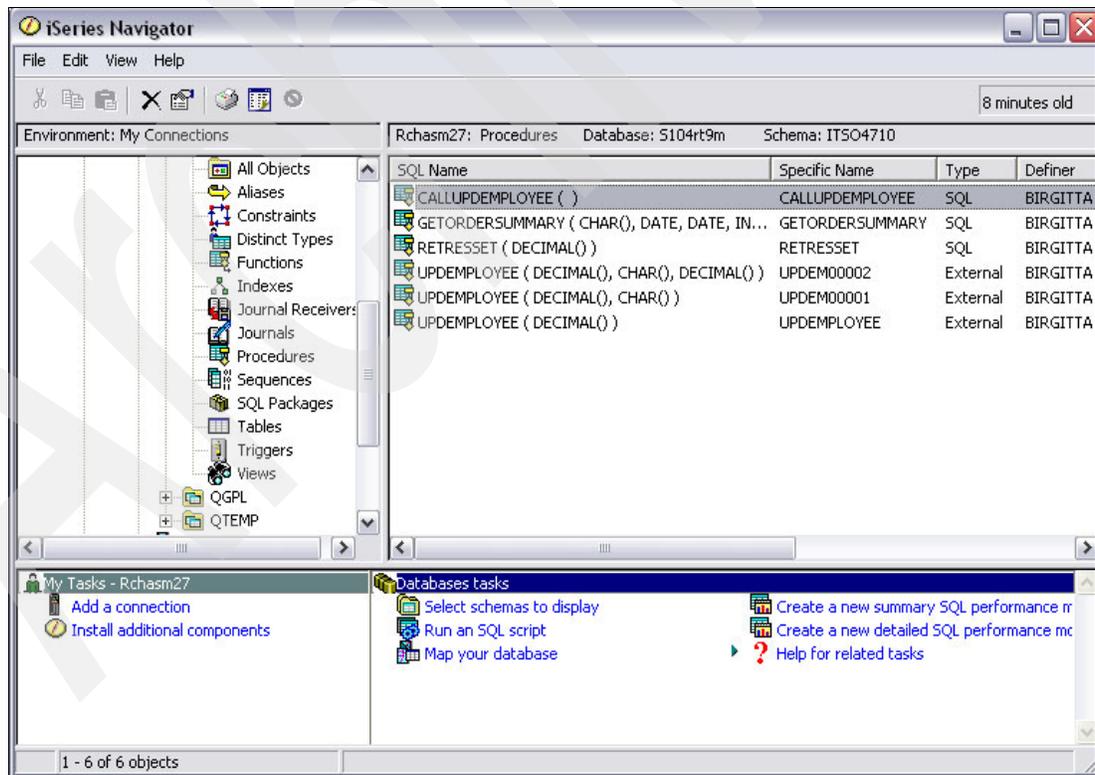


Figure 8-10 Overloaded stored procedures

Example 8-21 on page 151 shows a procedure that calls the procedure UpdEmployee with different parameters.

### Example 8-21 Calling overloaded procedures

---

```
Create Procedure ITS04710/CallUpdEmployee
    Language SQL
    Not Deterministic
    Contains SQL
    Called on NULL input

BEGIN
    Declare RAISE Decimal(5, 0);
    Declare DEPARTMENT Character(10);
    Declare BIRTHYEAR Decimal(4, 0);

    Set RAISE = 1;
    Call UpdEmployee(RAISE);

    Set RAISE = 2;
    Set DEPARTMENT = 'DEPT01';
    Call UpdEmployee(RAISE, DEPARTMENT);

    Set DEPARTMENT = 'DEPT02';
    Set BIRTHYEAR = 1964;
    Call UpdEmployee(RAISE, DEPARTMENT, BIRTHYEAR);
END;
```

---

## 8.2.5 Deleting or replacing a stored procedure

SQL does not provide a REPLACE or CHANGE PROCEDURE statement.

If you try to create a procedure with the same signature as an existing procedure (identical name and identical number or parameters), the procedure will not be created. An error occurs:

```
SQL State: 42723
Vendor Code: -454
Message: [SQL0454] Routine MYPROCEDURE in MYSCHEMA already exists.
```

If you create a procedure with the same name as an existing procedure, but with a different number of parameters, the procedure is overloaded, which means that a procedure with a different signature is created.

If you want to replace a procedure, you first have to delete the old one by using the SQL statement DROP PROCEDURE. As long as the procedure is not overloaded, you simply specify DROP PROCEDURE and add the ProcedureName.

There are two methods to delete overloaded procedures:

- ▶ Adding the data types and the length of the parameters of the procedure after the ProcedureName in the DROP PROCEDURE statement
- ▶ Using the specific name or signature as follows:

```
DROP SPECIFIC PROCEDURE SpecificName
```

Example 8-22 shows how stored procedures can be deleted.

### Example 8-22 Deleting stored procedures

---

```
-- Deleting a procedure that is not overloaded
DROP PROCEDURE UpdEmployee;
```

```
-- Deleting an overloaded procedure
DROP PROCEDURE UpdEmployee(Dec(5, 2), Char(10));

DROP SPECIFIC PROCEDURE UPDEM00004;
```

---

## 8.2.6 Getting information about stored procedures

Information about external and SQL stored procedures is saved in the following system tables:

<b>SYSPROCS</b>	Provides all information about stored procedures. The program body for SQL stored procedures is saved in the ROUTINE_DEFINITON column.
<b>SYSPARMS</b>	Delivers the description of the parameters, including the order, parameter mode, data type, and length definition.
<b>SYSROUTINES</b>	The SYSROUTINES table contains one row for each procedure created by the CREATE PROCEDURE statement and each function created by the CREATE FUNCTION statement.

For more information refer to the redbook *Stored Procedures, Triggers and User Defined Functions on DB2 UDB for iSeries*, SG24-6309.

## 8.3 User defined functions

User defined functions (UDFs) are host - language functions for performing customized, frequently used tasks in applications. UDFs allows the programmers to modularize a database application, creating a function that can be used in SQL.

DB2 Universal Database for iSeries comes with a rich set of built-in functions, but users and programmers may have different particular requirements not covered by them. UDFs comes to play a very important role by allowing users and programmers to enrich the database manager by providing their own functions.

Some of the advantages of UDFs are:

- ▶ Customization

Functions specifically required by your application not existing in the set of DB2 built-in functions can be created. Whether the function is a simple transformation, a trivial calculation, or a complex multivariate analysis, you may choose a UDF to do the job.

- ▶ Flexibility

You can use functions with the same name in the same library that accepts different sets of parameters.

- ▶ Standardization

Many of the programs that you implement use the same basic set of functions, but there are minor differences in all the implementations. If you correctly implement your business logic as an UDF, you can reuse those UDFs in your other applications using SQL.

- ▶ Object-relational support

UDF also provides additional functions for User-defined Distinct Type (UDT) created in the database. UDFs act as methods for UDTs. More information on UDTs and how UDFs are used to encapsulate methods for them are in *DB2 UDB for AS/400 Object Relational Support*, SG24-5409.

- ▶ Performance

A UDF can run in the database engine and is very useful for performing calculations in the database manager server. Another area where performance may be increased is in dealing with Large Objects (LOBs). UDFs may be used for extracting or modifying portions of the information contained in a LOB directly in the database manager server instead of sending the complete LOB to the client side.

- ▶ Migration

When migrating from other database managers, there could be built-in functions that are not defined in DB2 Universal Database for iSeries. UDFs allow us to create those functions in order to make the migration process easier.

UDFs are useful for the following reasons:

- ▶ Supplement built-in functions: A user defined function is a mechanism with which you can write your own extensions to SQL. The built-in functions supplied with DB2 are a useful set of functions, but they may not satisfy all of your requirements. So, you may need to extend SQL. For example, porting applications from other database platforms may require coding of some platform-specific functions.
- ▶ Handle user-defined data types: You can implement the behavior of a User-defined Distinct Type (UDT) using UDFs. When you create a distinct type, the database provides only cast functions and comparison operators for the new type. You are responsible for providing any additional behavior. It is best to keep the behavior of a distinct type in the database where all of the users of the distinct type can easily access it. Therefore, UDFs are the best implementation mechanism for UDTs.
- ▶ Provide function overloading: Function overloading means that you can have two or more functions with the same name in the same library. For example, you can have several instances of the SUBSTR function that accept different data types as input parameters. Function overloading is one of the key features required by the object-oriented paradigm.
- ▶ Allow code re-use and sharing: A business logic implemented as a UDF becomes part of the database, and it can be accessed by any interface or application using SQL.

User defined functions can be invoked:

- ▶ In a SET statement to change the value of a variable
- ▶ In a Select statement to convert values
- ▶ In an Update statement to set new values
- ▶ In a Insert statement to convert values

A function is a relationship between a set of input values and a set of result values. When invoked, a function performs some operation (for example, concatenate) based on the input and returns a single or multiple results to the invoker. Depending on the nature of the return value or values, user defined functions can be classified into:

- ▶ User Defined (Scalar) Functions with one single return value
- ▶ User defined table functions with a set (=table) of return values

Depending on the way they are coded, there are three different types of UDFs:

- ▶ External user defined functions
- ▶ SQL user defined functions
- ▶ Sourced user defined functions

All types of user defined functions are generated by using the SQL command CREATE FUNCTION.

### 8.3.1 External user defined functions

External UDFs are references to programs and service programs written in high-level languages such as C, C++, ILE CL, COBOL, ILE COBOL, FORTRAN, PLI, RPG, ILE RPG, or JAVA. Once the function is registered to the database, the database will invoke the program or service program whenever the function is referenced in a DML statement. As in SQL UDFs, external UDFs could return a scalar value or a table.

Some of the reasons to work with external UDFs are:

- ▶ To perform non-database functions
- ▶ To access non-relational data
- ▶ To reuse existing code
- ▶ To leverage existing skills

In the following examples we create a service program, CENTER, containing two external functions: CENTER to center a text in a character field, and RIGHTADJ to right adjust a text in a character field.

Example 8-23 shows the prototypes for the functions CENTER and RIGHTADJ. Both functions return an alphanumeric value.

*Example 8-23 Prototypes for the functions CENTER and RIGHTADJ*

---

```

* Reference fields
D Text          S          20A  based(DummyPtr)

* Function CENTER
D Center        PR          like(Text)
D ParText       like(Text) const

* Function RightAdj
D RightAdj      PR          like(Text)
D ParText       like(Text) const

```

---

Example 8-24 shows the source code for the two functions CENTER and RIGHTADJ.

*Example 8-24 Source code for the functions CENTER and RIGHTADJ*

---

```

H NoMain
H Debug BndDir('MYBNDDIR')
*-----
* Prototypes
D/Copy QPROLESRC,CENTER
*****
* Function CENTER
*****
P Center        B          Export
D Center        PI          like(Text)
D ParText       like(Text) const

D LenParText    C          const(%Size(ParText))
D RetText       S          like(Text)
D Start        S          3U 0
*-----
/Free
  Select;
  When ParText = *Blanks;
    Return *Blanks;

```



```

SRCMBR(*SRVPGM)
TEXT('Center / Right adjust textes')
ACTGRP(*CALLER)

/* Registering the service program in the binding directory */
ADDBNDDIRE BNDDIR(MYBNDDIR)
      OBJ((CENTER *SRVPGM))
      POSITION(*LAST)

/* Delete Module */
DLTMOD MODULE(QTEMP/CENTER)

```

---

Example 8-27 shows how these functions are called in an RPG program.

*Example 8-27 Calling the procedures CENTER and RIGHTADJ from RPG*

---

```

* Prototypes
D/Copy QPROLESRC,CENTER

* Field Definition
D TextIn      S              like(Text) inz('MyText')
D TextOut     S              like(Text)
*-----
/Free
  TextOut = Center(TextIn);
  Dsply TextOut;

  TextOut = RightAdj(TextIn);
  Dsply TextOut;

  Return;
/End-Free

```

---

If you want to use these functions in SQL you have to use them as user defined functions.

Example 8-28 shows the registration of the RPG function CENTER and RIGHTADJ as user defined functions.

**Note:** In contrast to programs, service programs can have several entry points, one for each exported procedure. To register user defined functions, the entry point or the function name must be specified. This is even necessary if a service program contains only one function with the same name as the service program, for example, EXTERNAL NAME MySchema.MySrvPgm(MyFunction).

*Example 8-28 Registering the RPG functions CENTER and RIGHTADJ as external UDFs*

---

```

Create Function ITS04710/CenterText
  (ParText CHAR(20) )
  Returns CHAR(20)
  Language RPGLE
  Not Deterministic
  No SQL
  Called on NULL Input
  DisAllow Parallel
  External Name 'ITS04710/CENTER(CENTER)'
  Parameter Style SQL ;

Create Function ITS04710/RightAdjust

```

```

(ParText CHAR(20) )
Returns CHAR(20)
Language RPGLE
Not Deterministic
No SQL
Called On NULL Input
DisAllow Parallel
External Name 'ITS04710/CENTER(RIGHTADJ)'
Parameter Style SQL ;

```

Example 8-29 shows how the external functions can be used with SQL.

*Example 8-29 Using user defined functions CenterText and RightAdjust in SQL*

```

Update MySchema/MyTable
  SET MyChar = CenterText(MyChar)
  MyChar1 = RightAdjust(MyChar1);

```

## Registering an external user defined function using iSeries Navigator

In the following steps we show you how to register these user defined functions using the iSeries Navigator.

Figure 8-11 shows the General tab for creating an external user defined function.

The screenshot shows a dialog box titled "New External Function in ITS04710 - Rchasm27(\$104rt9m)". The "General" tab is selected. The "Function:" field contains "CenterText" and the "Description:" field contains "Center Text - Call RPG function CENTER". Under "Data returned to invoking statement", the "Single value" radio button is selected, and the "Type:" dropdown is set to "CHARACTER". The "Length:" field is set to "20". There are checkboxes for "Can run in parallel", "Program does not call outside of itself (No External Action)", "Same result returned from successive calls with identical input (Deterministic)", and "Attempt to run in same thread as invoking statement (Not Fenced)". The "Data access:" dropdown is set to "No SQL". The "Specific name:" field is empty. At the bottom are "OK", "Cancel", and "Help" buttons.

Figure 8-11 General tab for user defined functions

Figure 8-12 on page 158 shows the Parameters tab for creating a user defined function.

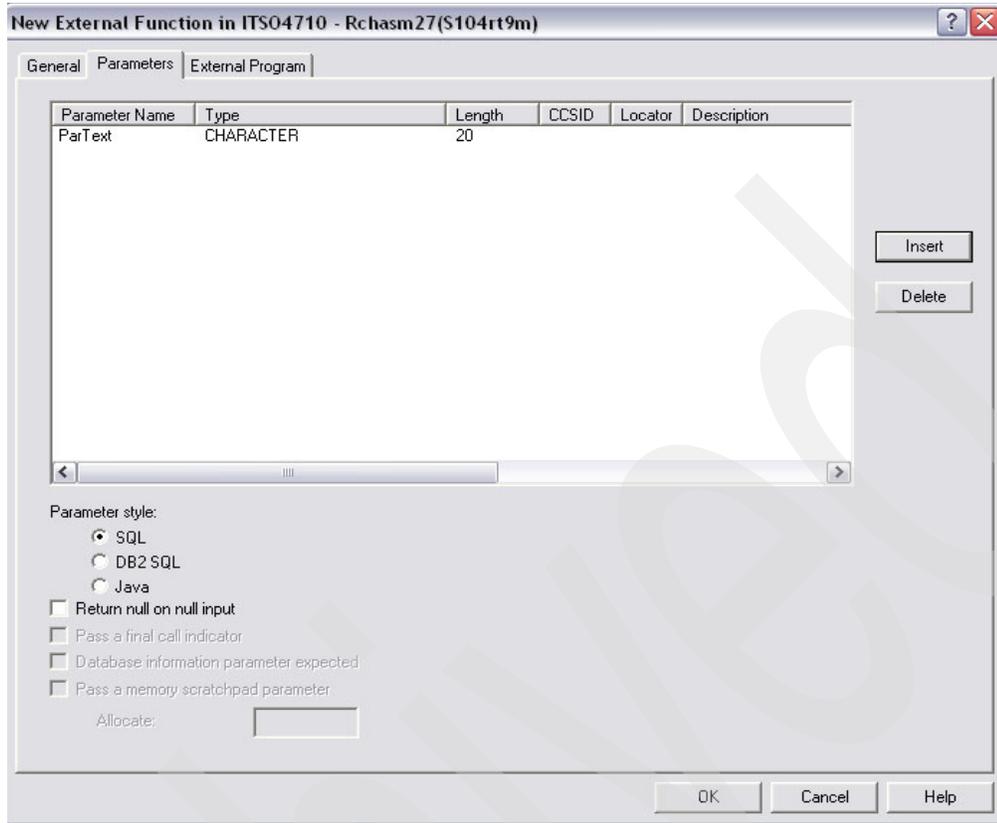


Figure 8-12 Parameter Definition in creating a User defined Functions

Figure 8-8 on page 143 shows the Program tab for external user defined functions.

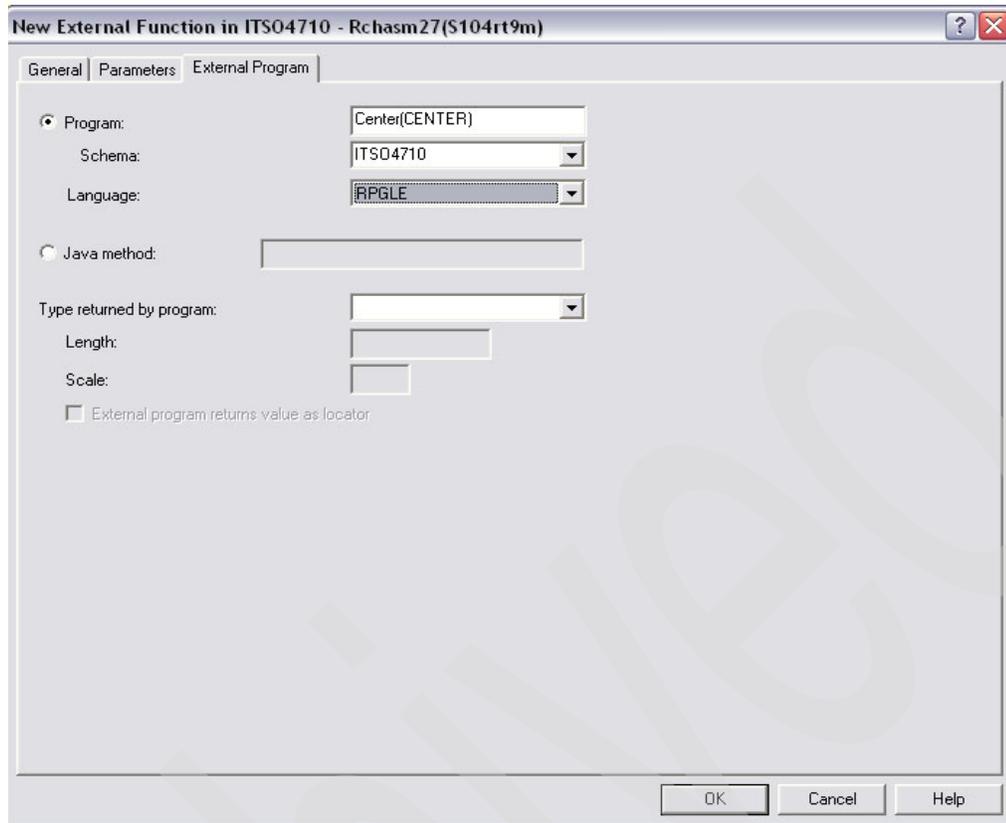


Figure 8-13 Program tab in creating a user defined function

### 8.3.2 SQL user defined scalar functions

SQL UDFs are functions written entirely using procedural SQL language. Their “code” is actually SQL statements embedded within the CREATE FUNCTION statement. SQL UDFs provide several advantages:

- ▶ They are written in SQL, making them portable to other database platforms.
- ▶ Defining the interface between the database and the function is by use of SQL declares, with no need to worry about details of actual parameter passing.
- ▶ They allow the passing of large objects, datalinks, and UDTs as parameters, and subsequent manipulation of them in the function itself.

Example 8-30 shows an SQL user defined function that converts a date into a text, in the format Friday, 10th September 2004.

Example 8-30 SQL User defined scalar function to convert a date into a text string

---

```

Create Function ITS04710/CvtDateToText (MyDate DATE )
  Returns Char(50)
  Language SQL
  Specific ITS04710/CvtDateToText
  Deterministic
  Contains SQL
  Returns NULL on NULL Input
  DisAllow PARALLEL

  Set Option DbgView = *Source,

```

```

        DatFmt = *ISO
Begin
    Return DayName(MyDate) concat ', ' concat
           Trim(Char(DayOfMonth(MyDate))) concat
           Case When DayOfMonth(MyDate) IN (1 , 21 , 31)
                Then 'st'
                When DayOfMonth(MyDate) IN (2 , 22)
                Then 'nd'
                When DayOfMonth(MyDate) = 3
                Then 'rd'
                else 'th'
            end concat ' ' concat
           MonthName(MyDate) concat ' ' concat
           Char(Year(MyDate)) ;
End

```

---

### 8.3.3 User defined table functions

User defined table functions are UDFs that are capable of returning a set of output values. This set of output values is known as a table or result set. User defined table functions return a table instead of a scalar value. Examples of this type of function are:

- ▶ A function that returns the names of sales representatives in a specified region
- ▶ A function that returns all employees whose annual compensation is above the average of the organizational unit to which they belong
- ▶ A function returning the k most profitable customers is a table UDF

**Note:** One very useful and important use of a table function is the ability to access data in non-relational objects with an SQL. A table function can be written to extract data out of a stream file in IFS, and then the invoking SQL statement is able to process that data just like data from an SQL-created table.

For an example of an user defined table function refer to 6.8.1, “User defined table functions for accessing non-relational data” on page 72.

### 8.3.4 User defined function signature and overloading

Like stored procedures, user defined functions can be overloaded, but the signatures are determined in a different way. The signature of a user defined function depends on the procedure name, the number, the sequence, and the data type of the parameters. The length of the parameters is not considered in the signature.

The data type of the value returned by the function is not considered to be part of the function signature.

The following user defined functions can coexist in the same schema:

- ▶ MyProcedure(Int)
- ▶ MyProcedure(SmallInt)

**Note:** Certain data types are considered equivalent when it comes to function signatures.

For example, DECIMAL and NUMERIC or CHAR and GRAPHIC are treated as the same type from the signature point of view. On the other hand, CHAR and VARCHAR are handled as different data types. If you specify an alphanumeric constant, it is treated as VARCHAR.

Distinct types are always treated as different data types, even though they are based on the same data type and length as the defined parameter.

Example 8-31 shows the definition of the data type DateNumISO, which represents a numeric date defined as Decimal(8, 0).

*Example 8-31 Distinct type DateNumISO*

---

```
Create Type ITS04710/DateNumIso
AS Decimal(8, 0) ;
```

---

Example 8-32 shows the definition of the original user defined function CvtNumToDate that converts a numeric date into a date value.

*Example 8-32 UDF CvtNumToDate - Converting numeric date to date definition*

---

```
Create Function ITS04710/CvtNumToDate
(DateNum Decimal(8, 0) )
Returns DATE
Language SQL
Deterministic
Contains SQL
Returns NULL on NULL Input
No External Action
Set Option DbgView = *Source,
DatFmt = *ISO

BEGIN
  Declare CvtDate DATE ;
  Declare InvalidDate Condition For '22007' ;
  Declare Continue HANDLER for InvalidDate
    set CvtDate = '0001-01-01';
  Set CvtDate = Date(Substring(Digits(DateNum), 1, 4) Concat '-' Concat
    Substring(Digits(DateNum), 5, 2) Concat '-' Concat
    Substring(Digits(DateNum), 7, 2) ) ;

  Return CvtDate;
END;
```

---

For numeric fields defined with the data type DateNumISO, you cannot use this function CvtNumToDate. You either have to convert the data type or overload the original function. This can be accomplished by creating a sourced user defined function that converts the data type into decimal and calls the original user defined function.

Example 8-33 shows the sourced user defined function that allows you to use the CvtNumToDate function for the data type DatNumISO.

*Example 8-33 Sourced function to convert numeric dates from DatNumISO to date*

---

```
Create Function ITS04710/CvtNumToDate
(DateNum DateNumIso )
Returns DATE
```

---

If you list the functions in the iSeries Navigator, you will see both functions with the same function name, but different signatures—one SQL defined and the other sourced. Figure 8-14 shows the user defined functions listed in the schema ITS04710.

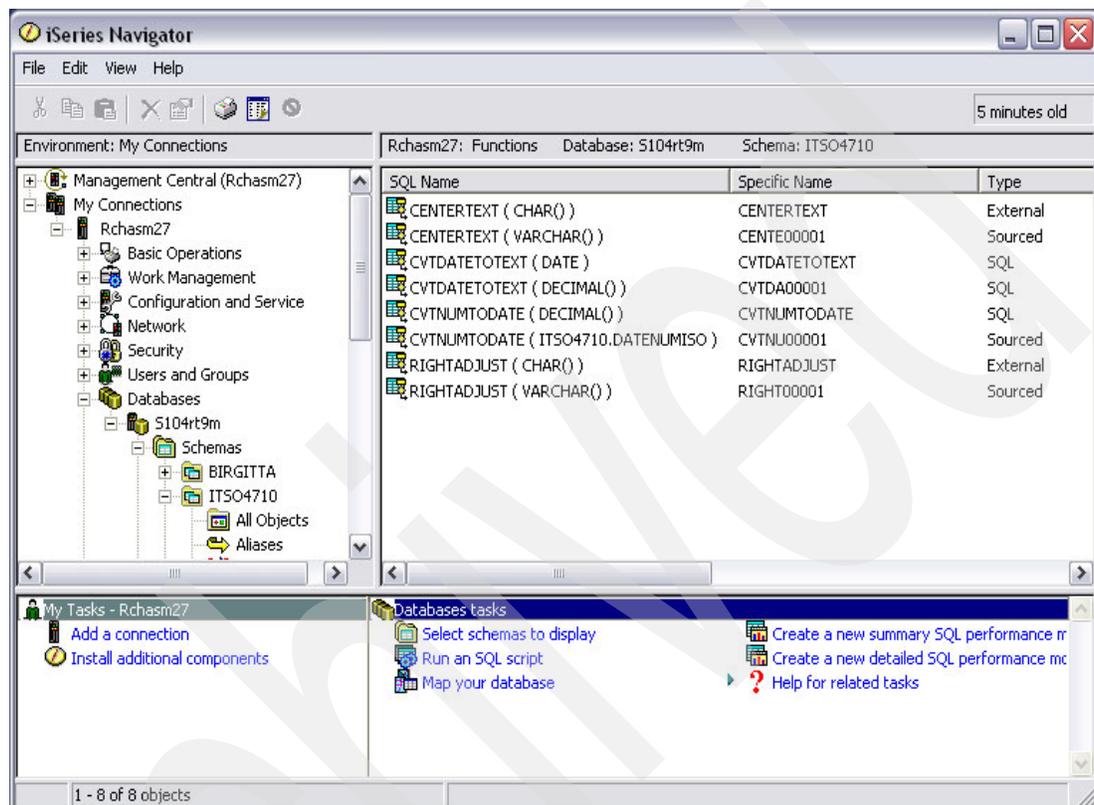


Figure 8-14 User defined functions

### 8.3.5 Deleting or replacing a user defined function

SQL does not provide a REPLACE or CHANGE FUNCTION statement.

If you try to create a user defined function with the same signature as an existing procedure (identical name, identical number of parameters with the same data types), the UDF will not be created. An error occurs:

```
SQL State: 42723
Vendor Code: -454
Message: [SQL0454] Routine CALLUPDEMPLOYEE in ITS04710 already exists.
```

If you create a UDF with the same name as an existing function, but with a different number of parameters or the same number of parameters but with different data types, the user defined function is overloaded, which means a user defined function with a different signature is created.

If you want to replace a user defined function, you first have to delete the old one by using the SQL statement DROP FUNCTION. As long as the FUNCTION is not overloaded, you simply specify DROP FUNCTION and add the FunctionName.

There are two methods to delete overloaded user defined functions:

- ▶ Adding the data types and the length of the parameters of the procedure after the FunctionName in the DROP FUNCTION statement
- ▶ Using the specific name or signature as follows:

```
DROP SPECIFIC FUNCTION SpecificName
```

Example 8-34 shows how user defined functions can be deleted.

*Example 8-34 Deleting user defined functions*

---

```
-- Deleting a UDF that is not overloaded
DROP FUNCTION CvtDateToText;

-- Deleting an overloaded procedure
DROP FUNCTION CvtDateToText(Date);

DROP SPECIFIC FUNCTION CVTDA00002;
```

---

### 8.3.6 Getting information about user defined functions

Information about external and SQL user defined functions are saved in the following system tables:

<b>SYSFUNCS</b>	Provides all information about user defined functions. The program body for SQL user defined functions is saved in the ROUTINE_DEFINITON column.
<b>SYSPARMS</b>	Delivers the description of the parameters, including the order, parameter mode, data type, and length definition.
<b>SYSROUTINES</b>	The SYSROUTINES table contains one row for each procedure created by the CREATE PROCEDURE statement and each function created by the CREATE FUNCTION statement.

For more information on user defined functions refer to the redbook *Stored Procedures, Triggers and User Defined Functions in DB2 UDB for iSeries*, SG24-6503.

## 8.4 SQL programming language

Triggers, stored procedures, and user defined functions can be written by only using SQL as a programming language, called SQL/PSM (Persistent stored modules).

Using SQL/PSM, you can use all SQL statements and scalar functions. It is possible to insert, update, or delete multiple rows in different tables. You can use the SELECT INTO statement to retrieve one single row or value. Furthermore, you can define and handle serial and scroll cursors, like in embedded SQL. Variables can be defined, but in contrast to the host variables used in embedded SQL, they must not be preceded by a colon (:). It is even possible to create and use SQL statements dynamically.

In embedded SQL we learned how to get access to the database data, how to modify them, and how to use the SQL SET statement. But to control the logic program flow we used RPG statements like the operation codes IF or DOU. If we want to move from embedded SQL to SQL/PSM we need those control statements in SQL.

For the use in SQL triggers, SQL stored procedures, and SQL user defined functions, SQL provides a set of control statements that allow SQL to be used in a manner similar to writing a

program in a structured programming language. SQL control statements provide the capability to control the logic flow, declare and set variables, and handle warnings and exceptions. Some SQL control statements include other nested SQL statements.

### 8.4.1 Compound statement

As soon as more than one SQL statement must be executed in a SQL trigger, SQL stored procedure, or SQL User defined Function, they must be embedded in a compound statement.

A compound statement begins with BEGIN and ends with END. The END clause must be ended with a semi colon (;).

Every SQL statement embedded in the compound statement must be ended by a semi colon (;).

When the compound statement is used, there is an order that must be followed:

1. Local variable declarations
2. Local cursor declarations
3. Local handler declarations
4. SQL procedure logic, all other SQL statements

If you want to compare it with RPG, the Definition specifications must be located before the Calculation specifications.

Compound statements can be nested. Nested compound statements can be used to scope handlers, cursors, and variables to a subset of the statements in a procedure. This can simplify the processing done for each SQL procedure statement.

Nested compound statements can be compared with internal procedures in ILE programs. If the same procedure is needed several times, you will transform it in your ILE program into a exported procedure. Before copying the same nested compound statement into several procedures, it would be better to create a stored procedure or a user defined function containing these statements, and call it.

### 8.4.2 Control statements

In the following section we want to give you a short overview over the SQL control statements and compare them with the RPG equivalent.

For more information on the SQL control statements refer to the SQL Reference book.

#### Conditional control

Both RPG and SQL provide two methods for condition handling. In the first one, IF handles a single and even sometimes nested condition, while the SQL CASE statement or the RPG operation code SELECT can handle multiple conditions.

Table 8-2 on page 165 shows the SQL conditional control statements and the RPG equivalent.

Table 8-2 SQL conditional control statements

SQL Syntax	SQL Example	RPG Syntax
IF Condition THEN SQL-Statement; additional SQL-Statements;  ELSEIF Condition THEN SQL-Statement; additional SQL-Statements;  ELSE SQL-Statement; additional SQL-Statements;  END IF;	<pre> IF Month(MyDate) between 1 and 3   THEN Set Quartal = 1;  ELSEIF Month(MyDate) between 4 and 6   THEN Set Quartal = 2;  ELSEIF Month(MyDate) between 7 and 9   THEN Set Quartal = 3;  ELSE Set Quartal = 4;  END IF; </pre>	IF           ELSEIF           ELSE           ENDIF
CASE   WHEN Condition THEN SQL-Statement; additional SQL-Statements;  ELSE SQL-Statement; additional SQL-Statements;  END CASE;	<pre> CASE WHEN Month(MyDate) between 1 and 3   THEN Set Quartal = 1; WHEN Month(MyDate) between 4 and 6   THEN Set Quartal = 2; WHEN Month(MyDate) between 7 and 9   THEN Set Quartal = 3; ELSE Set Quartal = 4; END CASE; </pre>	SELECT           WHEN           OTHER           ENDSL

### Iterative control

Both SQL and RPG provide a number of methods for iterative control, but SQL statements and RPG operation codes differ slightly.

SQL provides four methods for iterative control:

► **LOOP / END LOOP**

The SQL statement LOOP allows you to execute a series of instructions repeatedly.

The SQL LOOP statement can be compared with the DO operation code in RPG.

► **WHILE / END WHILE**

All statements embedded between WHILE and END WHILE are executed as long the specified condition is true. It is important to note that the exit condition is checked in the WHILE condition each time an iteration is going to start. The exit condition must be set in some place of the iteration.

The SQL WHILE statement can be compared with the DOW (Do While) operation code in RPG. It even can be used like the RPG FOR operation code, but the iteration must be done in a separate statement within the WHILE statement.

► **REPEAT / END REPEAT**

All statements embedded between REPEAT and END REPEAT are executed until the specified condition is true. It is important to note that the exit condition is checked at the end of the iteration, while using the WHILE statement the exit condition is checked at the beginning of each iteration.

The SQL REPEAT statement can be compared with the DOU (Do Until) operation code in RPG. In contrast to RPG, where the condition is specified at the beginning of the iteration (within the DOU operation code), in the SQL REPEAT statement the UNTIL condition is defined at the end.

► **FOR / END FOR**

The SQL FOR-Statement combines the DECLARE, OPEN, iterative FETCH, and CLOSE statements for a SERIAL cursor, in one single statement.

There is no equivalent in the RPG programming language.

**Note:** The SQL FOR statement and the RPG operation code FOR cannot be compared.

Table 8-3 shows the SQL iterative control statements and compares them with RPG.

Table 8-3 SQL iterative control statements

SQL Syntax	SQL Example	RPG Syntax
(Label:) LOOP SQL-Statement; additional SQL-Statements;  END LOOP (Label);	NextLoop: LOOP FETCH Cursor into OutPut; IF OutPut = ' ' ; LEAVE; END IF; SET Counter = Counter + 1; END LOOP NextLoop;	DO / ENDDO FOR / ENDFOR
(Label:) WHILE Condition DO SQL-Statement additional SQL-Statements; END WHILE (Label);	WHILE Counter < 10 DO FETCH Csr1 into OutPut; SET Counter = Counter + 1; END WHILE;	DOW / ENDDO
(Label:) REPEAT SQL-Statement; additional SQL-Statements; UNTIL Condition END REPEAT (Label);	REPEAT  FETCH Csr1 into OutPut; SET Counter = Counter + 1; UNTIL SqlState = '02000'; END REPEAT;	DOU / ENDDO
(Label:) FOR Variable as CURSOR FOR SELECT-Statement DO SQL-Statement; additional SQL-Statements;  END FOR (Label);	FOR v1 AS c1 CURSOR FOR SELECT firstme, midinit, lastname FROM employee DO SET fullname = lastname concat ' ' concat firstme concat ' ' concat midinit; INSERT INTO TNAMES VALUE ( fullname ); END FOR;	Combination of SETLL, DO and READ

In some situations it is important to leave a loop or to skip to the next iteration. SQL provides three control statements to achieve this functionality.

- ▶ LEAVE  
LEAVE allows you to leave the iteration. It is an equivalent to the RPG operation code LEAVE.
- ▶ ITERATE  
ITERATE allows you to skip to the next iteration. It is an equivalent to the RPG operation code ITER.
- ▶ GOTO  
GOTO can be used to branch to a label. It is an equivalent to the RPG operation codes GOTO or CABxx.

**Note:** Use the GOTO statement sparingly. This statement interferes with the normal sequence of processing, thus making a routine more difficult to read and maintain. Often, another statement, such as ITERATE or LEAVE, can eliminate the need for a GOTO statement.

Table 8-4 shows the SQL statements that allow you to skip to the next iteration or to leave it.

Table 8-4 Additional SQL iterative statements

SQL Syntax	SQL Example	RPG Syntax
LEAVE	IF EndCond = 1 THEN LEAVE;	LEAVE
ITERATE	ELSEIF NextCond = 1 THEN ITERATE; END IF;	ITER
GOTO Label	IF Cond = Ende THEN GOTO Exit;	GOTO / CAB

### Other SQL control statements

There are some other statements:

► **Assignment statement SET**

Allows you to change the value of a variable. The assignment statement SET can be compared with the RPG operation code EVAL.

► **CALL**

Allows you to call a registered stored procedure or a program. The SQL statement CALL can be compared with the RPG operation codes CALL or CALLP for programs.

► **RETURN**

RETURN is used in user defined functions (UDF) and user defined table functions (UDTF) to return the return values.

Return can be compared with the RPG operation code RETURN where factor 2 is used to return values.

Table 8-5 shows the syntax and examples for these statements.

Table 8-5 Additional control statements

SQL	SQL Example	RPG
SET Variable = Expression;	SET MyVar = Quantity * Price;	EVAL (can be omitted in free format RPG)
CALL Procedure (Parm1, Parm2, ... ParmN);	CALL MyProc (Parm1, Parm2);	CALL + Parm CALLP for programs (can be omitted in free format RPG)
RETURN expression	RETURN Quantity * Price;	RETURN

### 8.4.3 Error handling in SQL

If an error occurs and the error is not handled within the SQL trigger, stored procedure, or user defined function, an escape message is sent to the caller. This is the common handling on the iSeries.

In RPG we have some methods to detect and handle errors:

- ▶ Adding the (E)-Extender to particular operation codes
- ▶ Using a monitor group
- ▶ Defining a \*PSSR
- ▶ Registering ILE Condition handlers

To handle errors in embedded SQL we inquiry for the content of the SQLCODE or SQLSTATE that was returned after executing an SQL statement. In procedural SQL there is an additional method to handle errors, which is the use of Condition handlers.

## Condition handler

A handler declaration associates a handler with an exception or completion condition in a compound statement.

In the text below you will see the DECLARE HANDLER statement:

```
DECLARE Handler Type
        FOR condition
        SQL-Statements
```

A Condition handler is always fired when a condition occurs that matches the condition specified in the handler definition.

Three different handler types can be defined:

- |                 |   |
|-----------------|---|
| <b>CONTINUE</b> | If the handler is invoked successfully, the control is returned to the SQL statement following the one that raised the exception. |
| <b>EXIT</b>     | If the handler is invoked successfully, the control is returned to the caller.  |
| <b>UNDO</b>     | If the handler is invoked successfully, a ROLLBACK is executed and the control is returned to the caller.                         |

There are different conditions that can be specified

- |                      |  |
|----------------------|--|
| <b>NOT FOUND</b>     | Identifies any condition that results in an SQLCODE of +100 or an SQLSTATE beginning with the characters '02'.   |
| <b>SQL EXCEPTION</b> | Identifies any condition that results in a negative SQLCODE.   |
| <b>SQLWARNING</b>    | Identifies any condition that results in a positive SQL return code other than +100. The corresponding SQLSTATE value will begin with the characters '01'. |
| <b>Variable</b>      | It is even possible to declare a variable for a specific SQLSTATE that can be used as condition.   |
| <b>SQLSTATE</b>      | It is even possible to use the value of the SQL state directly in the DECLARE HANDLER statement.   |

The following example shows the definition of several Condition handlers.

*Example 8-35 Declare Condition handler*

```
DECLARE not_found CONDITION FOR '02000';
DECLARE CONTINUE HANDLER
        FOR not_found
        SET at_end = 1;

DECLARE UNDO HANDLER
        FOR SQL EXCEPTION
```

```
SET ErrMsg = 'Error Rollback';  
  
DECLARE EXIT HANDLER  
FOR SqlState value '02000';
```

---

## **SIGNAL and RESIGNAL statement**

Until now we handled system errors, but sometimes we want to create our own exception or escape conditions to end a program or cause a rollback. In RPG we can return a parameter that signals the error or sends an escape message by using the API QMHSNDPM.

The SQL/PSM database language supports two programming constructs that can be used to handle the user-defined errors:

► **SIGNAL**

The SIGNAL statement signals an error or warning condition explicitly. If a handler is defined to handle the exception, it is called immediately by the SIGNAL statement; otherwise the control is returned to the caller.

► **RESIGNAL**

The RESIGNAL statement can only be coded as part of the SQL/PSM Condition handler and is used to re-signal an error or warning condition. It returns SQLSTATE and SQL Message text to the invoker.

For a more complete description of error handling in stored procedures, triggers, and user defined functions, refer to the redbook *Stored Procedures, Triggers and User Defined Functions in DB2 UDB for iSeries*, SG24-6305.

Archived



## Other considerations

In this chapter we focus on additional considerations that we have to take into account when we start programming with SQL. Additionally, we compare RPG coding with SQL coding in areas such as:

- ▶ Differences in the RPG and SQL data types
- ▶ Handling of null values
- ▶ Handling of data and time calculations
- ▶ Handling of different data types

## 9.1 Comparing RPG and SQL data types

To move from native record level access in RPG to static or dynamic SQL it is important to understand the differences in definition between RPG and SQL data types. There are even data types that are supported within RPG where there is no equivalent in SQL and vice versa, so alternate data types must be used.

In this section we address the differences and some workarounds to solve this issue.

Table 9-1 compares the different RPG, DDS, and SQL data types.

Table 9-1 Comparing RPG, DDS, and SQL data types

Description	RPG		DDS		SQL		
	Data Type	Keyword	Data Type	Key word	Data Type	CCSID	
Character fixed length	A	--	A	--	CHAR CHARACTER		
Character varying length	A	varying	A	VARLEN	VARCHAR CHAR VARYING CHARACTER VARYING		
Indicator	N	--	--	--	--		
UCS-2 fixed length	C	--	G	--	GRAPHIC	13488 = UCS - 2 1200 = UTF-16	
UCS-2 varying length	C	varying	G	VARLEN	VARGRAPHIC		
Graphic fixed length	G	--	J E O G	--	GRAPHIC	65535 oder CCSID mit DBCS Encoding Scheme	
Graphic varying Length	G	varying	J E O G	VARLEN	VARGRAPHIC GRAPHIC VARYING		
Binary fixed length	--	SQLTYPE(BINARY: Length)	--	--	BINARY		65535
Binary varying length	--	SQLTYPE(BINARY: Length)	--	--	VARBINARY BINARY VARYING		65535
Character Large Object	--	SQLTYPE(CLOB: Length) SQLTYPE(CLOB_LOCATOR) SQLTYPE(CLOB_FILE)	--	--	CLOB CHAR LARGE OBJECT CHARACTER LARGE OBJECT		
Double Byte Large Object	--	SQLTYPE(DBCLOB: Length) SQLTYPE(DBCLOB_LOCATOR) SQLTYPE(DBCLOB_FILE)	--	--	DBCLOB	CCSID with DBCS Encoding Scheme	

## 9.1.1 Character data types

Character data or strings may contain one or more single-byte or double-byte characters, depending on the format specified.

► Single-byte character set (SBCS) data

Data in which every character is represented by a single byte. Each SBCS data character string has an associated Coded Character Set Identifier (CCSID). If necessary, an SBCS string is converted before it is used in an operation with a character string that has a different CCSID.

► Double-byte character set (DBCS) data

Data in which every character is represented by a character from the double-byte character set (DBCS) that does not include the shift-out or shift-in characters. Every DBCS graphic string has a CCSID that identifies a double-byte coded character set. If necessary, a DBCS graphic string is converted before it is used in an operation with a DBCS graphic string that has a different DBCS CCSID.

### Character data types within RPG

The character data type represents character values and may have any of the following formats:

- A** Character data type
- N** Indicator
- G** Graphic data type
- C** Universal Character Set 2 (USC-2) Data type

Table 9-2 summarizes the different character data-type formats.

Table 9-2 Overview RPG character data types

Data Type	Data Definition	Number of Bytes / Characters	Maximum Length		CCSID
Character	A	one or more single-byte characters	65,535	Byte	
Indicator	N	one single byte character	1	Byte	
Graphic	G	one or more double-byte characters	32,766	Byte	65535 CCSID or DBCS Encoding Scheme
			16,383	Character	
UCS-2	C	one or more double-byte characters	32,766	Byte	13488 = UCS - 2 1200 = UTF-16
			16,383	Character	

#### Character data type

The data type character is used for single-byte character representation.

You define a character field by specifying A in the Data-Type entry of the appropriate specification. You can also define one using the LIKE keyword on the Definition specification where the parameter is a character field.

The default initialization value is blanks.

The length of a character field must be defined between 1 and 65535 bytes.

### ***Indicator***

The indicator format is a special type of character data. Indicators are all one byte long and can only contain the character values '0' (off) and '1' (on).

You define an indicator field by specifying N in the Data-Type entry of the appropriate specification. You can also define an indicator field using the LIKE keyword on the Definition specification where the parameter is an indicator field.

The default value of indicators is '0'.

**Note:** There is no equivalent in SQL. When indicators must be saved in database files, the appropriate column must be defined as CHARACTER with a length of one byte.

### ***Graphic data type***

The graphic format is a character string where each character is represented by 2 bytes. Fields defined as graphic data do not contain shift-out (SO) or shift-in (SI) characters.

The length of a graphic field, in bytes, is two times the number of graphic characters in the field. The fixed-length graphic format is a character string with a set length where each character is represented by 2 bytes.

You define a graphic field by specifying G in the Data-Type entry of the appropriate specification. You can also define one using the LIKE keyword on the Definition specification where the parameter is a graphic field.

The default initialization value for graphic data is X'4040'. The value of \*HIVAL is X'FFFF', and the value of \*LOVAL is X'0000'.

### ***Universal Character Set 2 (USC-2) Data type***

The UCS-2 format is a character string where each character is represented by 2 bytes.

This character set can encode the characters for many written languages. Fields defined as UCS-2 data do not contain shift-out (SO) or shift-in (SI) characters.

The length of a UCS-2 field, in bytes, is two times the number of UCS-2 characters in the field. The fixed-length UCS-2 format is a character string with a set length where each character is represented by 2 bytes.

You define a UCS-2 field by specifying C in the data type entry of the appropriate specification. You can also define one using the LIKE keyword on the Definition specification where the parameter is a UCS-2 field.

The default initialization value for UCS-2 data is X'0020'. The value of \*HIVAL is X'FFFF', \*LOVAL is X'0000', and the value of \*BLANKS is X'0020'.

**Note:** SQL and DDS do not have different data types for graphic and unicode. Double byte characters are always defined with GRAPHIC data type. It depends on the CCSID if Unicode or any other DBCS is used.

### **Character data types within SQL**

Character strings defined with SQL can have one of the following formats:

- ▶ Single byte character strings with fixed and varying length
- ▶ Graphic strings with fixed and varying length

- ▶ Binary strings with fixed and varying length
- ▶ Large objects (LOB)
  - Character large objects (CLOBs)
  - Double byte character large objects (DBCLOBs)
  - Binary large objects (BLOBs)

Table 9-3 summarizes the different SQL character data-type formats.

Table 9-3 Overview over SQL character data types

Description	Data Definition	Number of Bytes / Characters	Maximum Length	CCSID
Character fixed length	CHAR	one or more single-byte characters with fixed length	32,766 Byte	
	CHARACTER			
Character varying length	VARCHAR	one or more single-byte characters with varying length	32,740 Byte	
	CHAR VARYING			
	CHARACTER VARYING			
Graphic	GRAPHIC	one or more double-byte characters with fixed length	32,766 Byte	65535 or CCSID with DBCS Encoding Scheme
			16,383 Character	
Unicode	GRAPHIC	one or more double-byte characters with fixed length	32,766 Byte	13488 = UCS - 2 1200 = UTF-16
			16,383 Character	
Graphic varying length	VARCHAR	one or more single-byte characters with varying length	32,740 Byte	65535 or CCSID with DBCS Encoding Scheme
	GRAPHIC VARYING			
	VARGRAPHIC		16,370 Character	
Unicode varying length	VARCHAR	one or more single-byte characters with varying length	32,740 Byte	13488 = UCS - 2 1200 = UTF-16
	GRAPHIC VARYING			
	VARGRAPHIC		16,370 Character	
Binary	BINARY	one or more bytes	32,766 Byte	65535
Binary varying length	VARBINARY	one or more bytes with varying length	32,740 Byte	65535
	BINARY VARYING			
Character large object	CLOB	single byte characters	2,147,483,647 Byte	
	CHAR LARGE OBJECT		2 Giga byte	
	CHARACTER LARGE OBJECT			
Double byte large object	DBCLOB	double byte characters	2,147,483,647 Byte	CCSID with DBCS Encoding Scheme
			2 Giga byte	
			1,073,741,823 Character	
Binary large object	BLOB	bytes	2,147,483,647 Byte	65535
			2 Giga byte	

### Single byte character strings

A character string is a sequence of bytes where one character is represented by a single byte. Character strings are defined through data type CHAR or CHARACTER and the length

in bytes the string can have. This data type represents the equivalent to the character data type A in RPG.

The maximum length a single byte character string with fixed length can have is 32766 bytes.

**Note:** A RPG character field can be defined up to 65 535 bytes. If you have to store character fields that can contain more than 32766 bytes, you have to define a CLOB in SQL.

### **Graphic strings**

In contrast to RPG, SQL does not use different data types for unicode and other DBCS. If unicode is used, the CCSID 13488 for UCS-2 or 1200 for UTF-16 must be associated.

Graphic strings are defined through data type GRAPHIC, the number of characters the string can have, and the CCSID.

The length attribute for graphic strings with fixed length must be between 1 and 16383 inclusive, which corresponds to 32767 bytes. Contrary to the character strings, the maximum length for graphical strings is identical for RPG and SQL.

**Note:** RPG has two different data types for double-byte characters. The UCS-2 Unicode data type (C) matches with CCSID 13488 and 1200, while all other double-byte characters must be defined with the graphic data type (G).

### **Binary strings**

A binary string is a sequence of bytes. The length of a binary string is the number of bytes in the sequence. A binary string has a CCSID of 65535.

The length attribute must be between 1 and 32766 inclusive.

**Note:** In RPG no data type directly matches binary strings. However, in ILE RPG, a BINARY fixed-length binary-string variable can be declared using the SQLTYPE keyword.

The following example shows how to define a field with BINRARY data type within RPG:

```
D MySqlBinary      S              SqlType(BINARY: 1000)
```

## **9.1.2 Character fields with fixed and varying length**

Character fields can be defined with fixed or varying length.

All values of a fixed-length character-string column have the same length.

The storage allocated for variable-length character fields is 2 bytes longer than the declared maximum length. The left-most 2 bytes are an unsigned integer field containing the current length in characters, graphic characters, or UCS-2 characters. The actual character data starts at the third byte of the variable-length field.

### **Why you should use variable-length fields**

Using variable-length fields can improve the performance of string operations, as well as make your code easier to read since you do not have to save the current length of the field in another variable for SUBSTRING-, or use TRIM functions to ignore the extra blanks. When using TRIM Functions the character field is scanned backwards beginning from the last byte.

This is a disadvantage if the full field length is only occasionally used. Compared to varying fields, the current data can be directly accessed through the length saved in the first two bytes.

### Character fields with varying length in RPG

In RPG varying character fields are defined, adding the keyword VARYING to the field definition in the D-Specifications. The associated length is the maximum length the field can have. The length is measured in single bytes for the character format and in double bytes for the graphic and UCS-2 formats, and must be between 1 and 65535 bytes for single-byte characters and between 1 and 16383 characters for double-byte characters.

Example 9-1 shows how to define character fields with varying length within RPG.

*Example 9-1 Character fields with varying length within RPG*

D MyVarChar	S	65535A	varying
D MyVarGraph	S	16383G	varying
D MyVarUCS2	S	16383C	varying

**Note:** When using the VARYING keyword, the length definition is always required, which means varying fields cannot be referenced through the keyword LIKE.

### Character fields with varying length in SQL

SQL used different data types for fixed and varying length fields.

#### *Varying single byte character strings*

Varying single byte character strings in SQL must be defined with either VARCHAR, CHAR VARYING, or CHARACTER VARYING.

For a VARCHAR column, the length attribute must be between 1 and 32740 inclusive, that is, less than the maximum length fixed character fields can have.

**Note:** While in RPG, varying character fields can be defined with the same maximum length as fixed character fields. The maximum length for SQL-defined varying fields is always shorter as the maximum length of their fix counterparts.

#### *Varying graphic strings*

A graphic string is a sequence of two-byte characters.

Varying graphic strings in SQL must be defined with either VARGRAPHIC or GRAPHIC VARYING.

The length attribute must be between 1 and 16370 inclusive, which is the maximum number of characters the column can hold. The maximum length a graphic field with varying length can have differs from the maximum length a graphic field with fixed length can have.

#### *Varying binary string*

A binary string is a sequence of bytes. The length of a binary string is the number of bytes in the sequence. A binary string has a CCSID of 65535.

Varying binary strings in SQL must be defined with either VARBINARY or BINARY VARYING.

For a VARBINARY column, the length attribute must be between 1 and 32740 inclusive, which is less than the maximum length fixed binary fields can have.

### 9.1.3 Numeric data types

The numeric data type represents numeric values. We can differentiate between three main types with some subtypes:

- ▶ Decimal with a fixed number of decimal positions
  - Zoned numeric data type
  - Packed numeric data type
- ▶ Binary with no decimal positions
  - Binary data type in RPG
  - Integer data type in RPG
  - Unsigned data type in RPG
- ▶ Float with a varying number of decimal positions
  - Float
  - Double

Packed, zoned, and binary formats should be specified for fields when:

- ▶ Using values that have implied decimal positions, such as currency values
- ▶ Manipulating values having more than 19 digits
- ▶ Ensuring a specific number of digits for a field is important

Binary or integer format should be used for fields:

- ▶ When no decimal positions are needed
- ▶ When interacting with other programming languages like C or JAVA that do not have native packed or decimal data types but integer data types and no decimal positions are necessary.

Float format should be specified for fields when:

- ▶ The same variable is needed to hold very small and/or very large values that cannot be represented in packed or zoned values. However, float format should not be used when more than 16 digits of precision are needed.
- ▶ When interacting with other programming languages like C or JAVA that do not have native packed or decimal data types but floating point data types and decimal positions are required.

#### Zoned numeric data type

Zoned-decimal format means that each byte of storage can contain one digit or one character. In the zoned-decimal format, each byte of storage is divided into two portions: A 4-bit zone portion and a 4-bit digit portion.

The zone portion of the low-order byte indicates the sign (positive or negative) of the decimal number. The standard signs are used: Hexadecimal F for positive numbers and hexadecimal D for negative numbers. In zoned-decimal format, each digit in a decimal number includes a zone portion; however, only the low-order zone portion serves as the sign.

A decimal value is a packed or zoned decimal number with an implicit decimal point. The position of the decimal point is determined by the precision (total number of the digits) and the scale (number of digits to the right of the decimal point) of the number. The scale cannot be

negative or greater than the precision. The maximum length a zoned data type can have is 63 digits, which applies to both SQL and RPG.

All values of a decimal column have the same precision and scale.

**Note:** The total number of digits and decimal positions decimal numbers can have are identical in both RPG and SQL. A zoned field can have up to 63 digits.

### ***Zoned numeric fields in RPG***

To define zoned numeric fields in RPG data type S must be specified and the length and the number of the decimal positions must be added.

**Note:** Keep in mind that RPG translates the numeric data types as far as possible to packed numeric data types. If you really want to work with the zoned data type in RPG and not with converted packed data types, you may embed your zoned numeric field into either an internal or external data structure.

### ***Zoned numeric data in SQL***

To define zoned numeric data in SQL, the data type NUMERIC must be specified along with the precision and the scale of the column.

### **Packed numeric data type**

Packed-decimal format means that each byte of storage (except for the low-order byte) can contain two decimal numbers. The low-order byte contains one digit in the left-most portion and the sign (positive or negative) in the right-most portion. The standard signs are used: Hexadecimal F for positive numbers and hexadecimal D for negative numbers.

**Note:** The total number of digits and decimal positions that decimal numbers can have are identical in both RPG and SQL. The maximum number of digits a packed numeric field can have is 63.

### ***Packed numeric fields in RPG***

To define packed numeric fields in RPG, the precision and the number of the decimal positions must be specified, while the data type P is optional.

**Note:** As already pointed out, RPG converts, as far as possible, numeric data types to packed. But there is at least one exception when packed data are converted to zoned data.

If you list packed numeric fields that are defined through file description (F-Specification) without any length or data type, numeric fields are handled as zoned fields.

### ***Packed numeric data in SQL***

To define packed numeric data in SQL, the data type DEC or DECIMAL must be specified along with the precision and the scale of the column.

### **Binary data types**

The binary representation is the most compact representation of a numeric value. In one byte up to 256 ( $2^8$ ) different values can be stored, in comparison with zoned representation in which only 10 different values are used per byte, and with packed representation 100 values.

The maximum value a binary field can have depends on the number of used bytes. Binary data can be saved in 1, 2, 4, and 8 bytes.

In RPG three different binary data types can be defined:

**B** Binary  
**I** Integer  
**U** Unsigned

In SQL only the integer data type is available. Depending on the number of bytes, the values are stored in different data types:

**Small Integer** For 2 bytes  
**Integer/large integer** For 4 bytes  
**Big Integer** For 8 bytes

Table 9-4 compares the different binary data types in RPG and SQL.

Table 9-4 Overview binary data types in RPG and SQL

Number of Bytes	Integer - Definition		
	RPG	Range	SQL
1	3I 0	-128 - 128	--
2	5I 0	-32,768 - 32,767	Small Integer SMALLINT
4	10I 0	-2,147,483,648 - 2,147,483,647	Integer / Large Integer INTEGER INT
8	20I 0	-9,223,372,036,854,775,808 - 9,223,372,036,854,775,807	Big Integer BIGINT
Number of Bytes	Unsigned - Definition		
	RPG	Range	SQL
1	3U 0	0 - 255	Integer - Definition
2	5U 0	0 - 65,565	
4	10U 0	0 - 4,294,967,295	
8	20U 0	0 - 18,446,744,073,709,551,615	
Number of Bytes	Binary Definition		
	RPG	Range	SQL
2	4B 0	-9,999 - 9,999	Integer Definition
4	9B 0	-999,999,999 - 999,999,999	

**Note:** Specify integer for fields when no decimal positions are needed. Binary representation is the most compact representation of numeric values.

### Binary data type in RPG

Binary format means that the sign (positive or negative) is in the left-most bit of the field and the numeric value is in the remaining bits of the field. Positive numbers have a zero in the sign bit; negative numbers have a one in the sign bit.

To define binary fields the data type B must be specified and the precision and number of decimal positions added.

A binary field can be from one to nine digits in length and can be defined with decimal positions. If the length of the field is from one to four digits, the compiler assumes a binary field length of 2 bytes. If the length of the field is from five to nine digits, the compiler assumes a binary field length of 4 bytes. Because of these length restrictions, the highest decimal

value that can be assigned to a 2-byte binary field is 9 999, and the highest decimal value that can be assigned to a 4-byte binary field is 999 999 999. In comparison, in a two-byte binary field up to  $2^{16}$  values can be saved that correspond to a range from -32 768 to +32 768. In a 4-byte binary field up to  $2^{32}$  can be saved that correspond to a range from -2 147 483 648 to +2 147 483 647.

In RPG the binary fields are converted to packed numeric fields with the adequate number of digits and decimal positions, and they are under the restrictions (that is, overflow) of the packed fields. In general, a binary field of n digits can have a maximum value of n 9s. This discussion assumes zero decimal positions.

**Note:** The binary data type should only be used when decimal positions for binary fields are needed. Otherwise, integer data types should be preferred, because fields defined with RPG's binary data type cannot hold the complete range a binary field can have. Integer fields are not converted by the RPG compiler to packed numeric fields.

If binary fields are used as data structure sub fields they either have to be defined through their length or using the from/to length specification.

Example 9-2 shows different ways to define binary fields as data structure sub fields within RPG.

*Example 9-2 Defining binary fields as data structure subfields in RPG*

D MyFirstDS	DS		
D FirstBin2			4B 0
D FirstBin4			9B 0
D MySecondDS	DS		
D SecondBin2		1	2B 0
D SecondBin4		3	6B 0

**Note:** There will be no data type in SQL that directly matches with RPG's binary data type. If RPG binary data have to be saved in SQL columns, they have to be defined with either small integer or integer data type, depending on the number of bytes.

### **Integer data type in RPG**

The integer format is similar to the binary format with two exceptions:

- ▶ The integer format allows the full range of binary values.
- ▶ The number of decimal positions for an integer field is always zero.

You define an integer field by specifying I in the Data-Type entry of the appropriate specification. Decimal position must always be inserted with zero. You can also define an integer field using the LIKE keyword on a Definition specification where the parameter is an integer field.

**Note:** In contrast to the binary fields, integer fields are not converted into packed fields in RPG. Because of their compact representation it is the fastest way to access numeric data without decimal positions.

The length of an integer field is defined in terms of number of digits; it can be 3, 5, 10, or 20 digits long. A 3-digit field takes up 1 byte of storage; a 5-digit field takes up 2 bytes of storage; a 10-digit field takes up 4 bytes; a 20-digit field takes up 8 bytes. The range of values allowed for an integer field depends on its length.

Table 9-5 shows the definitions of the different integer data types and their valid ranges.

Table 9-5 Integer data types within RPG

Number of bytes	Field definition	Range of allowed values
1	3I 0	-128 - 128
2	5I 0	-32,768 - 32,767
4	10I 0	-2,147,483,648 - 2,147,483,647
8	20I 0	-9,223,372,036,854,775,808 - 9,223,372,036,854,775,807

**Note:** In contrast to RPG, there are different SQL data types to store two, four, and eight byte integer values:

- ▶ The equivalent for the two-byte integer data type is small integer (SMALLINT).
- ▶ The equivalent for the four-byte integer is integer (INT or INTEGER).
- ▶ The equivalent for the eight-byte integer is big integer (BIGINT).

In SQL, there is no data type that matches directly to the one byte integer data type. A small integer definition must be used instead.

If integer fields are used as data structure sub fields they either have to be defined through their length or when using the from/to length specification.

Example 9-3 shows how to define integer fields as data structure sub fields.

Example 9-3 Defining integer fields as data structure subfields in RPG

```

D MyFirstDS      DS
D  FirstInt1    3I 0
D  FirstInt2    5I 0
D  FirstInt4   10I 0
D  FirstInt8   20I 0

D MySecondDS    DS
D  SecondInt1   1  1I 0
D  SecondInt2   3  4I 0
D  SecondInt4   5  8I 0
D  SecondInt8   9 16I 0

```

### Integer data type in SQL

Depending on the number of bytes the values are stored in, SQL differs between three data types:

- 2-byte integer:** Small integer that is adequate with the data type SMALLINT
- 4-byte integer:** Integer or large integer that is adequate with the data type INTEGER or INT
- 8-byte integer:** Big integer that is adequate with the data type BIGINT

The length and the scale of the SQL integer fields are defined through the specified data type.

**Note:** There is no data type to define one-byte binary fields. Small integer must be used instead.

Table 9-6 on page 183 displays the list of different integer data types and their allowed data ranges.

Table 9-6 Overview SQL integer data types

Number of bytes	Description	Data type	Range of allowed values	
2	Small Integer	SMALLINT	-32,768	32,767
4	Large Integer / Integer	INTEGER	-2,147,483,648	2,147,483,647
		INT		
8	Big Integer	BIGINT	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

**Note:** In contrast to RPG, it is not possible to define unsigned integer fields within SQL. When unsigned field values must be stored in SQL columns, integer or decimal data types must be used. The maximum value of an unsigned field is twice as high as the maximum value of the integer field. So it may be necessary to change to the larger integer definition or use a decimal data type.

### Unsigned data type in RPG

The unsigned integer format is like the integer format except that the range of values does not include negative numbers. You should use the unsigned format only when non-negative integer data is expected.

You define an unsigned field by specifying U in the Data-Type entry of the appropriate specification. The decimal positions must always be inserted with zero. You can also define an unsigned field using the LIKE keyword on the Definition specification where the parameter is an unsigned field.

The length of an unsigned field is defined in terms of number of digits; it can be 3, 5, 10, or 20 digits long. A 3-digit field takes up 1 byte of storage; a 5-digit field takes up 2 bytes of storage; a 10-digit field takes up 4 bytes; a 20-digit field takes up 8 bytes. The range of values allowed for an unsigned field depends on its length.

Because there are no negative values allowed and the valid range begins with zero, the maximum value is twice as high as the maximum value of the integer field. This must be considered when handling with integer, unsigned and decimal fields.

Table 9-7 shows the definition of the different integer data types and their valid ranges.

Table 9-7 Unsigned data types within RPG

Number of bytes	Field definition	Range of allowed values	
1	3U 0	0	255
2	5U 0	0	65,565
4	10U 0	0	4,294,967,295
8	20U 0	0	18,446,744,073,709,551,615

**Note:** In SQL, there is no equivalent for unsigned data types. Alternatively, integer or decimal data types must be defined. You have watch the maximum values that integer and unsigned fields can hold. It may be necessary to switch to the larger integer definition. If you have to save the maximum value of an 8-byte unsigned field within SQL columns, you have to define a decimal field with the appropriate number of digits.

### Floating point numeric data type

The float format consists of two parts:

- ▶ The mantissa
- ▶ The exponent

The value of a floating-point field is the result of multiplying the mantissa by 10 raised to the power of the exponent. For example, if 1.2345 is the mantissa and 5 is the exponent, then the value of the floating-point field is:

$$1.2345 * (10 ** 5) = 123450$$

Floating point numbers can be stored in either 4 (single-precision floating point) or 8 (double-precision floating point) bytes. The range of magnitude for single-byte precision is approximately  $1.17549436 \times 10^{-38}$  to  $3.40282356 \times 10^{38}$ , while the range for double-precision floating point is approximately  $2.2250738585072014 \times 10^{-308}$  to  $1.7976931348623158 \times 10^{308}$ .

**Note:** Float variables conform to the IEEE standard as supported by the OS/400 operating system. Since float variables are intended to represent scientific values, a numeric value stored in a float variable may not represent the exact same value as it would in a packed variable. Float should not be used when you need to represent numbers exactly to a specific number of decimal places, such as monetary amounts.

Table 9-8 shows the different floating point definitions in RPG compared with SQL and their valid ranges.

Table 9-8 Comparing SQL and RPG floating point data types

Number of Bytes	RPG	SQL	Length	Valid data range
4	4F	REAL	--	$1.17549436 \times 10^{-38}$ - $3.40282356 \times 10^{38}$
		FLOAT(Integer)	1 - 24	
8	8F	FLOAT	--	$2.2250738585072014 \times 10^{-308}$ - $1.7976931348623158 \times 10^{308}$
		FLOAT(Integer)	25 - 53	
		DOUBLE	--	

Float format should be specified for fields when the same variable is needed to hold very small and/or very large values that cannot be represented in packed or zoned values. However, float format should not be used when more than 16 digits of precision are needed.

### Floating point data types in RPG

You define a floating-point field by specifying F in the data type entry of the appropriate specification.

The length of a floating point field is defined in terms of the number of bytes. It must be specified as either 4 or 8 bytes.

The decimal positions must be left blank. However, floating-point fields are considered to have decimal positions. As a result, float variables may not be used in any place where a numeric value without decimal places is required, such as an array index, do loop index, etc.

**Note:** In SQL, there are different data formats for four-byte and eight-byte floating point data types:

- ▶ The equivalent for the 4-byte floating point data type is REAL or FLOAT with a length between 1 and 24.
- ▶ The equivalent for the 8-byte floating point data type is DOUBLE or FLOAT with a length between 25 and 53 or FLOAT.

### ***Floating point data types in SQL***

There are three different data types that describe floating point data types:

- ▶ REAL without any length or scale specification for a single precision floating point.
- ▶ FLOAT with an length between 1 and 24 for a single-precision floating point. FLOAT with an length between 25 and 53 for an double-precision floating point. When FLOAT data type is used without any length specification, the default value 53 is used.
- ▶ DOUBLE without any length specification for double-byte precision.

## **9.1.4 Date and time data types**

There are three different date and time data types:

- ▶ Date data type
- ▶ Time data type
- ▶ Timestamp data type

### **Date data type**

A date is a three-part value (year, month, and day) designating a point in time under the Gregorian calendar, which is assumed to have been in effect from the year 1 A.D.

**Note:** The Gregorian calendar was first implemented October 15th, 1582 by pope Gregor XIII. The days between October 4th and 15th, 1582 were eliminated, while the week day counting was not changed (even though the Gregorian calendar was not introduced in all European countries at the same time—in Germany in 1700 and USA and Great Britain in 1752).

The Lillian Date is the number of days from October 15th, 1582 until the specified date. October 15th is day 1.

The internal representation of a date is a string of 4 bytes that contains an integer. The integer (called the Scaliger number) represents the date.

Depending on the associated date format, the valid date range differs.

- ▶ Date formats with a two digits year, YMD, \*DMY, \*MDY, and \*JUL, have a valid year range from year 1940 to 2039.
- ▶ Date formats with a four-digit year, ISO, \*USA, \*EUR, \*JIS, and \*LONGJUL, have a valid year range from year 0001 to 9999.
- ▶ Dates formats with a three-digit year, that are \*CYMD, \*CMDY, and \*CDMY, have a valid year range from year 1900 to 2899.

**Note:** Dates with two-digit and four-digit date formats, with the exception of \*LONGJUL, can be defined in RPG. When defining SQL Date columns, no date format is explicitly associated, but a date format is required to represent a date. To fix the date format, there are several possibilities:

- ▶ In the CL command STRSQL are options DATFMT, DATSEP, TIMFMT, and TIMSEP to set the date and time formats.
- ▶ In iSeries Navigator → Run SQL Scripts → Connection → JDBC Setup → Format date and time formats can be set.
- ▶ The compile commands CRTSQLxxxI have options (DATFMT, DATSEP, TIMEFMT, TIMESEP) to set the date and time format.
- ▶ Additionally, there is an SQL statement SET OPTION that allows you to set the date and time formats. This statement can be used either in embedded SQL or SQL Programming Language.

Dates with two-digit and four-digit years can be defined within RPG, and are therefore named as internal date formats. Dates with three-digit years cannot be defined, but are handled within RPG, and are therefore named as external date formats. Defining dates with a three-digit year is only available in DDS using the keyword DATFMT. These three-digit year date formats are not valid for the date (L) type field. They are only valid on logical file zoned, packed, or character types having a physical file based on date type fields

Table 9-9 gives an overview of the internal and external date data types, their valid separators, and valid data ranges.

Table 9-9 Internal and external date formats

Date Format	Description	Layout with default separators	Separators	Length	Example	Valid Range
<b>Internal date formats</b>						
<b>2-digit year formats</b>						
*MDY	Month/Day/Year	MM/DD/YY	/-., '&'	8	11/24/04	01.01.1940 - 31.12.2039
*DMY	Day/Month/Year	DD/MM/YY	/-., '&'	8	24/11/04	01.01.1940 - 31.12.2039
*YMD	Year/Month/Day	YY/MM/DD	/-., '&'	8	04/11/24	01.01.1940 - 31.12.2039
*JUL	Julian Date	YYDDD	/-., '&'	6	04/125	01.01.1940 - 31.12.2039
<b>4-digit year formats</b>						
*ISO	International Standard Organisation	YYYY-MM-DD	-	10	2004-11-24	01.01.0001 - 31.12.9999
*EUR	European Standard	DD.MM.YYYY	.	10	24.11.2004	01.01.0001 - 31.12.9999
*USA	USA Standard	MM/DD/YYYY	/	10	11/24/2004	01.01.0001 - 31.12.9999
*JIS	Japanese Industry Standard	YYYY-MM-DD	-	10	2004-11-24	01.01.0001 - 31.12.9999
<b>External date formats</b>						
<b>2-digit year formats</b>						
*JobRun	Determined at run time					
<b>2-digit year formats</b>						
*CYMD	Cent./Year/Month/Day	CMM/DD/YY	/-., '&'	9	104/11/24	01.01.1900 - 31.12.2899
*CMDY	Cent./Month/Day/Year	CDD/MM/YY	/-., '&'	9	111/24/04	01.01.1900 - 31.12.2899
*CDMY	Cent./Day/Month/Year	CYY/MM/DD	/-., '&'	9	124/11/04	01.01.1900 - 31.12.2899
<b>4-digit year formats</b>						
*LongJUL	Long Julian Date	CYYYYDDD	/-., '&'	8	2004/329	01.01.1900 - 31.12.2899

### **Date data type in RPG**

You define a date field by specifying D in the data type entry of the appropriate specification. It is not necessary to input a length, because it is predetermined through the data type and the date format.

The default internal format for date variables is \*ISO. This default internal format can be overridden globally by the control specification keyword DATFMT and individually by the Definition specification keyword DATFMT.

The hierarchy used when determining the internal date format and separator for a date field is:

1. From the DATFMT keyword specified on the Definition specification
2. From the DATFMT keyword specified on the control specification
3. \*ISO

The Date separators can be set by adding them to the date format keyword either in the definition or in the control specifications. The date separator can only be set for dates with a two-byte year portion.

Example 9-4 shows how to define date fields with different date formats within RPG.

*Example 9-4 Defining date fields with different date formats in RPG*

D MyDateIso	S	D	
D MyDateEur	S	D	DatFmt(*Eur)
D MyDateYMD	S	D	DatFmt(*YMD-)

Date values are stored as Scaliger numbers, and the date format provides only a method to present the date in a readable manner. Date constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators.

**Note:** Date formats provide only a method to represent the internal encrypted 4-byte integer date value in a readable manner. They do not convert the internal value in any case.

### **Date data type in SQL**

Date columns are defined through the data type DATE. This data type is associated with the internal data format. Like in RPG, a date format is necessary to convert the internal date value into a readable date.

In SQL the date format can be set as follows:

- ▶ In the CL command STRSQL are options (DATFMT, DATSEP, TIMEFMT, TIMESEP) to set the date and time formats.
- ▶ In iSeries Navigator → Run SQL Scripts → Connection → JDBC Setup → Format date and time formats can be set.
- ▶ The compile commands CRTSQLxxI have options (DATFMT, DATSEP, TIMEFMT, TIMESEP) to set the date and time format.
- ▶ Additionally, there is an SQL statement SET OPTION that allows you to set the date and time formats. This statement can be used either in embedded SQL or SQL Programming Language.

**Note:** Contrary to RPG, a date value can be inserted or changed in SQL, even if it is out of the valid range of the current date format. Problems with date and time formats and values within embedded SQL are caused by the RPG restrictions.

SQL and RPG should use identical date formats, or at least a format with identical year ranges.

## Time data type

A time is a three-part value (hour, minute, and second) designating a time of day using a 24-hour clock. The range of the hour part is 0 to 24, while the range of the minute and second parts is 0 to 59. If the hour is 24, the minute and second specifications are both zero.

The internal representation of a time is a string of 3 bytes. Each byte consists of two packed decimal digits. The first byte represents the hour, the second byte the minute, and the last byte the second.

Table 9-10 shows an overview of the time data types, their valid separators, and valid data ranges.

Table 9-10 Overview of time formats

Time Format	Description	Representation with default separator	Separator	Length	Example	Valid Range
*HMS	Hour:Minute:Second	HH:MM:SS	.,&	8	14:12:25	00:00:00 - 24:00:00
*ISO	International Standard Organisation	HH.MM.SS	.	8	14.12.25	00.00.00 - 24.00.00
*EUR	European Standard	HH.MM.SS	.	8	14.12.25	00.00.00 - 24.00.00
*USA	USA Standard	HH.MM am / HH.MM PM	.	8	02.12 PM	00:00 AM - 12:00 PM
*JIS	Japanese Industrie Standard	HH:MM:SS	:	8	14:12:25	00:00:00 - 24:00:00

### Time data type in RPG

You define a time field by specifying T in the data type entry of the appropriate specification. It is not necessary to input a length, because it is predetermined through the data type and the time format.

The default internal format for time variables is \*ISO. This default internal format can be overridden globally by the control specification keyword TIMFMT and individually by the Definition specification keyword TIMFMT.

The hierarchy used when determining the internal time format and separator for a time field is:

1. From the TIMFMT keyword specified on the Definition specification
2. From the TIMFMT keyword specified on the control specification
3. \*ISO

The time separators can be set by adding them to the TIMFMT keyword either in the definition or in the control specifications. The time separator can only be set for the \*HMS time format.

Time constants or variables used in comparisons or assignments do not have to be in the same format or use the same separators. Also, times used for I/O operations such as input

fields, output fields, or key fields are also converted (if required) to the necessary format for the operation.

**Note:** When using USA Standard time data type in RPG, the seconds portion is overwritten by AM or PM and the seconds get lost. If you need to calculate with seconds you should pick any other time format.

### **Time data type in SQL**

Time columns are defined through the data type TIME. This data type is associated with the internal data format. To present the time data a time format is necessary.

In SQL the time format can be set as follows:

- ▶ In the CL command STRSQL are options (DATFMT, DATSEP, TIMFMT, TIMSEP) to set the date and time formats.
- ▶ In iSeries Navigator → Run SQL Scripts → Connection → JDBC Setup → Format date and time formats can be set.
- ▶ The compile commands CRTSQLxxI have options (DATFMT, DATSEP, TIMEFMT, TIMESEP) to set the date and time format.
- ▶ Additionally, there is an SQL statement SET OPTION that allows you to set the date and time formats. This statement can be used either in embedded SQL or SQL Programming Language.

**Note:** Contrary to RPG, when using USA standard time format, the seconds are saved. When defining time fields in RPG with the USA standard time format, the seconds will get lost.

### **Timestamp data type**

A timestamp is a seven-part value (year, month, day, hour, minute, second, and microsecond) that designates a date and time as defined previously, except that the time includes a fractional specification of microseconds.

The internal representation of a timestamp is a string of 10 bytes. The first 4 bytes represent the date, the next 3 bytes the time, and the last 3 bytes the microseconds (the last 3 bytes contain six packed digits).

Until now there was only one timestamp format available. The year is always saved as a four-digit year.

Table 9-11 shows the timestamp format.

*Table 9-11 Timestamp format*

Format	Description	Layout	Length	Example
*ISO	International Standard Organisation	YYYY-MM-DD-HH.MM.SS.MICROS	26	2004-11-24-15.19.27.123000

### **Timestamp data type in RPG**

You define a timestamp field by specifying Z in the data type entry of the appropriate specification. You do not fill in a field length because it is predetermined through the data type.

**Note:** In RPG and through RPG Operations codes and built-in functions only the left-most three digits of the microsecond portion are used, while SQL special register CURRENT TIMESTAMP and scalar functions use all of these 6 digits.

## 9.2 NULL values

A NULL is an attribute that a column can have to indicate a missing or unknown value. All data types can be assigned a NULL value. Although all data types can be assigned a null value, there are some considerations to keep in mind. For example:

- ▶ Constants cannot contain NULL values.
- ▶ Columns that are defined as NOT NULL (WITH DEFAULT) cannot contain NULL values.
- ▶ Special registers cannot contain NULL values.
- ▶ The COUNT and COUNT\_BIG functions cannot return a NULL value.
- ▶ ROWID columns cannot store a null value, although a null value can be returned for a ROWID column as the result of a query.

In traditional DDS-described files NULL values are not used. Besides, there is a keyword, ALWNULL, that allows creating files with fields that can contain NULL values. With the begin of the data transfer between iSeries and PCs and using SQL-based tables, where the use of NULL values instead of default values is standard, NULL values must be handled.

There are two methods to allow NULL values handling in RPG:

- ▶ ALWNULL option in compile commands

The compile commands CRTBNDRPG and CRTRPGMOD and even CRTRPGPGM (for RPG/400) have an option, ALWNULL, to determine if NULL values from externally described files can be handled.

The default value for the ALWNULL option is \*NO, which means NULL values from externally described files are not handled.

The compile command CRTSQLRPGI does not provide this option. When using native I/O with tables that can contain NULL values and embedded SQL, the ALWNULL keyword in the Control Specifications must be specified.

- ▶ Keyword ALWNULL in the Control Specifications

Using the keyword ALWNULL in the control specification has some advantages over using the compile options. First, you do not have to remember the option you set when you have to recreate your program or module. When using embedded SQL and native I/O, the ALWNULL keyword must be used, because the compile command CRTSQLRPGI does not provide an ALWNULL option.

If the ALWNULL keyword is not entered in the Control Specifications, the compile option ALWNULL is used.

Both the ALWNULL option and the keyword ALWNULL can have the same characteristics.

- ▶ \*NO, which is the default value

Specifies that the ILE RPG program will not process records with null-value fields from externally described files. If you attempt to retrieve a record containing null values, no data in the record is accessible to the ILE RPG program and a data-mapping error occurs.

- ▶ \*INPUTONLY or \*YES

Specifies that the ILE RPG program can successfully read records with null-capable fields containing null values from externally described input-only database files. When a record containing null values is retrieved, no data mapping errors occur and the database default values are placed into any fields that contain null values. The program cannot do any of the following:

- Use null-capable key fields.
- Create or update records containing null-capable fields.
- Determine whether a null-capable field is actually null while the program is running.
- Set a null-capable field to be null.

► \*USRCTL

Specifies that the ILE RPG program can read, write, and update records with null values from externally described database files. Records with null keys can be retrieved using keyed operations. The program can determine whether a null-capable field is actually NULL, and it can set a null-capable field to be NULL for output or update. The programmer is responsible for ensuring that fields containing null values are used correctly within the program.

**Note:** Both the compiler option ALWNULL and the keyword ALWNULL only affect externally described files that are defined in the File specifications or that are used as externally described data structures.

## 9.2.1 Handling NULL values in RPG with native I/O

When using RPG native I/O, NULL values can be detected or set by using the built-in function %NULLIND(FieldName). The built-in function only can be used in combination with an external file description. All fields that are defined in DDS with the keyword ALWNULL are NULL capable.

If the file used for an externally described data structure has null-capable fields defined, the matching RPG subfields are defined to be null-capable. Similarly, if a record format has null-capable fields, a data structure defined with LIKERECD will have null-capable subfields. When a data structure has null-capable subfields, another data structure defined like that data structure using LIKEDS will also have null-capable subfields. However, using the LIKE keyword to define one field like another null-capable field does not cause the new field to be null-capable.

The %NULLIND built-in function can be used to query or set the null indicator for null-capable fields. This built-in function can only be used if the ALWNULL(\*USRCTL) keyword is specified on a control specification or if the compiler option ALWNULL is set to \*USRCTL. The field name can be a null-capable array element, data structure, stand-alone field, subfield, or multiple occurrence data structure.

%NULLIND can only be used in expressions in extended factor 2 or in free format coding.

When used on the right-hand side of an expression, this function returns the setting of the null indicator for the null-capable field. The setting can be \*ON or \*OFF. When used on the left-hand side of an expression, this function can be used to set the null indicator for null-capable fields to \*ON or \*OFF. The content of a null-capable field remains unchanged. The content of a field can only be changed if the NULL indicator is set to \*OFF.

Example 9-5 on page 192 shows the DDS description for the file OrdHead, where all the fields with the exception of OrdHNbr are NULL-capable fields.

*Example 9-5 Defining a physical file with fields that can contain NULL values*

```

A          UNIQUE
A          R ORDHEADF
*
A          ORHNBR          5          COLHDG('ORDER NUMBER  ')
A          CUSNBR          5          COLHDG('CUSTOMER NUMBER ')
A          ALWNULL
A          ORHDTE          L          COLHDG('ORDER DATE    ')
A          ALWNULL
A          ORHDLY          L          COLHDG('ORDER DELIVERY ')
A          ALWNULL
A          SRNBR           10         COLHDG('ORDER SALESREP ')
A          ALWNULL
A          ORHTOT          11P 2     COLHDG('ORDER TOTAL   ')
A          ALWNULL

A          K ORHNBR

```

Example 9-6 shows an example of how to handle NULL values in RPG. In the example:

- ▶ If the delivery date contains NULL values and order total contains neither NULL values nor zeros, delivery date is set to current date.
- ▶ If the delivery date contains NULL values and order total contains zeros, order total is set to NULL.
- ▶ If delivery date is not NULL and order total contains either NULL values or zeros, delivery date and order total are set to NULL.

*Example 9-6 Handling NULL values in RPG*

```

H Option(*NoDebugIO) Debug
H AlwNull(*UsrCtl)
*-----
FOrdHead  UF  E          K DISK
*-----
/Free
  SetLL *Start OrdHead;
  DoU %EOF(OrdHead);
  Read OrdHeadF;

  If %EOF;
    leave;
  Endif;

  select;
  When %NullInd(OrHDly) = *On
    and %NullInd(OrHTot) = *Off and OrHTot <> *Zeros;
    %NullInd(OrHDly) = *Off;
    OrHDly = %Date();
  When %NullInd(OrHDly) = *On
    and %NullInd(OrHTot) = *Off and OrHTot = *Zeros;
    %NullInd(OrHTot) = *On;
  When %NullInd(OrHDly) = *Off
    and ( %NullInd(OrHTot) = *Off and OrHTot = *Zeros
      or %NullInd(OrHTot) = *On);
    %NullInd(OrHDly) = *On;
    %NullInd(OrHTot) = *On;
  EndS1;

  update OrdHeadF;

```

```

EndDo;

Return;
/End-Free

```

---

## 9.2.2 Using indicator variables in SQL

While the ALWNULL keyword and built-in function %NULLIND can only be used with native I/O, SQL values must be handled by using indicator variables.

Indicator variables can be used in these ways:

- ▶ To detect NULL values in host variables
- ▶ To verify that a retrieved value has not been truncated
- ▶ To set variables to NULL values in Insert or Update statements

An indicator variable must be defined as a 2-byte binary field that matches with the RPG definition 5I 0 and the SQL definition SMALL INTEGER. You specify an indicator variable (preceded by a colon) immediately after the host variable.

Example 9-7 shows how indicator variables can be defined as stand-alone fields and be used with host variables.

*Example 9-7 Using indicator variables in embedded SQL*

---

```

D DelDate      S          D
D OrderTotal   S          11P 2

D IndDelDate   S          5I 0
D IndOrderTotal S          5I 0
*-----
C/EXEC SQL
C+ Select OrHDly, OrHTot
C+   into :DelDate :IndDelDate, :OrderTotal :IndOrderTotal
C+   from OrdHead
C+   where OrHNbr = :OrderNo
C/END-EXEC

```

---

If a host structure is used for the retrieval values, you define a 2-byte binary (integer) array with as many elements as data structure subfields. You specify the indicator array (preceded by a colon) immediately after the host structure.

Example 9-8 shows how a host structure can be used with an indicator array.

*Example 9-8 Using an indicator array in embedded SQL*

---

```

D DsHostVar    DS
D DelDate      D
D OrdTotal     11P 2

D Arr1HostVar  S          5I 0 Dim(2)
*-----
C/EXEC SQL
C+ Select OrHDly, OrHTot
C+   into :DsHostVar :Arr1HostVar
C+   from OrdHead
C+   where OrHNbr = :OrderNo

```

C/END-EXEC

---

It is neither possible to use a indicator data structure nor single indicators in combination with a host structure. If you prefer named indicators, you can define a data structure with named indicators and overlay the data structure through an array.

Example 9-9 defines a data structure for the indicator values, that is overlaid by an array. In the SQL statement the indicator array is used.

*Example 9-9 Embedding an indicator array in a data structure*

---

```
D DSHostVar      DS
D  DelDate      D
D  OrdTotal     11P 2

D Ds2IndHostVar DS
D  Ind2DelDate  5I 0
D  Ind2OrdTotal 5I 0
D Arr2HostVar   5I 0 Dim(2) overlay(DS2IndHostVar)
*-----
C/EXEC SQL
C+ Select OrHDly, OrHTot
C+   into :DSHostVar :Arr2HostVar
C+   from OrdHead
C+   where OrHNbr = :OrderNo
C/END-EXEC

C           If      Ind2DelDate = -1           NULL value
C           EndIf
```

---

If a host structure array is used, you define a data structure containing as many subfields as the host structure array, and add either the keyword OCCUR(Elements) to create a multi occurrence data structure or keyword DIM(Elements) to create an array data structure.

### Checking NULL values through indicator variables

An indicator variable is used to indicate whether its associated host variable has been assigned a null value:

- ▶ If the value for the result column is NULL, SQL puts a -1 in the indicator variable.
- ▶ If you do not use an indicator variable and the result column is a NULL value, a negative SQLCODE is returned.
- ▶ If the value for the result column causes a data mapping error, SQL sets the indicator variable to -2.

Example 9-10 shows how indicator variables can be checked in RPG.

*Example 9-10 Checking indicator variables in RPG*

---

```
/Free
select
when Ind2DelDate = -1;      //NULL value
    DelDate = %Date();
when Ind2DelDate = -2;      //Data mapping Error
    DelDate = *LoVal;
EndSL;
/End-Free
```

---

## Verifying retrieval string length through indicator variables

You can also use an indicator variable to verify that a retrieved string value has not been truncated.

- ▶ If truncation occurs, the indicator variable contains a positive integer that specifies the original length of the string.
- ▶ If the string represents a large object (LOB), and the original length of the string is greater than 32767, the value that is stored in the indicator variable is 32767, since no larger value can be stored in a half word integer. When the database manager returns a value from a result column, you can test the indicator variable.
- ▶ If the value of the indicator variable is less than zero, you know the value of the results column is null. When the database manager returns a null value, the host variable will be set to the default value for the result column.

## Indicator variables used to set NULL values

You can use an indicator variable to set a NULL value in a column. When processing UPDATE or INSERT statements, SQL checks the indicator variable (if it exists).

- ▶ If the indicator variable contains a negative value, the column value is set to null.
- ▶ If the indicator variable contains a value greater than -1, the associated host variable contains a value for the column.

Example 9-11 shows how an indicator variable can be used to set columns to NULL values.

*Example 9-11 Indicator variables used to set NULL values*

---

```
D OrderTotal      S          11P 2
D IndOrderTotal  S          5I 0
*-----*
C                  eval      OrderTotal  = 100
C                  eval      IndOrderTotal = -1

C/EXEC SQL
C+ Update ITS04710/ORDHEAD
C+   set  ORHTOT = :OrderTotal :IndOrderTotal
C+   Where ORHDLY = Date('2004-09-02')
C/END-EXEC

C                  Return
```

---

You can directly use the SQL special word NULL to set a column to a NULL value.

Example 9-12 shows how columns can be updated by directly specifying the NULL value.

*Example 9-12 Setting to NULL values*

---

```
update ITS04710/Order_Header
      set Order_Total = NULL
      where Order_Delivery > current date;
```

---

## 9.2.3 Particular characteristics of NULL values in SQL statements

Because NULL values are out of the valid range a data type can have, NULL values cannot be compared by using >, <, <>, or =.

If you want to select NULL values, you have to add MyField IS NULL in the where clause. IS NOT NULL can be used to select all rows that do not contain any NULL VALUES in a particular column.

Example 9-13 shows how NULL values can be selected.

*Example 9-13 Selection of NULL values*

---

```
select *
  from Order_Header
  where (Order_Date = Current Date or Order_Date is NULL)
        and Order_Delivery is not NULL
```

---

If you want to count rows using SQL column function COUNT(\*), all rows are counted, even if one or several rows contain only NULL values. If you use COUNT(FieldName) instead, and FieldName contains NULL values, only the rows without NULL value are considered.

If you want to calculate the average using the SQL column function AVG(FieldName) and one or more rows contain a NULL value, they are not considered. Let us assume that we have three rows, containing, 2, 4, and NULL; the average will be 3.

Other SQL column functions like STDDEV (to calculate the biased standard deviation) and VARIANCE (to calculate the biased variance) do not consider rows containing NULL values either.

If you want to calculate the average, standard deviation, or variance over all rows, you have to convert the NULL values into default values. This can be done by using the scalar function COALESCE or VALUE.

**Note:** COALESCE should be preferred for conformance to the SQL 1999 standard.

The following example shows how the NULL value can be replaced by a zero using SQL scalar function COALESCE:

```
SELECT Avg(Coalesce(ORER_TOTAL, 0)) FROM ORDER_HEADER
```

There is an SQL scalar function NULLIF that converts specified values into NULL values. The following example shows how a zero value can be replaced through a NULL value using the SQL scalar function NULLIF:

```
SELECT Avg(NULLIF(ORER_TOTAL, 0)) FROM ORDER_HEADER
```

## 9.3 Date and time calculation

There are a lot of applications that restore all date and time information in numeric or character fields. When modernizing our database, we should consider converting these numeric or character fields into real date or time fields.

In the first step, we can add additional fields in our tables containing the date and time information. Then we have to fill the new fields by translating the existent numeric or character values into real date or time information. To guarantee that the new fields are always updated, we can add before update triggers for the numeric and alphanumeric fields that fill the date and time values.

After having modernized our tables, we can modernize the date and time calculation in our programs, using the date and time fields instead of the numeric or character date and time

values. RPG and SQL both provide a set of functions to facilitate date calculation, but they handle the date calculation differently. This may be an advantage, because you can always choose the best method.

If all programs use only the new date and time fields, the original numeric and character fields as well as the trigger that updates the date and time fields can be removed. In the following section we show RPG just as in SQL:

- ▶ How to convert numeric or character date or time fields into real date and time fields and vice versa
- ▶ How to check valid dates in numeric or character fields
- ▶ How to trap the system and job date
- ▶ How to calculate time differences and how to add and subtract time durations.
- ▶ We will present you with some useful SQL scalar functions

### 9.3.1 Converting from numeric/character date values to real date values

We need some mechanisms to get our numeric or character fields converted to DATE formats.

#### Converting from numeric/character date values to date in RPG

RPG provides several methods to convert numeric or alphanumeric date or time values into date or time values.

- ▶ operation code MOVE
- ▶ Built-in functions %DATE, %TIME, %TIMESTAMP
- ▶ Overlaying subfields in data structures

#### **operation code MOVE**

The operation code MOVE can only be used in classical RPG coding. It is not supported in RPG free format.

In factor 1 the date or time format of the numeric or alphanumeric date or time can be specified. If factor 1 is \*Blanks, \*ISO format is used as default.

If the date or time is alpha numeric, but does not contain any date or time separator, zero must be added to the date or time format. Date separators can be added in factor 1 for two-digit year formats. Time separators can be added for time format \*HMS.

**Note:** The date or time format cannot be used as a variable. If alternate date or time formats are needed, you have to code a separate statement for each data or time format.

Example 9-14 shows how numeric and alphanumeric dates can be converted to dates using the operation code MOVE.

*Example 9-14 Converting numeric and alphanumeric values to dates by using MOVE operation code*

D DateIso	S	D
D DateN4Year	S	8P 0
D DateN2Year	S	6P 0
D DateA4YearSep	S	10A
D DateA2YearSep	S	8A
D DateA4Year	S	8A

```

D DateA2Year      S          6A
*-----*
C                  eval      DateN4Year = 20040826
C                  move      DateN4Year   DateIso
C   DateIso       Dsply
C
C                  eval      DateN4Year = 27082004
C   *Eur          move      DateN4Year   DateIso
C   DateIso       Dsply
C
C                  eval      DateN2Year = 082504
C   *MDY          move      DateN2Year   DateIso
C   DateIso       Dsply
C
C                  eval      DateA4YearSep = '2004-08-26'
C                  move      DateA4YearSep DateIso
C   DateIso       Dsply
C
C                  eval      DateA4Year = '27082004'
C   *Eur0         move      DateA4Year   DateIso
C   DateIso       Dsply
C
C                  eval      DateA2YearSep = '08-25-04'
C   *MDY-         move      DateA2YearSep DateIso
C   DateIso       Dsply
C
C                  eval      DateA2Year = '082704'
C   *MDY0         move      DateA2Year   DateIso
C   DateIso       Dsply
C
C                  Return

```

### **Built-in functions %DATE, %TIME, %TIMESTAMP**

The operation code MOVE is not supported in RPG free format. To convert numeric or alphanumeric fields in free format RPG into date or time values, the built-in functions %DATE, %TIME, or %TIMESTAMP must be used.

The three built-in functions can have between 0 and 2 parameters.

If no parameter is specified, the current date, time, or timestamp is used. In parameter 1 the alpha numeric, numeric field, or a date string can be specified. If the date or time format is not \*ISO, the date or time format of the alpha numeric or numeric date must be specified in parameter 2.

Example 9-15 shows how numeric and alpha numeric dates can be converted to dates using the built-in function %DATE.

#### *Example 9-15 Converting numeric and alphanumeric values to dates using built-in function %DATE()*

```

D DateIso      S          D
D DateN4Year   S          8P 0
D DateN2Year   S          6P 0
D DateA4YearSep S          10A
D DateA2YearSep S          8A
D DateA4Year   S          8A
D DateA2Year   S          6A
*-----*

```

```

/Free
DateN4Year = 20040826;
DateIso    = %Date(DateN4Year);
Dsply DateIso;

DateN4Year = 27082004;
DateIso    = %Date(DateN4Year: *Eur);
Dsply DateIso;

DateN2Year = 082504;
DateIso    = %Date(DateN2Year: *MDY);
Dsply DateIso;

DateA4YearSep = '2004-08-26';
DateIso      = %Date(DateA4YearSep);
Dsply DateIso;

DateA4Year = '27082004';
DateIso    = %Date(DateA4Year: *Eur0);
Dsply DateIso;

DateA2YearSep = '08-25-04';
DateIso      = %Date(DateA2YearSep: *MDY-);
Dsply DateIso;

DateA2Year = '082704';
DateIso    = %Date(DateA2Year: *MDY0);
Dsply DateIso;

Return;
/END-FREE

```

---

### **Overlaying subfields in data structures**

This method can only be used for character dates or times containing date or time separators. The character fields can be defined as data structure subfields and be overlaid by fields with date or time data types. Depending on the date format the character field can have, the keyword DATFMT or TIMFMT with the corresponding format must be added.

Example 9-16 shows a data structure where character fields are overlaid by date fields. Additionally, it is shown how the character and date fields can be used.

#### *Example 9-16 Overlaying alphanumeric fields with dates in data structures*

---

```

H DEBUG DatFmt(*ISO) TimFmt(*ISO)
*-----
D DsDate          DS
D DateCharIso      10A
D DateIso          D overlay(DateCharIso)
D DateCharEur     10A
D DateEur         D overlay(DateCharEur) DatFmt(*Eur)
D DateCharMDY     8A
D DateMDY         D overlay(DateCharMDY) DatFmt(*MDY)
*-----
/Free
DateCharIso = '2004-08-29';
Dsply DateIso;

DateCharEur = '01.07.2004';
Dsply DateEur;

```

```
DateCharMDY = '08/29/2004';
Dsply DateMDY;

Return;
/End-Free
```

---

## Converting from numeric/character date values to date in SQL

In SQL the scalar functions DATE, TIME, and TIMESTAMP can be used to convert character strings and numeric values into date, time, or timestamp data types.

In contrast to RPG, you cannot convert numeric date or time representations directly into date or time formats. You first have to convert your numeric field into a character string containing a valid date or time representation.

If you have to convert numeric fields into dates, it would be easier to use the RPG functions. Just write a small function in RPG and create from this function an UDF that can be used in SQL.

### Scalar function DATE

Using the scalar function DATE, any representation of a date format with a four-digit year can be converted, independent from the date format that is currently used. Both single-byte and double-byte character representation of a date can be converted.

A string with an actual length of 7 that represents a valid date in the form *YYYYNNN*, where *YYYY* are digits denoting a year, and *NNN* are digits between 001 and 366 denoting a day of that year.

Example 9-17 shows an INSERT statement, where different date columns are filled with dates, and every date is based on a character string in a different date format.

*Example 9-17 Using the SQL scalar function DATE to convert character strings to dates*

---

```
Insert into MySchema/MyTable
  Values(date('2004-08-27'),
        date('28.08.2004'),
        date('08/29/2004'),
        date('2004245'))
```

---

Example 9-18 leads to the same result as Example 9-17.

*Example 9-18 Inserting character strings into date fields with SQL*

---

```
Insert into MySchema/MyTable
  Values('2004-08-27',
        '28.08.2004',
        '08/29/2004',
        '2004245')
```

---

If your character string contains only a 2-digit year, you have to convert to a 4-digit year. This can be done using a case expression.

Example 9-19 shows how a date field can be updated based on a character field with a date representation like YY-MM-DD.

*Example 9-19 Converting a character string with a 2-digit year with SQL*

---

```
update MySchema/MyTable
```

```

set MyDate = Date(case when substr(MyDateAlpha, 1, 2)
                    between '40' and '99'
                    then '19'
                    else '20'
                    end
                  concat MyDateAlpha)
Where MyDate is NULL

```

---

If you frequently have to deal with character string conversion to dates, it is a good idea to create an user defined function.

It is also possible to convert numeric values with the scalar function DATE, but in contrast to RPG, the numeric value represents the number of days since 0001-01-01. For example, 731 587 represents July 1st, 2004.

To determine the number of dates since 0001-01-01, the scalar function DAYS(Date) can be used.

### **Scalar function TIME**

The scalar function TIME allows you to convert a valid character representation of a valid time format into a time field, independent from the time format that is currently used. The character string can either contain single or double byte characters.

Example 9-20 shows how time can be inserted using character strings in different time formats.

*Example 9-20 Inserting character string containing time values into time fields with SQL*

---

```

Insert into MySchema/MyTable
  values(time('18:47:22'),
         time('18.48.22'),
         time('06:47 PM'))

```

---

### **Scalar function TIMESTAMP**

The scalar function TIMESTAMP allows you to convert a valid character representation (see Chapter 4, "Modernizing database definitions" on page 29) of a valid date or timestamp format into a timestamp.

A timestamp function returns a string with an actual length of 7 that represents a valid date in the form *YYYYNNN*, where *YYYY* are digits denoting a year, and *NNN* are digits between 001 and 366 denoting a day of that year.

A timestamp function returns a string with an actual length of 14 that represents a valid date and time in the form *YYYYXXDDHHMMSS*, where *YYYY* is year, *XX* is month, *DD* is day, *HH* is hour, *MM* is minute, and *SS* is seconds.

Example 9-21 shows an INSERT statement where different timestamp formats are used.

*Example 9-21 Inserting character strings containing timestamp values into timestamps with SQL*

---

```

Insert into MySchema/MyTable
  values(Timestamp('2004-08-31-18.23.45.123456'),
         Timestamp('20040831182345'),
         Timestamp('2004-08-31', '18.12.34'))

```

---

## 9.3.2 Converting from date fields to character or numeric representation

There may be some reasons why you have to restore your date and time fields as numeric or character values in your files. But to deal with the functions that RPG and SQL provide you need date fields. You have already learned how to convert numeric and character strings to date fields. Now let us see how date and time fields can be converted to numeric or alphanumeric representations.

### Converting from date to character or numeric date in RPG

RPG provides several methods to convert date or time fields into character or numeric values.

- ▶ Operation code MOVE
- ▶ Built-in functions %CHAR and %DEC

#### Operation code MOVE

The operation code MOVE can only be used in classical RPG coding. It is not supported in RPG free format.

In factor 1 the date or time format of the numeric or alphanumeric date or time can be specified. If factor 1 is \*Blank, \*ISO format is used as default.

If the date or time is character representation without any date or time separator, zeros must be added to the date or time format. Date separators can be added in factor 1 for two-digit year formats. Time separators can be added for time format \*HMS.

**Note:** The date or time format cannot be used as a variable. If alternate date or time formats are needed, you have to code a separate statement for each data or time format.

Example 9-22 shows how date fields can be converted to character or numeric representation using the operation code MOVE.

*Example 9-22 Converting dates to character or numeric representation using operation code MOVE*

---

D MyDateAlpha	S		10A	
D MyDateNum	S		8P 0	
D MyDateIso	S		D	
*-----				
C		eval		MyDateIso = D'2004-08-29'
C		Move	MyDateIso	MyDateAlpha
C	MyDateAlpha	dsply		
C	*Eur	Move	MyDateIso	MyDateAlpha
C	MyDateAlpha	dsply		
C	*MDY0	MoveL(P)	MyDateIso	MyDateAlpha
C	MyDateAlpha	dsply		
C		Move	MyDateIso	MyDateNum
C	MyDateNum	dsply		
C		Return		

---

#### Built-in functions %CHAR and %DEC

Because the operation code MOVE is not supported in free format RPG, built-in functions must be used instead.

The built-in function %CHAR allows you to convert date, time, or timestamp fields into character representations. The second parameter is optional and contains the date or time format the character string must have. If the second parameter is not specified, the date or time format \*ISO is used.

If the character string must not contain date or time separators, a zero must be added to the date or time format.

Example 9-23 shows how date fields can be converted to character representations using the built-in function %CHAR.

*Example 9-23 Converting date fields to character representation using the built-in function %CHAR()*

---

```

D MyDateAlpha      S          10A
D MyDateIso        S          D
*-----
/Free
  MyDateIso = D'2004-08-29';

  MyDateAlpha = %Char(MyDateIso);
  Dsply MyDateAlpha;

  MyDateAlpha = %Char(MyDateIso: *Eur);
  Dsply MyDateAlpha;

  MyDateAlpha = %Char(MyDateIso: *MDY0);
  Dsply MyDateAlpha;

  Return;
/End-Free

```

---

Beginning with release V5R3M0, date and time fields can be directly converted to numeric fields using the built-in function %DEC.

Before release V5R3M0, the built-in function %DEC could only be used to format numeric values or to convert character representations to a numeric field.

When the first parameter of the built in function %DEC is a date, time, or timestamp expression, the optional second format parameter specifies the format of the value returned. The converted decimal value will have the number of digits that a value of that format can have, and zero decimal positions. For example, if the first parameter is a date, and the format is \*YMD, the decimal value will have six digits.

Example 9-24 shows how date fields can be converted into numeric fields using the built-in function %DEC.

*Example 9-24 Converting date fields into numeric values using the built-in function %DEC()*

---

```

D MyDateNum        S          8P 0
D MyDateNum6       S          6P 0
D MyDateIso        S          D
*-----
/Free
  MyDateIso = D'2004-08-29';
  MyDateNum = %Dec(MyDateIso);
  Dsply MyDateNum ;

  MyDateNum = %Dec(MyDateIso: *Eur);
  Dsply MyDateNum ;

```

```

MyDateNum6 = %Dec(MyDateIso: *MDY);
Dsply MyDateNum6;

Return;
/End-Free

```

---

Prior to release V5R3M0, you either have to use a combination of built-in-functions, %CHAR and %INT or %CHAR and %DEC, or use operation code MOVE.

Example 9-25 shows how date fields can be converted to numeric fields using the built-in functions %CHAR and %INT or %DEC.

*Example 9-25 Converting date fields to numeric representation with different built-in functions*

---

```

D MyDateNum      S              8P 0
D MyDateIso      S              D
*-----*
/Free
  MyDateIso      = D'2004-08-29';

  MyDateNum      = %Int(%Char(MyDateIso: *IS00));
  Dsply MyDateNum;

  MyDateNum      = %Dec(%Char(MyDateIso: *USA0): 8: 0);
  Dsply MyDateNum;

Return;
/End-Free

```

---

## Converting from date to character or numeric date in SQL

To convert date or time fields into character representations, SQL delivers the scalar function CHAR.

Like in RPG, the first parameter represents the date or time field, while the second is optional and represents the date or time format of the returned value. In SQL the date and time format must be specified without a leading asterisk (\*).

**Note:** In contrast to RPG, only four-digit year date formats can be specified as the second parameter.

If no date or time format is specified, the job's date format or the format that is defined in the set option or the compile parameter in an embedded SQL program is used. In this way you can get a two-digit year representation.

**Note:** If in RPG no date format is specified, the \*ISO format is used. If in SQL no date format is specified, the job's format is specified.

If you need a date representation with a two-digit year, it is better to convert it with RPG. If you need those conversions frequently, just write a small function in RPG and create an UDF.

Example 9-26 shows how the scalar function CHAR can be used to convert date fields into character representations.

*Example 9-26 Converting dates to character representation using the SQL scalar function CHAR*

---

```
Select char(MyDate),      char(MyDate, ISO),
      char(MyDate, Eur), char(MyDate, USA)
from MyTable
```

---

Using SQL to convert date or time fields into numeric representation is a little tricky. You have to convert your date or time field into a character representation without date or time separators, and then cast the result into a numeric field.

If you need to convert date or time fields, it would be wise to create a little function in RPG and convert it into a UDF.

Example 9-27 shows how a date field can be converted into a numeric representation within SQL.

*Example 9-27 Converting a date into a numeric representation using SQL*

---

```
update MySchema/MyTable
  set MyDateNum = cast(substr(char(MyDate, ISO), 1, 4) concat
                      substr(char(Mydate, ISO), 6, 2) concat
                      substr(char(MyDate, ISO), 9, 2)
                      as dec(8, 0))
  where MyDateNum = 0 or MyDateNum is NULL
```

---

### 9.3.3 Checking for a valid date or time

Before converting numeric or character strings to date or time fields, you may wish to check if the string contains valid date or time values.

#### Checking valid date or time in RPG

RPG provides two methods to check for valid date or time in character or numeric strings:

- ▶ Operation code TEST
- ▶ Monitor group and built-in functions %DATE, %TIME or %TIMESTAMP

#### **Operation code TEST**

The TEST operation code allows users to test the validity of date, time, or timestamp fields prior to using them.

According to whether date, time, or timestamp values must be checked, you have to use the operation code TEST with different extenders:

- ▶ Extender (D) to check a valid date
- ▶ Extender(T) to check a valid time
- ▶ Extender(Z) to check a valid timestamp

Additionally, you have to specify extender (E) to initialize the %Status and %Error indicator.

In format 1 you can specify the date, time, or timestamp format that has to be checked. If factor 1 is blank, format \*ISO is checked.

If the content of the string operand is not valid, program status code 112 is signaled. Then the error indicator is set on or the %ERROR built-in function is set to return '1', depending on the error handling method specified.

Example 9-28 shows how a date string can be checked by using the operations code TEST.

*Example 9-28 Checking for valid date and time values using operation code TEST*

---

```

D MyDateAlpha      S          10A
D MyTimeNum        S          6P 0

D MyDate           S          D
D MyTime           S          T
*-----
/Free
  MyDateAlpha = '09/31/2004';
  Test(ED) *USA MyDateAlpha;
  If %Error;
    MyDate = *LoVal;
  else;
    MyDate = %Date(MyDateAlpha);
  Endif;

  MyTimeNum = 250026;
  Test(ET) MyTimeNum;
  If %Status = 112;
    MyTime = *LoVal;
  Else;
    MyTime = %Time(MyTimeNum);
  EndIf;

  Return;
/End-Free

```

---

### **Monitor group and built-in functions %DATE, %TIME, or %TIMESTAMP**

Instead of using the operation code TEST, you can try to convert your date or time expression into a date or time field by using one of the built-in functions %DATE, %TIME, or %TIMESTAMP. If a data mapping error occurs, status 112 is returned. All status can be trapped by using a monitor group.

Example 9-29 shows how a date can be checked by using the built-in function %DATE and a monitor group.

*Example 9-29 Checking for valid date and time values using a monitor group*

```

D MyDateAlpha      S          10A
D MyTimeNum        S          6P 0

D MyDate           S          D
D MyTime           S          T
*-----
/Free
  Monitor;
    MyDateAlpha = '2004-09-31';
    MyDate      = %Date(MyDateAlpha);
  On-Error 112;
    MyDate      = *LoVal;
  EndMon;

  Monitor;
    MyTimeNum = 250026;
    MyTime    = %Time(MyTimeNum);
  On-Error;
    MyTime    = *LoVal;
  EndMon;

```

```
Return;  
/End-Free
```

---

### Checking valid date or time in SQL

SQL does not provide a separate function to check for valid dates or times. If you need this functionality in SQL it is the best to write a small function in RPG.

Nevertheless, you can use an indicator variable to detect data mapping errors. If -2 is returned in the indicator variable, the character or numeric string contains an invalid date or time.

Example 9-30 shows how an indicator variable can be used to check for valid dates or times.

*Example 9-30 Checking for valid date and time values using SQL*

---

```
D MyDateAlpha    S          10A  
D MyTimeAlpha   S           8A  
  
D MyDate        S           D  
D IndMyDate     S          5I 0  
  
D MyTime        S           T  
D IndMyTime     S          5I 0  
*-----  
C                eval    MyDateAlpha = '2004-09-31'  
C/EXEC SQL  
C+ Set :MyDate :IndMyDate = :MyDateAlpha  
C/END-EXEC  
C                If      IndMyDate = -2  
C                eval    MyDate = *LoVal  
C                Endif  
  
C                eval    MyTimeAlpha = '13:00 PM'  
C/EXEC SQL  
C+ Set :MyTime :IndMyTime = :MyTimeAlpha  
C/END-EXEC  
C                If      IndMyTime = -2  
C                eval    MyTime = *LoVal  
C                Endif  
C    MyTime      dsply  
  
C                Return
```

---

### 9.3.4 Retrieving current date and time

There are a lot of situations where the current date or time must be used, for example, to print invoices or to calculate the best before date.

#### Retrieving system date and job date in RPG

In RPG you can either retrieve the job or the system date. While the system date is the current date, the job date is the date when the job is started. In most cases both dates are identical, but there are situations when they are different, for example, when a job is started on Monday morning and ended on Friday evening.

#### Retrieving the system date in RPG

To retrieve the system date or time RPG provides several methods:

► Operation code TIME

The operation code TIME can only be used in classical RPG. In free format coding, built-in functions must be used instead.

Depending on the format of the result field, time or date and time are returned either as numeric with a two-digit or four-digit year or as date, time, or timestamp values.

Example 9-31 shows how the operation code TIME can be used to retrieve the system time.

*Example 9-31 Retrieving system time using operation code TIME*

---

```

D SysDate      S          D
D SysTime      S          T
D SysStamp     S          Z
D SysDateTimeN S          14P 0
*-----*
C              time          SysDate
C  SysDate     Dsply
C              time          SysTime
C  SysTime     Dsply
C              time          SysStamp
C  SysStamp    Dsply
C              time          SysDateTimeN
C  SysDateTimeN Dsply
C              Return

```

---

► Built-in functions %DATE, %TIME, and %TIMESTAMP

If the built-in functions %DATE, %TIME, or %TIMESTAMP are used without specifying any parameter, the current date, time, or timestamp is retrieved.

Example 9-32 shows how the current time, date, and timestamp can be retrieved by using built-in functions.

*Example 9-32 Retrieve the system date and time by using built-in functions*

---

```

D SysDate      S          D
D SysTime      S          T
D SysStamp     S          Z
*-----*
/Free
  SysDate = %Date();
  Dsply SysDate;

  SysTime = %Time();
  Dsply SysTime;

  SysStamp = %Timestamp();
  Dsply SysStamp;

  Return;
/End-Free

```

---

► Initializing date and time fields in the Definition specifications

Date, time, and timestamp fields can be initialized with either the system or the job date and time, by using the special values \*SYS or \*JOB with the INZ keyword.

**Note:** If a field is initialized in the global Definition specifications, the initialization is only executed the first time the program or service program is called. This must be considered when a program is ended with RETURN instead of \*INLR and the appropriate activation group is not \*NEW.

If the field is defined in the local Definition specification without specifying the keyword STATIC, the initialization is executed every time the procedure is called.

Example 9-33 shows how date and time fields can be initialized with system date and time.

*Example 9-33 Initialization of date and time fields with system time*

---

```
D SysDate      S          D   inz(*Sys)
D SysTime     S          T   inz(*Sys)
D SysStamp    S          Z   inz(*Sys)
```

---

### **Retrieving the job date in RPG**

RPG provides several methods to retrieve the job date

- ▶ Using the *user date special words*

UPDATE and \*DATE specify the job date. UPDATE returns a numeric date with a 2-digit year while \*DATE returns a numeric date with a 4-digit year. The format of UPDATE or \*DATE depends on the format specified in the DATEDIT keyword in the control specifications, or if not specified the job format is used.

- With UYEAR and \*YEAR the year portion of the job date is returned. UYEAR returns a 2-digit year, while \*YEAR returns a 4-digit year.
- With UMONTH or \*MONTH the month portion of the job date is returned.
- With UDAY or \*DAY the day portion of the job is returned.

- ▶ Using the *built-in function %DATE()*

The date special word can be used with the built-in function %DATE to get the job date.

Example 9-34 shows how the job date can be retrieved using the built-in function %DATE and the date special word \*DATE.

*Example 9-34 Retrieving the job date by using the built-in function %Date()*

---

```
D MyJobDate    S          D
*-----
/Free
  MyJobDate = %Date(*Date);
  Dsply MyJobDate;

  Return;
/End-Free
```

---

- ▶ Initializing date field in the Definition specifications

Date, time, and timestamp fields can be initialized with either the system or the job date and time, by using the special values \*SYS or \*JOB with the INZ keyword.

The following example shows how date and time fields can be initialized with the job date:

```
D JobDate      S          D   inz(*Job)
```

## Retrieving current date in SQL

SQL provides a couple of special registers and scalar functions to get the current date, time, or timestamp, based on a reading of the time-of-day clock. But it is not possible to retrieve the job date with SQL.

- ▶ Special registers
  - Current Time or Current\_Time specifies the current time.
  - Current Date or Current\_Date specifies the current date.
  - Current Timestamp or Current\_Time stamp specifies the current time stamp.

**Note:** The SQL 1999 Core standard uses the form with the underscore.

- ▶ Scalar functions
  - CURTIME retrieves the current time.
  - CURDATE retrieves the current date.
  - NOW retrieves the current timestamp.

**Note:** Special registers should be used for maximal portability

## 9.3.5 Adding and subtracting date and time values

The most interesting thing in date calculation is not how to convert and initialize date and time fields, but how to calculate with date and time values.

### Adding and subtracting date and time values with RPG

RPG provides two methods to add and subtract time values.

- ▶ Operation code ADDDUR and SUBDUR
- ▶ Built-in functions

#### ***Operation code ADDDUR and SUBDUR***

The Operation code ADDDUR and SUBDUR can only be used in classical RPG. In free format built-in functions must be used instead.

The ADDDUR adds the duration specified in factor 2 to a date or time and places the resulting date, time, or timestamp in the result field.

The SUBDUR operation can be used to subtract a duration specified in factor 2 from a field or constant specified in factor 1 and place the resulting date, time, or timestamp in the field specified in the result field.

Factor 1 is optional and may contain a date, time, or timestamp field, subfield, array, array element, literal or constant. If factor 1 contains a field name, array, or array element, then its data type must be the same data type as the field specified in the result field. If factor 1 is not specified, the duration is added to the field specified in the result field.

Factor 2 is required and contains two subfactors. The first is a duration and may be a numeric field, array element, or constant with zero decimal positions. If the duration is negative then it is subtracted from the date. The second subfactor must be a valid duration code indicating the type of duration.

Table 9-12 on page 211 shows the valid duration codes.

Table 9-12 Duration codes

Unit	Duration code		Valid for
	long	short	
Year	*YEARS	*Y	Timestamp and Date
Month	*MONTHS	*M	Timestamp and Date
Day	*DAYS	*D	Timestamp and Date
Hour	*HOURS	*H	Timestamp and Time
Minute	*MINUTES	*MN	Timestamp and Time
Second	*SECONDS	*S	Timestamp and Time
Microsecond	*MSECONDS	*MS	Timestamp

The duration code must be consistent with the result field data type:

- ▶ To a date field only years, months, or days can be added or subtracted.
- ▶ To a time field only hours, minutes, or seconds can be added or subtracted.
- ▶ To a timestamp field years, months, days, hours, minutes, seconds, or microseconds can be added or subtracted.

**Note:** If you add and subtract one month from a date, the result may not be the same date as the original date. For example, if you add one month to January 31st, the result will be February 28th or 29th. If you then subtract one month, the result will be January 28th or 29th.

Example 9-35 shows how durations can be added and subtracted using the operation codes ADDDUR and SUBDUR.

Example 9-35 Adding/subtracting durations using operation codes ADDDUR and SUBDUR

---

D MyDate	S	D	
D MyTime	S	T	
D NbrDays	S	3U 0 inz(30)	
D NbrMin	S	3U 0 inz(50)	
*-----			
C	eval	MyDate = %Date()	
C	SubDur	1:*Months	MyDate
C	MyDate	dsply	
C	AddDur	NbrDays:*Days	MyDate
C	MyDate	dsply	
C	eval	MyTime = %Time()	
C	SubDur	450:*S	MyTime
C	MyTime	dsply	
C	AddDur	NbrMin:*MN	MyTime
C	MyTime	dsply	
C	Return		

---

### Built-in functions

To add or subtract time or date values to a date or time, the numeric values must be converted into date or time compatible values.

RPG provides a couple of built-in functions to convert numeric values into date or time compatible values:

- ▶ %YEARS(NumValue) to add or subtract a number of years to a date or timestamp
- ▶ %MONTHS(NumValue) to add or subtract a number of months to a date or timestamp
- ▶ %DAYS(NumValue) to add or subtract a number of days to a date or timestamp
- ▶ %HOURS(NumValue) to add or subtract a number of hours to a time or timestamp
- ▶ %MINUTES(NumValue) to add or subtract a number of minutes to a time or timestamp
- ▶ %SECONDS(NumValue) to add or subtract a number of seconds to a time or timestamp
- ▶ %MSECONDS(NumValue) to add or subtract a number of microseconds to a timestamp

Example 9-36 shows how built-in functions can be used to add and subtract date and time values.

*Example 9-36 Adding/subtracting durations using built-in functions*

---

```

D MyDate          S          D
D MyTime          S          T

D NbrDays         S          3U 0 inz(30)
D NbrMin          S          3U 0 inz(50)
*-----
/Free
  MyDate = %Date() - %Months(1) + %Days(NbrDays);
  Dsply MyDate;

  MyTime = %Time() - %Seconds(450) + %Minutes(NbrMin);
  Dsply MyTime;

  Return;
/End-Free

```

---

**Note:** If you are dealing with microseconds in RPG, only the first three digits are supported. In contrast, SQL supports all six digits.

For example, if you use the built-in function %TIMESTAMP(), the returned values may be '2004-08-30-20.05.47.123000'. If you use CURRENT\_TIMESTAMP in SQL the result may be '2004-08-30-20.05.47.123456'.

To build a timestamp value, a date and a time value can be added by using a simple plus sign (+).

Example 9-37 shows how a date and a time value can be added to a timestamp value.

*Example 9-37 Creating a timestamp value from a date and a time value*

---

```

D MyDate          S          D
D MyTime          S          T
D MyTimeStamp     S          Z
*-----
/FREE
  MyDate = %Date() - %Days(3);
  MyTime = %Time() - %Hours(5);
  MyTimeStamp = MyDate + MyTime;
  Dsply MyTimeStamp;

  MyTimeStamp = %Date() - %Days(7) + %Time() - %Hours(3);
  Dsply MyTimeStamp;

```

```
Return;  
/END-FREE
```

---

### Adding and subtracting date and time values with SQL

To add or subtract date or time values with SQL no additional functions are needed. You only specify the number of days or months or any other compatible date or time portion and add a labeled duration, for example, DAYS or MONTHS, etc., in words.

The valid labeled durations are:

- ▶ YEAR / YEARS
- ▶ MONTH / MONTHS
- ▶ DAY / DAYS
- ▶ HOUR / HOURS
- ▶ MINUTE / MINUTES
- ▶ SECOND / SECONDS
- ▶ MICROSECOND / MICROSECONDS

Example 9-38 shows how date and time values can be added or subtracted in SQL.

*Example 9-38 Adding subtracting durations using SQL*

---

```
D MyDate          S          D  
D MyTime          S          T  
  
D NbrDays         S          3U 0 inz(30)  
D NbrMin          S          3U 0 inz(50)  
*-----  
C/Exec SQL  
C+ Set :MyDate = Current_Date - 1 Month + :NbrDays Days  
C/End-Exec  
  
C    MyDate      dsply  
  
C/Exec SQL  
C+ Set :MyTime = Current_Time - 450 Seconds + :NbrMin Minutes  
C/End-Exec  
  
C    MyTime      dsply  
  
C          Return
```

---

### 9.3.6 Calculating date and time differences

To calculate with date and time values means not only to add or subtract time durations but also to calculate the difference between two dates or time values.

RPG provides two methods to calculate time differences:

- ▶ Operation code SUBDUR
- ▶ Built-in function %DIFF

Time difference calculated in RPG can only contain one date or time type, but never a combination of all.

**Note:** The result is rounded down, with any remainder discarded. For example, 61 minutes is equal to 1 hour, and 59 minutes is equal to 0 hours.

The result of the difference between a timestamp and a date or between two dates can be:

- ▶ A number of years
- ▶ A number of months
- ▶ A number of days

The result of the difference between a timestamp and a time or between two dates can be:

- ▶ A number of hours
- ▶ A number of minutes
- ▶ A number of seconds

The result between a timestamp and a timestamp can be:

- ▶ A number of years
- ▶ A number of months
- ▶ A number of days
- ▶ A number of hours
- ▶ A number of minutes
- ▶ A number of seconds
- ▶ A number of microseconds

If you need to know how many hours and minutes and seconds are between two times, you have to calculate the time difference in seconds and then divide through 3600 to get the hours. The rest must be divided through 60 to get the minutes and the rest are the seconds.

SQL provides a much easier method. In SQL, date and times can be directly subtracted without using a scalar function. The result will be a numeric field that contains a combination of years, months, days, hours, minutes, seconds, and microseconds.

The result of the difference between two dates is a numeric 8-digit (8,0) result with:

- ▶ Position 1–4: Difference in years
- ▶ Position 5–6: Difference in months
- ▶ Position 7–8: Difference in days

The difference between January 1st, 2003 and July 3rd, 2004 is 10602; 1 year, 6 months and 2 days.

The result of the difference between two times is a numeric 6-digit (6,0) result with:

- ▶ Position 1–2: Difference in hours
- ▶ Position 3–4: Difference in minutes
- ▶ Position 5–6: Difference in seconds

The result of the difference between two timestamps is a numeric 20-digit with 6 decimal positions (20,6) result with:

- ▶ Position 1–4: Difference in years
- ▶ Position 5–6: Difference in months
- ▶ Position 7–8: Difference in days
- ▶ Position 9–10: Difference in hours
- ▶ Position 11–12: Difference in minutes
- ▶ Position 13–14: Difference in seconds
- ▶ Position 15–20: Difference in microseconds

If you want to subtract a date or time from a timestamp, the timestamp must be converted into a date or time value using the scalar functions DATE or TIME.

**Note:** While in RPG, a time difference can only be calculated for one date or time type. SQL provides a combination of the different date or time types. For example, if you subtract two times, RPG returns either hours *or* minutes *or* seconds, and SQL returns hours *and* minutes *and* seconds.

## Calculation date and time differences in RPG

RPG provides two methods to calculate date and time differences:

► **Operation code SUBDUR**

The operation code SUBDUR is only supported in classical RPG. If you want to code in free format you have to use the built-in function %DIFF instead.

The SUBDUR operation can also be used to calculate a duration between:

- Two dates
- A date and a timestamp
- Two times
- A time and a timestamp
- Two timestamps

Factor 1 and factor 2 must contain a compatible date, time, or timestamp field, subfield, array, array element, constant or literal.

The result field consists of two subfactors. The first is the name of a zero decimal numeric field, array, or array element in which the result of the operation will be placed. The second subfactor contains a duration code denoting the type of duration (look at Table 9-12 on page 211). The result field will be negative if the date in factor 1 is earlier than the date in factor 2.

Example 9-39 shows how time differences can be calculated using Operations-Code SUBDUR.

*Example 9-39 Calculation time differences using the operation code SUBDUR*

D	MyDate1	S		D	inz(D'2004-08-31')
D	MyTime1	S		T	inz(T'20.25.50')
D	MyTimeStamp1	S		Z	inz(Z'2004-08-30-10.15.45.000000')
D	MyDate2	S		D	inz(D'2004-04-09')
D	MyTime2	S		T	inz(T'14.30.55')
D	MyTimeStamp2	S		Z	inz(Z'2004-07-01-04.30.30.000000')
*-----					
C	MyDate1	SubDur	MyDate2	MyDiff:*Days	
C	MyDiff	Dsply			
C	MyTimeStamp1	SubDur	MyDate2	MyDiff:*Months	
C	MyDiff	Dsply			
C	MyTime1	SubDur	MyTime2	MyDiff:*S	
C	MyDiff	Dsply			
C	MyTimeStamp1	SubDur	MyTime2	MyDiff:*Hours	
C	MyDiff	Dsply			
C	MyTimeStamp1	SubDur	MyTimeStamp2	MyDiff:*S	
C	MyDiff	Dsply			
C		Return			

► Built-in function %DIFF

If you want to calculate time differences in free format RPG, you have to use the built-in function %DIFF.

%DIFF produces the difference (duration) between two date or time values. The first and second parameters must have the same or compatible types. The following combinations are possible:

- Date and date
- Time and time
- Timestamp and timestamp
- Date and timestamp (only the date portion of the timestamp is considered)
- Time and timestamp (only the time portion of the timestamp is considered)

The third parameter specifies the type of duration (look at Example 9-12 on page 211).

Example 9-40 shows how time differences can be calculated by using the built-in function %DIFF.

*Example 9-40 Calculating time differences using the built-in function %DIFF()*

---

```
D MyDate1      S          D  inz(D'2004-08-31')
D MyTime1      S          T  inz(T'20.25.50')
D MyTimeStamp1 S          Z  inz(Z'2004-08-30-10.15.45.000000')
```

```
D MyDate2      S          D  inz(D'2004-04-09')
D MyTime2      S          T  inz(T'14.30.55')
D MyTimeStamp2 S          Z  inz(Z'2004-07-01-04.30.30.000000')
```

```
D MyDiff       S          10I 0
*-----
/Free
MyDiff = %Diff(MyDate1: MyDate2: *Days);
Dsply MyDiff;

MyDiff = %Diff(MyTimeStamp1: MyDate2: *M);
Dsply MyDiff;

MyDiff = %Diff(MyTime1: MyTime2: *S);
Dsply MyDiff;

MyDiff = %Diff(MyTimeStamp1: MyTime2: *Hours);
Dsply MyDiff;

MyDiff = %Diff(MyTimeStamp1: MyTimeStamp2: *MN);
Dsply MyDiff;

Return;
/End-Free
```

---

### Calculating time differences with SQL

In SQL no additional scalar function is needed to subtract two time, date, or timestamp values.

The difference is presented as a combination of years, months, days, hours, minutes, seconds, and microseconds, depending on the used date and time values.

Example 9-41 on page 217 shows how the difference between two time values with the result in hours, minutes, and seconds is calculated by only using RPG.

**Example 9-41 Calculating time differences in hours, minutes, and seconds with RPG**

```
D MyTime1      S          T   inz(T'20.35.50')
D MyTime2      S          T   inz(T'14.30.45')

D MyDiff       S          10I  0

D DStime       DS
D   DStimeNum  6S  0
D   DsHours    2S  0 overlay(DStimeNum)
D   DsMinutes  2S  0 overlay(DStimeNum: *Next)
D   DsSeconds  2S  0 overlay(DStimeNum: *Next)
*-----
/Free
MyDiff      = %Diff(MyTime1: MyTime2: *Seconds);

DsHours     = %Div(MyDiff: 3600);
DsMinutes   = %Div(%Rem(MyDiff: 3600): 60);
DsSeconds   = %Rem(%Rem(MyDiff: 3600): 60);
Dsply DStime;

Return;
/End-Free
```

Example 9-42 shows the same result, but the calculation is done by SQL.

**Example 9-42 Calculating time differences in hours, minutes, and seconds with SQL**

```
D MyTime1      S          T   inz(T'20.35.50')
D MyTime2      S          T   inz(T'14.30.45')

D DStime       DS
D   DStimeNum  6S  0
D   DsHours    2S  0 overlay(DStimeNum)
D   DsMinutes  2S  0 overlay(DStimeNum: *Next)
D   DsSeconds  2S  0 overlay(DStimeNum: *Next)
*-----
C/EXEC SQL Set :DStimeNum = :MyTime1 - :MyTime2
C/End-EXEC
C   DStime     Dsply
C
C           Return
```

**Calculating time difference in days with SQL**

If you want to calculate the difference in days between two dates, you have to convert the date into the number of days since '0001-01-01' using the SQL scalar function DAYS and subtract the results.

Example 9-43 calculates the difference in days between two dates.

**Example 9-43 Calculating time difference in days using SQL**

```
D MyDate       S          D   inz(D'2004-07-01')
D NbrOfDays    S          10I  0
*-----
C/EXEC SQL
C+ Set :NbrOfDays = Days(Current_Date) - Days(:MyDate)
C/End-EXEC
C   NbrOfDays  Dsply
```

**Calculating time differences in seconds with SQL**

If you want to calculate the difference in seconds between two times, you have to convert your time into the number of seconds since midnight by using the SQL scalar function `MIDNIGHT_SECONDS` and subtract the results.

Example 9-44 calculates the difference in seconds between two times.

*Example 9-44 Calculation time differences in seconds using SQL*


---

```
D MyTime          S          T   inz(T'12.00.00')
```

D NbrOfSeconds	S	10I 0
----------------	---	-------

---

```
*-----
C/EXEC SQL
C+ Set :NbrOfSeconds =  Midnight_Seconds(Current_Time)
C+                   -  Midnight_Seconds(:MyTime)
C/End-EXEC
C   NbrOfSeconds  Dsply
```

---

C Return

---

**Calculating time differences using SQL scalar function `TIMESTAMPDIFF`**

Additional time differences can be calculated by using the SQL scalar function `TIMESTAMPDIFF`.

The scalar function `TIMESTAMPDIFF` has two arguments:

- ▶ The first argument must be a numeric value with zero decimal positions that describes the time difference to calculate:
  - 1 = fractions of a second
  - 2 = seconds
  - 4 = minutes
  - 8 = hours
  - 16 = days
  - 32 = weeks
  - 64 = months
  - 128 = quarters
  - 256 = years
- ▶ The second argument is the difference between two timestamps converted to a character string with a length of 22.

**Note:** `TIMESTAMPDIFF` can only be used for statistical purposes, because the following assumptions are used in estimating the difference:

- ▶ There are 365 days in a year.
- ▶ There are 30 days in a month.
- ▶ There are 24 hours in a day.
- ▶ There are 60 minutes in an hour.
- ▶ There are 60 seconds in a minute.

`TIMESTAMPDIFF` can only be used for statistical purposes, because the following assumptions are used in estimating the difference:

- ▶ There are 365 days in a year.
- ▶ There are 30 days in a month.

- ▶ There are 24 hours in a day.
- ▶ There are 60 minutes in an hour.
- ▶ There are 60 seconds in a minute.

Example 9-45 shows how the time difference in weeks between two timestamps can be calculated using the scalar function `TIMESTAMPDIFF`.

*Example 9-45 Calculating time differences using the SQL scalar function `TIMESTAMPDIFF`*

```

D MyTimeStamp      S              Z      inz(Z'2004-07-01-08.00.00.000000')
D MyResult         S              5I 0
*-----
C/EXEC SQL
C+ Set :MyResult = TimestampDiff(32,
C+                cast(current_timestamp - :MyTimeStamp as Char(22)))
C/End-EXEC
C   MyResult      Dsply
C
C                Return

```

### 9.3.7 Extracting a portion of a date, time, or timestamp

Sometimes it is necessary to determine a portion of a date, time, or timestamp.

While RPG offers two methods, SQL has a set of scalar functions to extract the particular portions of a date or timestamp.

#### Extracting a portion of a date, time, or timestamp with RPG

RPG provides two methods to extract a portion of a date, time, or timestamp.

- ▶ Operation code `EXTRCT`

The operation code `EXTRCT` can only be used in classical RPG. If you want to code free format RPG you have to use the built-in function `%SUBDT`.

In factor 2 the time, date, or timestamp must be specified, followed by a colon and the duration code (for valid duration codes look at Table 9-12 on page 211).

The results can be:

- For a date**                    Years, months, days
- For a time**                    Hours, minutes, seconds
- For a timestamp**            Years, months, days, hours, minutes, seconds, microseconds

**Note:** Duration code `*DAYS` always returns the day of the month, even when julian date format is used.

Example 9-46 shows how the end of month can be calculated. The day of the month is determined by using the operation code `EXTRCT`.

*Example 9-46 Calculating the last day of a month with RPG operation codes*

```

D MyDate          S              D      inz(D'2004-02-15')
D MyMonthEnd      S              D
D Days            S              3U 0
*-----
C
C                extrct  MyDate:*Days  Days
C   MyDate        SubDur  Days:*Days  MyMonthEnd
C                AddDur  1:*Months   MyMonthEnd

```

---

▶ **Built-in function %SUBDT(Substring Date)**

The built-in function %SUBDT is the alternate method to the operation code EXTRACT in free format RPG.

The built-in function %SUBDT has two arguments. In the first one, the date, time, or timestamp value must be specified. In the second one a compatible duration code must be entered.

The following example shows how the end of month can be calculated by using a built-in function. The day of the month is determined by using the built-in function %SUBDT.

*Example 9-47 Calculating the last day of a month using built-in functions*

---

```
D MyDate          S          D  inz(D'2004-02-15')
```

---

```
D MyMonthEnd      S          D
```

---

```
*-----
```

```
/Free
```

```
  MyMonthEnd = MyDate - %Days(%SubDt(MyDate: *Days)) + %Months(1);
```

```
  Dsply MyMonthEnd;
```

```
  Return;
```

```
/End-Free
```

---

### Extracting a portion of a date, time, or timestamp using SQL

In contrast to RPG, SQL provides a separate scalar function for every duration code:

- ▶ YEAR(date) returns the year's portion from a date or timestamp.
- ▶ MONTH(date) returns the month's portion from a date or timestamp.
- ▶ DAY(date) or DAYOFMONTH(date) returns the day's portion from a date or timestamp.
- ▶ HOUR(time) returns the hour's portion from a time or timestamp.
- ▶ MINUTE(time) returns the minute's portion from a time or timestamp.
- ▶ SECOND(time) returns the second's portion from a time or timestamp.
- ▶ MICROSECOND(timestamp) returns the microsecond's portion from a timestamp.

All scalar functions have one argument, where the date, time, or timestamp must be specified. The return value is always a numeric value without decimal positions, containing the number of durations.

**Note:** There is a difference between the scalar function DAY and the scalar function DAYS. DAY returns the day of the month, while DAYS returns the number of days since '0001-01-01'.

While the scalar functions DAY and DAYOFMONTH return the day of the month, the scalar function DAYOFYEAR can be used to return the current day of the year.

### 9.3.8 Additional SQL scalar functions for date calculation

SQL provides scalar functions to calculate the day of the week and also to calculate the week of the year.

## Calculating the day of the week

SQL provides three scalar functions to calculate the day of the week:

► **DAYOFWEEK(Date)**

The DAYOFWEEK function returns a numeric value between 1 and 7 that represents the day of the week for a date or timestamp, where 1 is Sunday and 7 is Saturday.

► **DAYOFWEEK\_ISO(Date)**

The DAYOFWEEK\_ISO function returns a numeric value between 1 and 7 that represents the day of the week for a date or timestamp, where 1 is Monday and 7 is Sunday.

► **DAYNAME(Date)**

This returns a mixed-case character string containing the name of the day (for example, Friday) for the day portion of the argument.

National language considerations: The name of the day that is returned is based on the language used for messages in the job. This name of the day is retrieved from message CPX9034 in message file QCPFMSG in library \*LIBL.

The following example shows the function CvtDateToText that edits a date, for example, Wednesday, 1st September 2004.

*Example 9-48 UDF to convert a date into a character representation with day and month name*

---

```
Create Function ITS04710/CvtDateToText (MyDate DATE )
  Returns Char(50)
  Language SQL
  Specific ITS04710/CvtDateToText
  Deterministic
  Contains SQL
  Returns NULL on NULL Input
  DisAllow PARALLEL
Return DayName(MyDate) concat ' ' concat
  Trim(Char(DayOfMonth(MyDate))) concat
  Case When DayOfMonth(MyDate) IN (1 , 21 , 31)
    Then 'st'
    When DayOfMonth(MyDate) IN (2 , 22)
    Then 'nd'
    When DayOfMonth(MyDate) = 3
    Then 'rd'
    else 'th'
  end concat ', ' concat
  MonthName(MyDate) concat ' ' concat
  Char(Year(MyDate)) ;

Comment on specific function ITS04710/CvtDateToText
  is 'Convert Date to Text' ;
```

---

## Calculating the week of the year

SQL provides two methods to determine the week of the year:

► **WEEK(date)**

The WEEK function returns a numeric value between 1 and 54 that represents the week of the year. The week starts with Sunday, and January 1 is always in the first week.

► **WEEK\_ISO**

The WEEK\_ISO function returns a numeric value between 1 and 53 that represents the week of the year. The week starts with Monday. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week containing January 4. Thus, it is

possible to have up to 3 days at the beginning of the year appear as the last week of the previous year or to have up to 3 days at the end of a year appear as the first week of the next year.

The following example shows how the order total cumulated per calendar week.

*Example 9-49 Generating sums on week level*

---

```
with a as (select case when      month(Order_Delivery)      = 12
                        and week_iso(Order_Delivery) = 1
                        then year(Order_Delivery) + 1
                        when      month(Order_Delivery)      = 1
                        and week_iso(Order_Delivery) >= 52
                        then year(Order_Delivery) - 1
                        else year(Order_Delivery)
                        end as DelYear,
            week_iso(Order_Delivery) as DelWeek,
            Order_Total
            from Order_Header)
select DelYear, DelWeek, sum(Order_Total)
       from a
       group by DelYear, DelWeek
       order by DelYear, DelWeek;
```

---



# Tools

In this section we cover the tools that can help us in this process of modernization. The iSeries Developer's Roadmap introduces modern development tools that can assist us in this process of modernization.

Archived

## DB2 Development Tools

As you modernize your business logic with SQL and DB2 UDB database definitions, you should also consider modernizing your development tools.

iSeries developers are familiar with tools such as Source Entry Utility (SEU), and the iSeries Developer Roadmap introduces modern development tools such as WebSphere Development Studio Client for iSeries. It not only supports new technology like Java, Web, or XML development, but also traditional RPG, COBOL, and so on.

In this chapter, we also introduce you to several interesting graphical tools such as:

- ▶ The graphical iSeries System debugger. This state-of-the-art debugger lets the developers debug programs that run on an iSeries server.
- ▶ DB2 Query Management Facility (QMF™). The graphical query tool, which can help the end users work with queries without knowledge of SQL syntax.
- ▶ iSeries Navigator.
- ▶ WebSphere Development Studio Client for iSeries (WDS*c*).

## 10.1 WebSphere Development Studio Client for iSeries (WDSC)

WebSphere Development Studio Client for iSeries is based on WebSphere Studio Site Developer (WSSD). The other version is WebSphere Development Studio Client for iSeries Advanced, which is based on WebSphere Studio Application Developer (WSAD).

The WebSphere Studio Workbench is the basis of all IBM WebSphere Studio products. It is based on the open-source Eclipse technology. The WSSD is the application development tool on top of the Workbench and the WSAD is expanded to support the Enterprise Java Beans (EJB). The core feature of the Eclipse user interface includes perspectives that are a collection of views and editor tools. A developer can use an appropriate editor to code, for example, Java, RPG, or SQL.

WebSphere Development Studio Client for iSeries (WDSC) and Remote System Explorer (RSE) provide a modern interface for iSeries application development and debugging. In addition, the WDSC has several wizards to help get started with the DB2 Web service and Extenders technology.

Figure 10-1 shows the RSE LPEX editor. This graphical editor includes color coding, syntax checking, and statement prompting. There is an Outline view that assists with program coding.

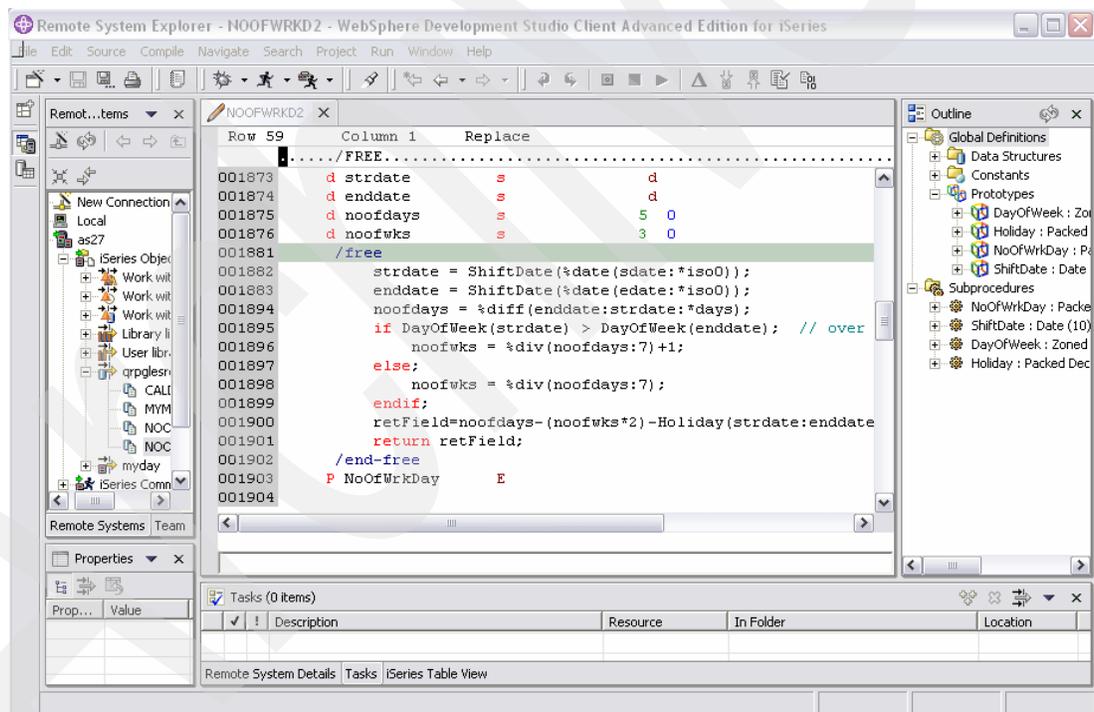


Figure 10-1 LPEX editor

For database development tasks, the Data perspective is part of the WSSD product, which includes views and editors for SQL development. You can connect to an iSeries server and import the database object definitions. There are several wizards to construct Select, Insert, Delete, and Update statements.

In the following example, we used WDCS Version 5.1.2 to demonstrate the connection to the iSeries server and how to build a SELECT statement task.

1. In the WDCS menu, we select **File** → **New** → **Other**, then the New window appears (Figure 10-2). The left pane shows the available perspectives and the right pane shows the corresponding wizards to the perspective. We select **Data perspective** and **SQL statement wizard** and click **Next**.

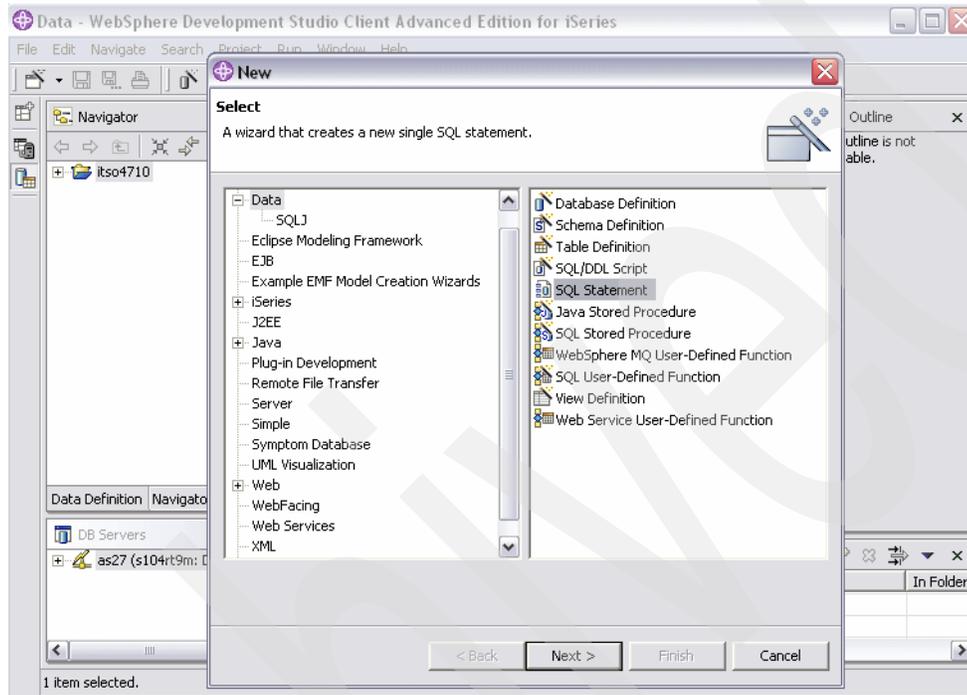


Figure 10-2 Data perspective to a new SQL statement

2. Figure 10-3 on page 228 shows the pull-down menu to select the SQL statement type. Since we need to connect to an iSeries server, we also select **Create the new database connection**. Click **Next**.

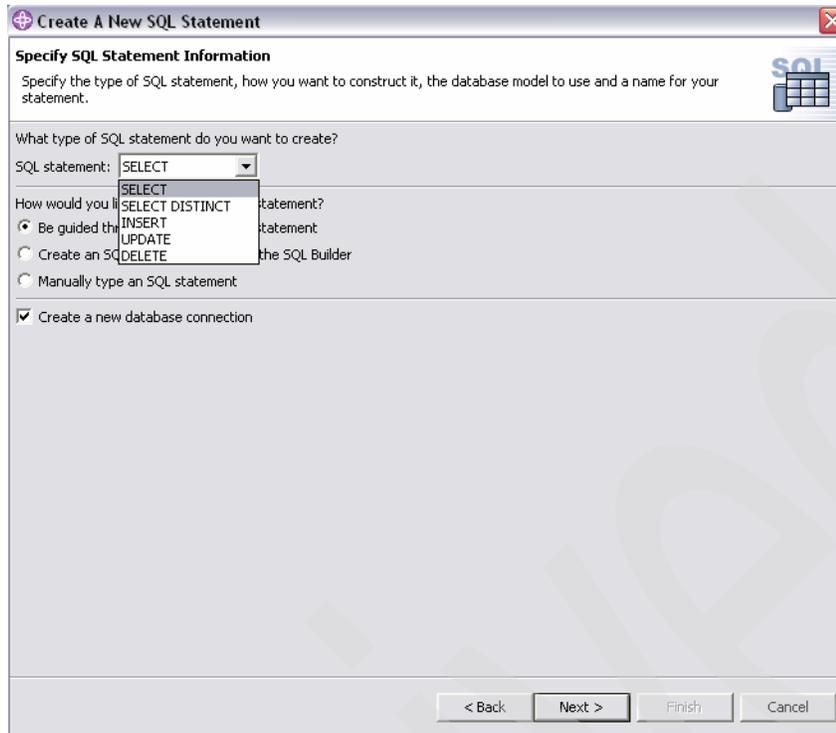


Figure 10-3 Select the SQL statement type

3. In the Database Connection window (Figure 10-4 on page 229), we specify the value of the connection name, database name, user ID, password, database vendor type, and host name. We select DB2 Universal Database for iSeries, V5R1 for the database vendor type. You can connect to the iSeries database now. However, all schemas and the objects in the schemas information on the iSeries server are gathered by the default. We can set the filters to include only the objects that we are interested in by clicking **Filters**.

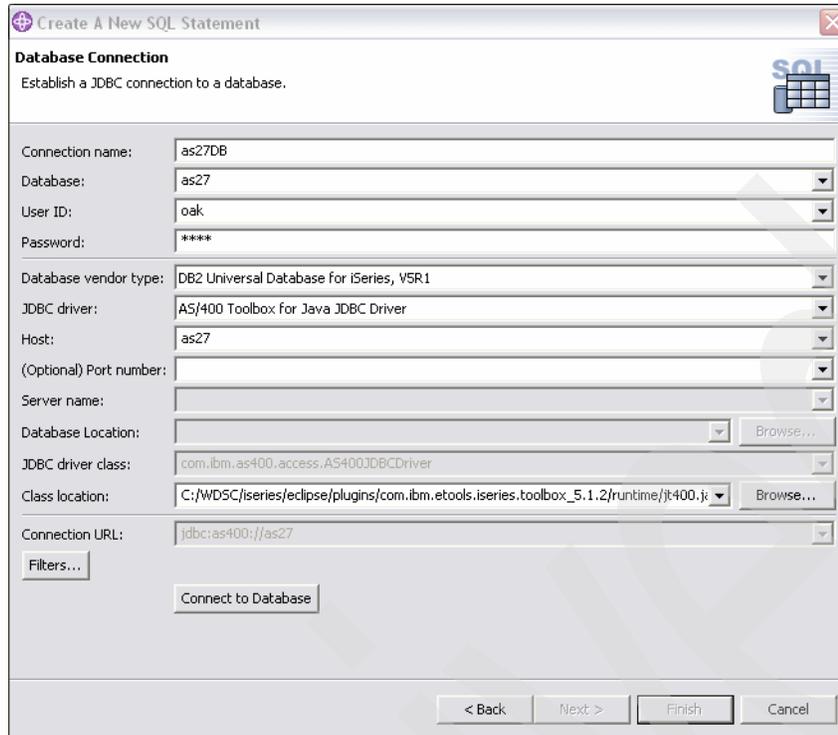


Figure 10-4 Database connection

4. When the Connection Filters window displays (Figure 10-5), we deselect **Exclude system schema** and specify a filter to include only ITSO4710 schema. Then click **Connect to Database** to collect the database information.

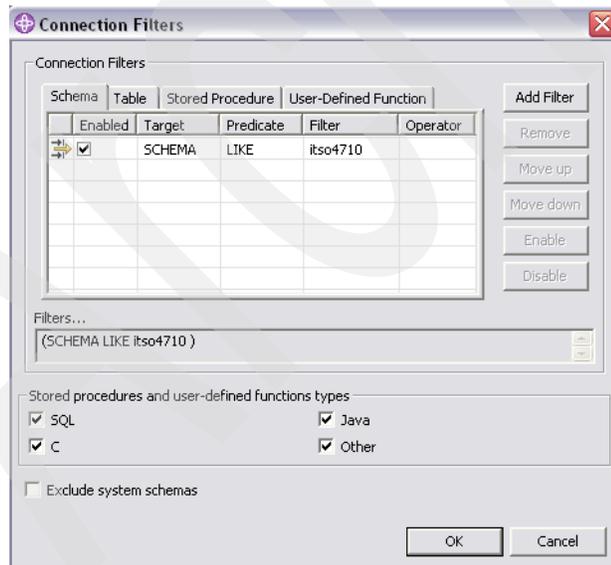


Figure 10-5 Connection filters

5. We specify the folder in which to keep the database information and also specify the SQL statement name, CUSTORDERSUMMARYBYNAME, then we click **Next**.

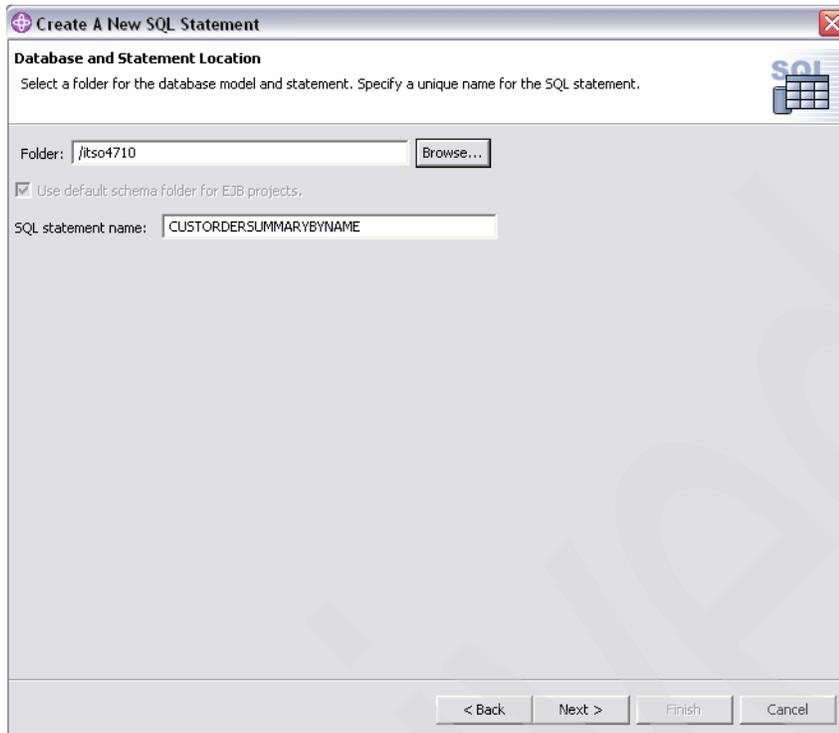


Figure 10-6 SQL statement name

6. The Construct an SQL Statement window appears and, as shown in Figure 10-7 on page 231, we use menu tabs to specify the following:
  - Tables names (ORDERHDR and CUSTOMER)
  - Join type (Inner join)
  - Join condition (ORDERHDR.CUSTOMER\_NUMBER = CUSTOMER.CUSTOMER\_NUMBER)
  - Group by and Order by conditions (CUSTOMER\_NAME)

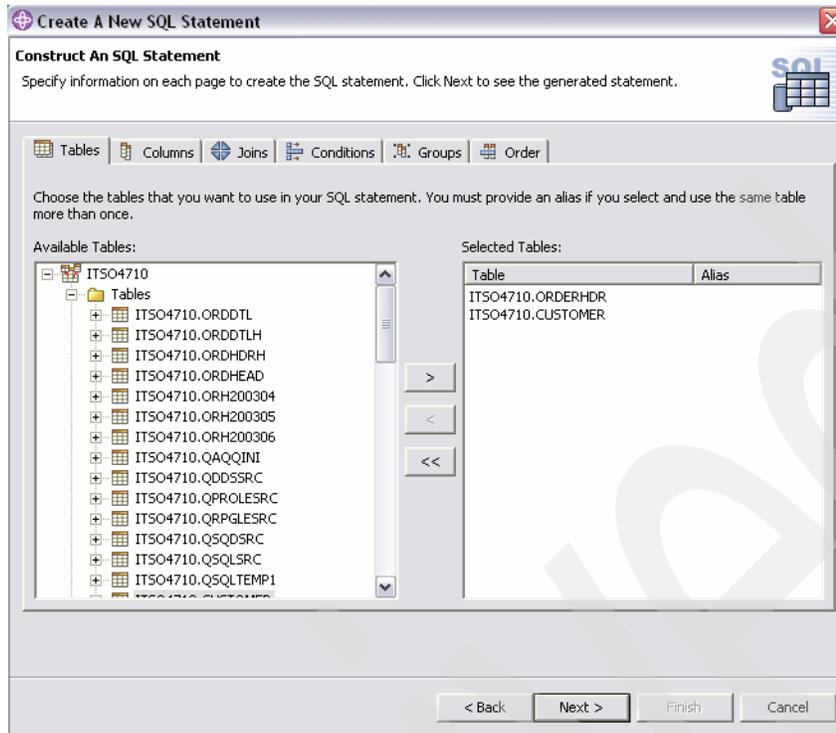


Figure 10-7 Construct an SQL statement

- Figure 10-8 shows the Create Join window, in which we select the source and target table, column, and join type.

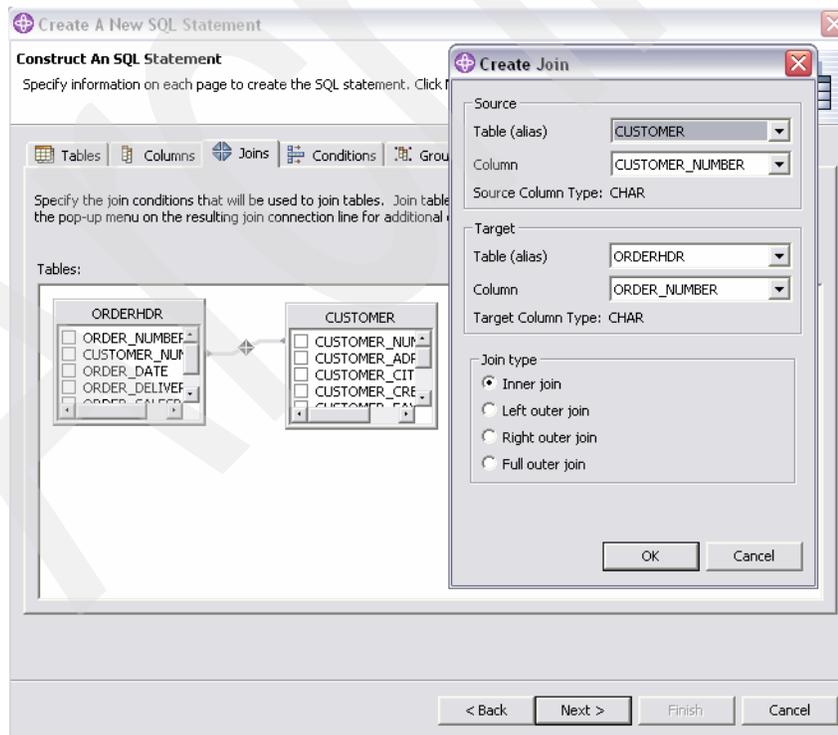


Figure 10-8 Join conditions

8. Click **Next**. The constructed SQL statement is shown (Figure 10-9). You can run the SQL statement by clicking **Execute**.

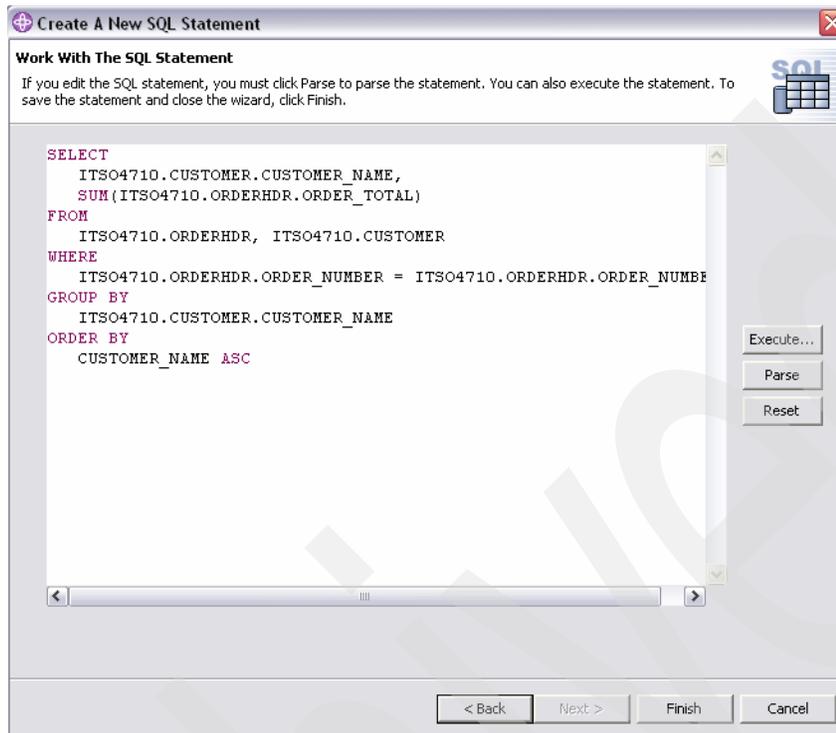


Figure 10-9 The SQL statement

9. Figure 10-10 on page 233 shows the result of the SQL. Click **Close** to close the Execute SQL statement window and then click **Finish**.



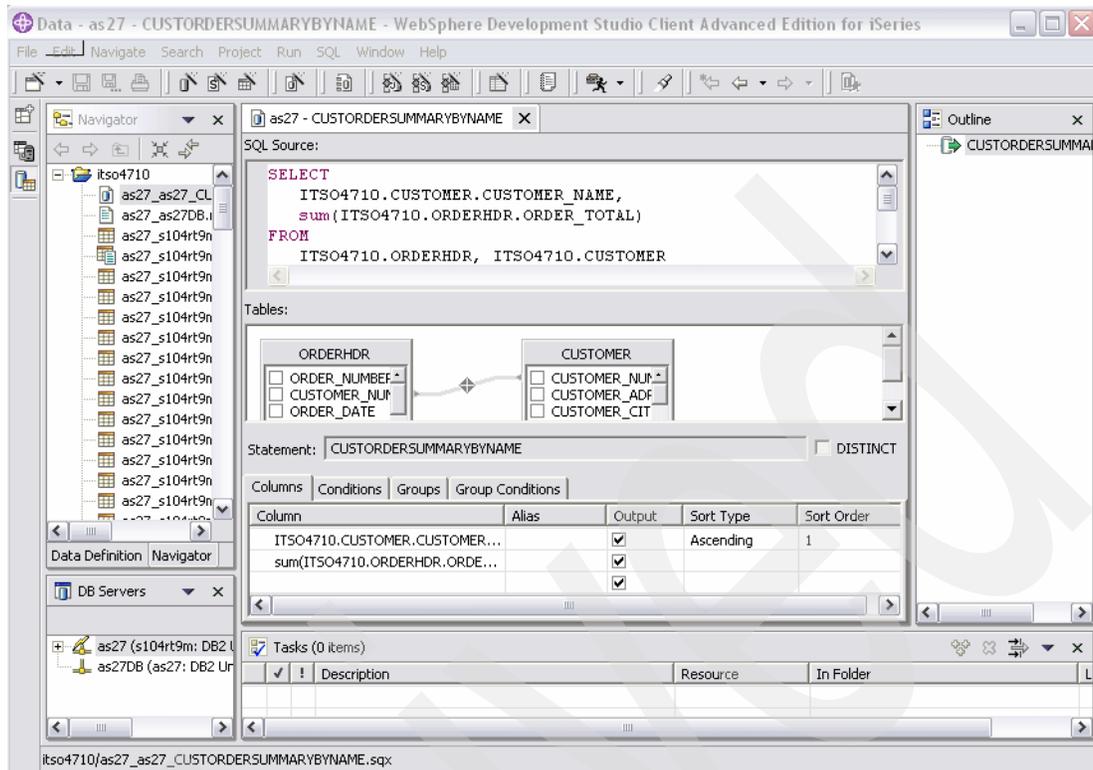


Figure 10-11 SQL Builder

In addition, the DB Servers view in the Data perspective also includes a tool that is likely to reverse engineer. This tool can generate the SQL DDL from the database information that was collected at the creation of the new database connection.

## 10.2 iSeries Navigator

iSeries Navigator is a graphical tool for database administration, and it can help in development activities. It includes the Databases component that is useful for SQL application development. The following are the features within the Databases component that benefit your development environment.

### 10.2.1 Database Navigator

Use Database Navigator to easily view and work with database objects and their related objects (for example, easily view indexes built over a table).

You can graphically work with DB2, including:

- ▶ Create work and administer database objects such as tables, views, alias, procedures, and so on. For example, the definition of the tables can be changed, such as add/remove columns.
- ▶ Define Referential Integrity rules/constraints.
- ▶ Reverse engineer SQL DDL from objects created with DDS interfaces.
- ▶ Manage database logging (journaling).
- ▶ Analyze SQL/Query job performance through Visual Explain.

The pop-up menu on the database objects in the navigator tree provides the editing and viewing of database objects.

## 10.2.2 Run SQL Scripts

The Run SQL Scripts allows you to execute SQL statements against the DB2 UDB for iSeries server and also view the results in the results window. You can store scripts of the SQL statements in files on PC or IFS. When you need to re-run those scripts, you simply open and run in this Run SQL Scripts.

SQL Assist has statement wizards to provide programmers with step-by-step processing of coding an SQL Select, Insert, Update, and Delete. This is especially useful for native programmers who are still learning SQL syntax. You can test your SQL statements to see the result and launch the Visual Explain to understand the optimizer's implementation of the query.

You can find an example of using this feature in the next section.

## 10.2.3 Visual Explain

Visual Explain provides a graphical way of identifying and analyzing database performance. It illustrates the decisions made by the query optimizer. It also recommends ways to improve query performance by building indexes or collecting columns statistics. Visual Explain is a very useful tool for database developers to analyze and tune the SQL performance.

Before we bring you to our Visual Explain example, we would like to introduce you to the Statistics Manager. It is good to know what the Statistics Manager is and how it relates to the column statistics.

### Statistics Manager

OS/400 is an object-based operating system. Tables and indexes are objects. Like all objects, information about the object's structure, size, and attributes is contained within the table and index objects. In addition, tables and indexes contain statistical information about the number of distinct values in a column and the distribution of those values in the table. The DB2 UDB for iSeries optimizer uses this information to determine how to best access the requested data for a given query request.

Starting with V5R2 of OS/400, DB2 UDB for iSeries has a new SQL query engine (SQE). As part of this new SQL query engine, a statistics manager component is responsible for generating, maintaining, and providing statistics to the SQE optimizer. As mentioned earlier, sources for statistics within DB2 UDB for iSeries come from default values and/or indexes. With SQE, the optimizer has another source, namely *column statistics* stored within the table object.

The column statistics will be generated by a low-priority background job, with the goal of having the column statistics available for future executions of this query. This automatic collection of statistics allows the SQE optimizer to benefit from columns statistics, without requiring an administrator to be responsible for the collection and management of statistics, as is true for other RDBMS products. Even though it is not required, statistics can also be manually requested for iSeries users who want to take on the task of statistics collection and management, without waiting for the statistics manager to recognize the need for a column statistic. The column statistics that are generated are only used by the new SQL query engine. The original Classic Query Engine (CQE) continues to use only default values and indexes for statistics.

Comparing an index (used for statistics) to column statistics, you will find the storage required for column stat is much smaller. On average, column statistics take up approximately 8K–12K per column. This additional space will be reflected in the table’s size. As the data changes in the table, column statistics are not maintained, and thus do not affect the I/O performance on the table. On the other hand, if the data in the table is changing a lot, then the statistics can become stale or outdated. Indexes are maintained immediately as the data in the table changes. This allows the indexes to always provide up-to-date statistics. The cost of this maintenance might be reflected in slightly longer insert, update, and delete times.

The automatic statistic collection is controlled by the system value QDBFSTCCOL. This system value is set to \*ALL by default. This value allows all requests for background statistics collections, whether initiated by the system or initiated by the user.

The following example shows you how to execute the SQL request and use Visual Explain in Databases feature of iSeries Navigator to collect the column statistics.

1. Click **Run an SQL script** in the Databases tasks pane (Figure 10-12).

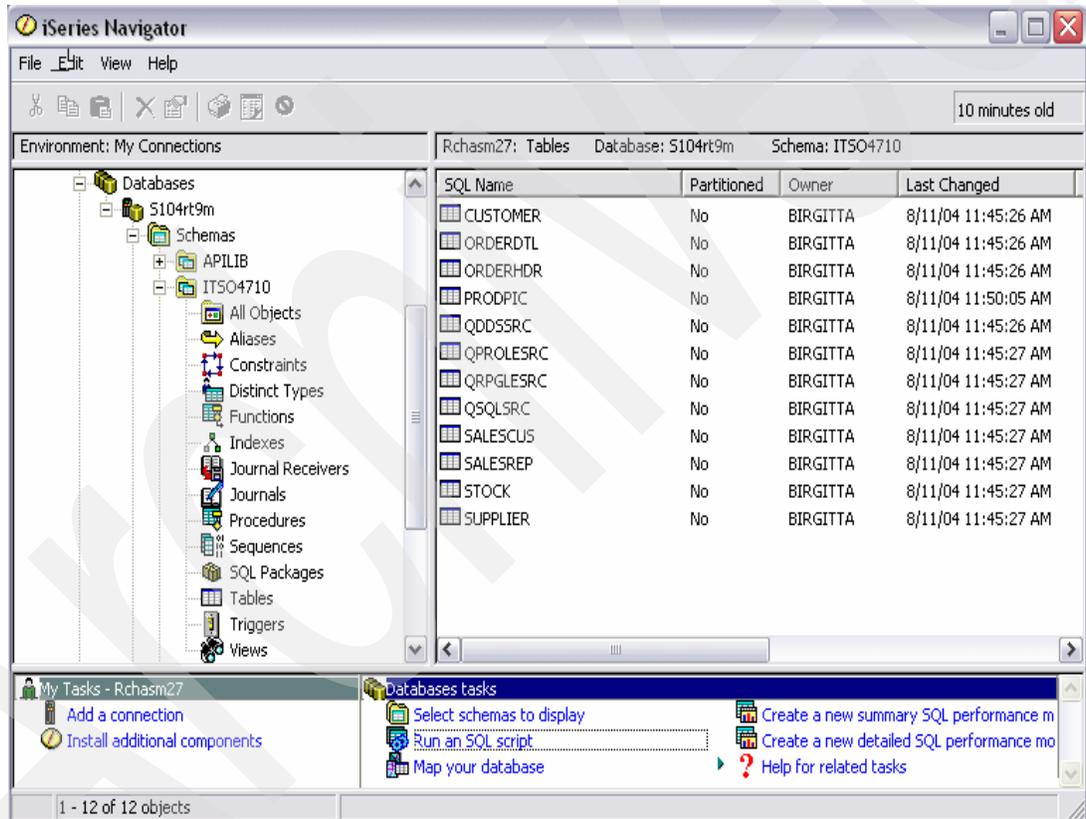


Figure 10-12 Select Run an SQL script

2. In the Run SQL Script window, as shown in Figure 10-13 on page 237, we enter the SQL statement in the text pane. For our example, we select the rows from the customer table that have a credit limit over 100000.

```
select * from its04710.customer where cuscrd > 100000
```

Then select **VisualExplain** → **Run and Explain**.

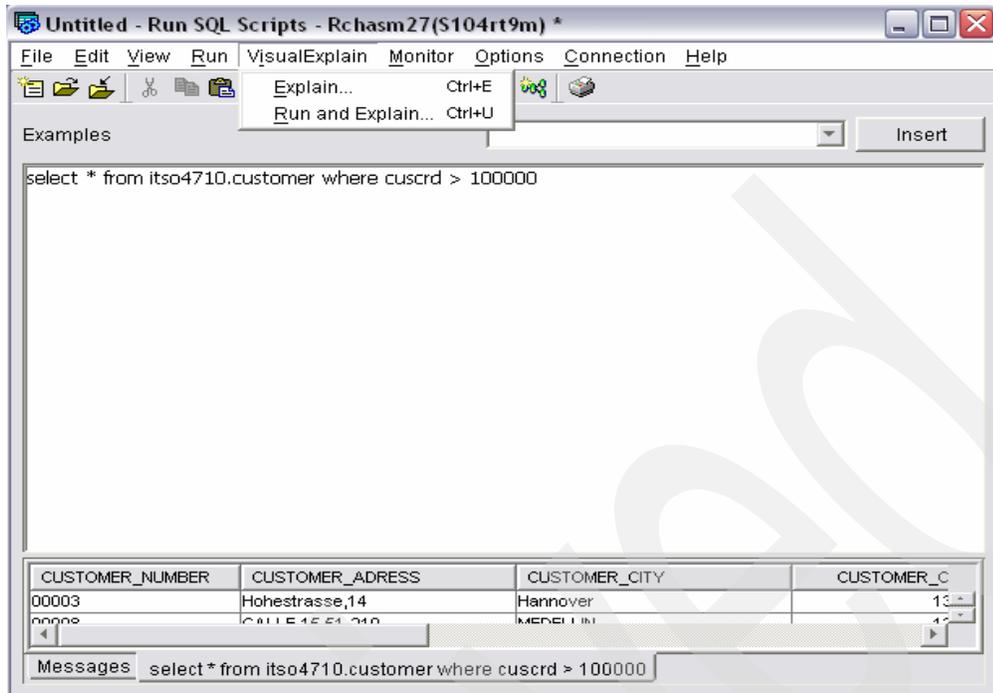


Figure 10-13 Select VisualExplain menu

3. In the Visual Explain window (Figure 10-14), select **Actions** → **Advisor** to see the Statistics and Index Advisor information.

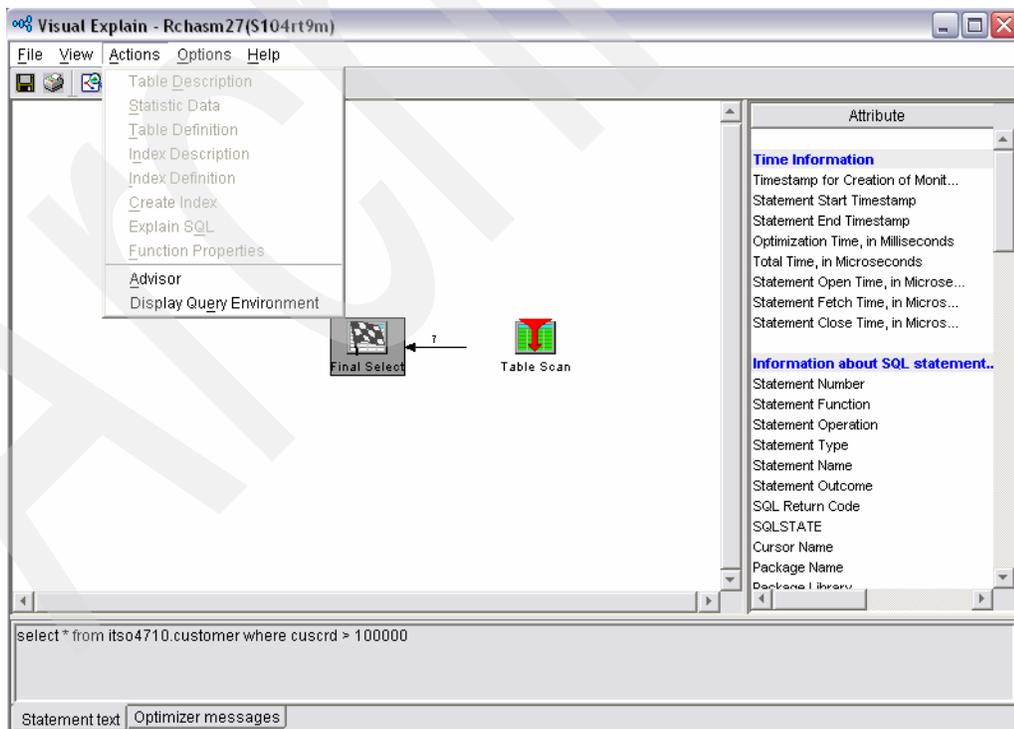


Figure 10-14 Visual Explain

- In the Statistics and Index Advisor window (Figure 10-15), you see the recommended column that will be needed to be collect the statistics. You can choose to collect either immediately or in the background.

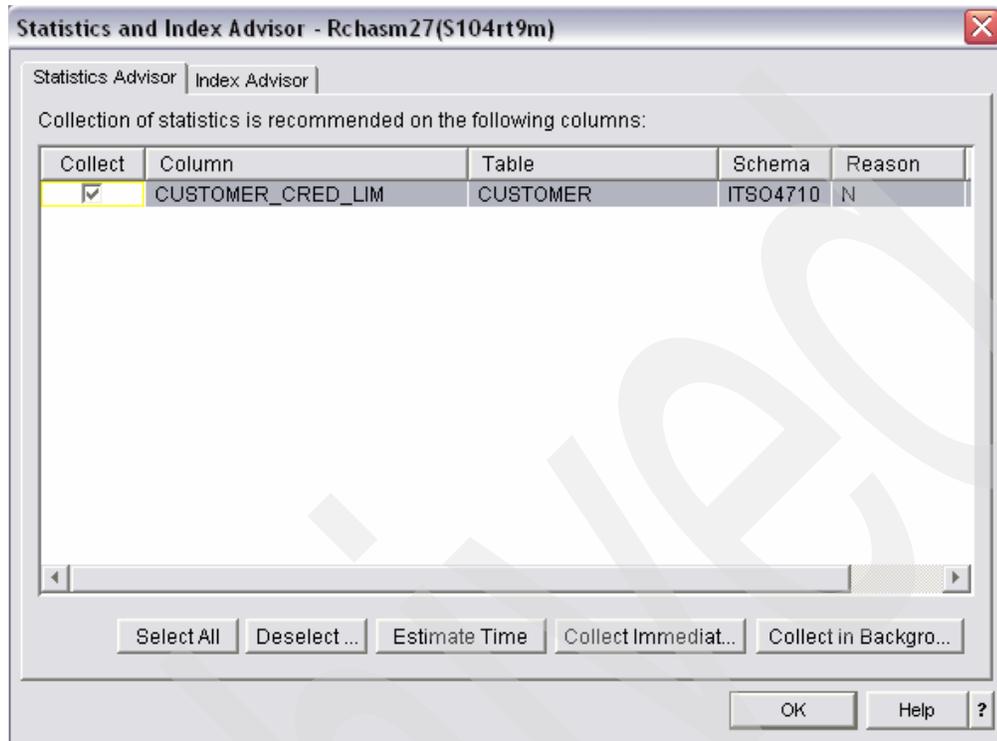


Figure 10-15 Statistics and Index Advisor window

For more information on Visual Explain refer to the redbook *DB2 Universal Database for iSeries Administration The Graphical Way on V5R3*, SG24-6092.

## 10.2.4 Graphical iSeries System Debugger

In iSeries Navigator V5R3, the IBM Toolbox for Java includes a graphical debugger that is a good tool to use when debugging stored procedures, functions, and triggers. It also supports the ILE languages, Java, and Original Program Model (OPM) RPG, and COBOL programs. The debugger includes an integrated call stack window, breakpoint groups, variable monitors, and a local variables display.

Here is an example of the steps to run the graphical debugger for an SQL stored procedure.

- Create the SQL procedure with the \*SOURCE debug view, as shown in Figure 10-16 on page 239. The SQL procedure can be created by:
  - An iSeries Navigator Run SQL Script session. The SET Option statement is used in the procedure to cause the SQL source-level debug view to be created during creation of the stored procedure.
  - The RUNSQLSTM command. The procedure source code can also be stored in a source physical file member and used to create the SQL procedure with the source-level debug view. Here is a sample RUNSQLSTM command:

```
RUNSQLSTM SRCFILE(MYLIB/MYSRC) SRCMBR(SHIP_IT) COMMIT(*NONE) NAMING(*SQL)
DBGVIEW(*SOURCE)
```

```

CREATE PROCEDURE myschema.ship_it(IN ordnum INTEGER, IN ordtype CHAR(1),
    IN ordweight dec(3,2))
LANGUAGE SQL
SET OPTION DBGVIEW =*SOURCE
sp: BEGIN
DECLARE ratecalc DECIMAL(5,2);
/* Check for international order */
IF ordtype='I' THEN
    SET ratecalc = ordweight * 5.50;
    INSERT INTO myschema.wshipment VALUES(ordnum,ordweight,ratecalc);
ELSE
    SET ratecalc = ordweight * 1.75;
    INSERT INTO myschema.shipments values(ordnum,ordweight,ratecalc);
END IF;
END

```

Figure 10-16 SQL procedure sample

- An iSeries Navigator Run SQL Script session needs to be active to start the debug mode. A debug session is initiated by going to the **Run** pull-down menu and selecting the **Debugger** task, as shown in Figure 10-17.

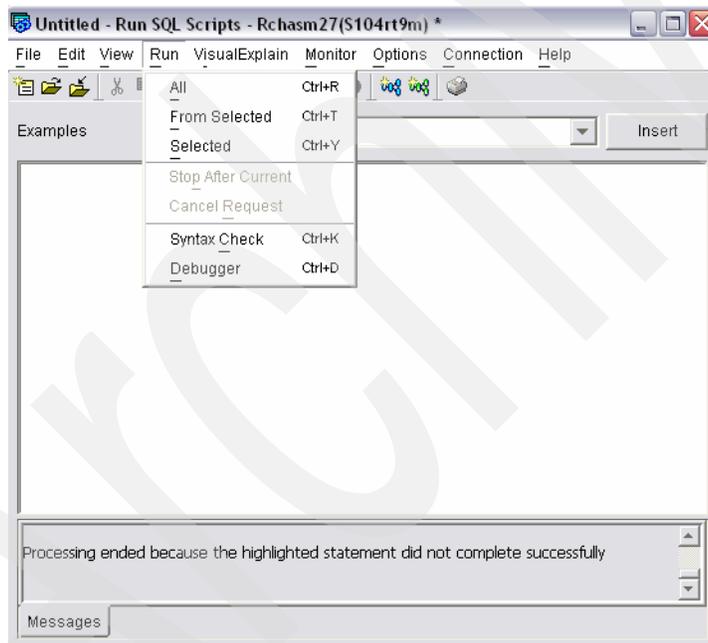


Figure 10-17 Start debugger

- The Debugger task will cause the Start Debug dialog window to be started on the client, as shown in Figure 10-18 on page 240.



Figure 10-18 Start Debug window

iSeries Navigator will automatically fill in the job information for the server job that will be used by the Run SQL Script session for execution and debug of the stored procedure.

The user will need to fill in the library and program name of the procedure. In this example, the library name is MYSCHEMA and procedure name is SHIP\_IT. When the procedure name is 10 characters or less, than the program and procedure name will be the same. If the procedure name is longer than 10 characters (for example, testprocedure), you need to look up the program name associated with the stored procedure. The SPECIFIC clause can be used to control the program name for a stored procedure with a long name.

4. Selecting the OK button causes the iSeries Graphical Debugger to be started on the client and load the SQL source-level debug view for the SHIP\_IT stored procedure. You can set a breakpoint by left-clicking Line 5 (IF ORDTYPE='I'). An enabled breakpoint is indicated by the red arrow, as shown in Figure 10-19 on page 241. Now that a breakpoint has been set, click the green resume arrow on the tool bar.

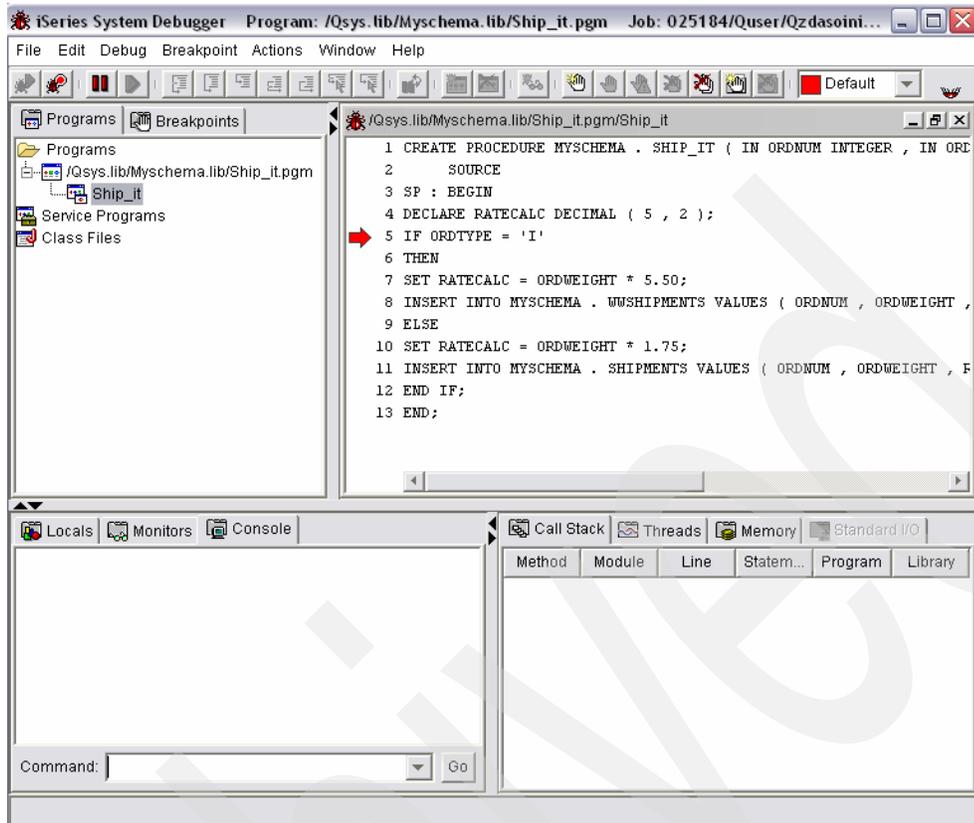


Figure 10-19 Set the breakpoint

- Return to your iSeries Navigator Run SQL Script session and issue the following SQL CALL statement:

```
CALL MYSCHEMA.SHIP_IT(33, 'I', 5.1)
```

The Debug Client will then take control at the breakpoint specified in the previous step. Figure 10-20 on page 242 shows how the yellow highlighting is used to indicate where execution was stopped for the breakpoint.

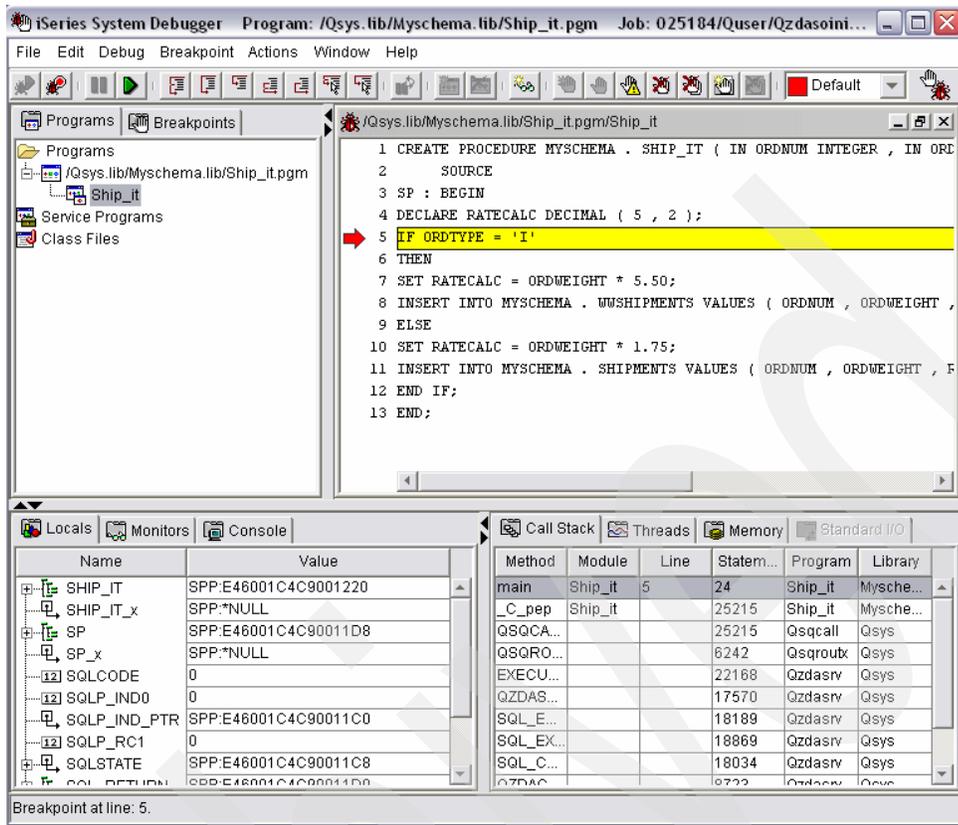


Figure 10-20 Execution stops at the breakpoint

- To view the contents of the ORDTYPE input parameter to determine which branch of the IF statement will be executed, left-click the **Console** tab in the lower left-hand corner and enter the following EVAL statement in the command window:

```
EVAL *SHIP_IT.ORDTYPE.
```

The content of this variable is then displayed in the Console window, as shown in Figure 10-21. All of the procedure parameter values can be displayed by just entering:

```
EVAL SHIP_IT
```

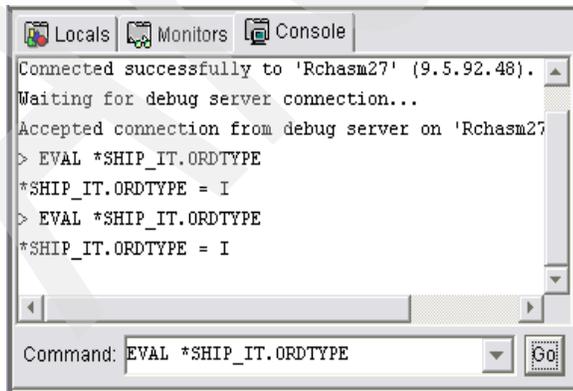


Figure 10-21 Debugger console

- To view the calculated shipping rate prior to the stored procedure ending, right-click Line 11 and select the **Run to Cursor** task. This will allow the debugger to execute all of the

code up to Line 11. When Line 11 is reached, the following command can be issued in the Console window to display the computed shipping rate:

```
EVAL SP:RATECALC
```

8. To complete execution of the SHIP\_IT stored procedure, click the green resume arrow on the tool bar.

For more information on the Graphical iSeries System Debugger, look at the debugger tool help by pressing F1. The iSeries InfoCenter also contains information on the debugger, just search the InfoCenter Web site:

<http://www.ibm.com/eserver/series/infocenter> for “graphical debugger”

For more information, see the white paper, *Graphical Debugger makes Procedural SQL Debug Even Easier*, at:

[http://www.ibm.com/servers/enable/site/education/abstracts/sqldebug\\_abs.html](http://www.ibm.com/servers/enable/site/education/abstracts/sqldebug_abs.html)

## 10.3 OS/400 utilities

There are some additional tools that OS/400 provides. We briefly cover them in this section.

### **DB2 SMP (Symmetric Multiprocessing)**

DB2 SMP is a licensed program that can also be used to boost the performance of SQL applications. By activating DB2 SMP, the query optimizer will try to speed the performance of the SQL SELECT operation by enabling a single operation to be split in multiple operations running across multiple CPUs concurrently.

### **PRTSQLINF**

PRTSQLINF is a CL command to print the information, which includes the SQL statements and the access plans used during the execution of the statements. For SQL embedded in a program and package objects, this command extracts the optimizer access method information out of the objects and places that information in a spooled file. The spooled file contents can then be analyzed to determine if any changes are needed to improve performance.

### **STRDBMON**

STRDBMON is a CL command that is used to collect the database performance data for a specified job or all jobs on the iSeries server. The Database Monitor gathers query execution statistics from the iSeries server jobs and records them in a database file. This database file is then analyzed to provide performance information to help tune the query or the database. You can use Visual Explain to analyze the collected performance data and give the recommendations.

## 10.4 DB2 Development Center

The DB2 Development Center (formerly known as the DB2 Stored Procedure Builder) is a rapid iterative development environment for developing stored procedures, user defined functions, and much more. This client-based development tool supports the entire DB2 UDB family of servers, therefore, it is an especially useful tool if you are developing and deploying procedures on multiple DB2 UDB servers.

As of January 2004, DB2 Development Center Version 8.1 only supports SQL stored procedure development when used with DB2 UDB for iSeries. Even though DB2 UDB for iSeries supports user defined function and Java stored procedures, the DB2 Development Center does not support the development of these objects on iSeries servers (support is planned for a future release of DB2 Development Center). Despite the current limitations, iSeries programmers will find the DB2 Development Center a useful tool for creating, modifying, and testing SQL stored procedures.

To access the DB2 Development Center, load the DB2 Application Development client, which is part of the DB2 Personal Developer's Edition and can be downloaded from IBM's Web site:

<http://www14.software.ibm.com/webapp/download/search.jsp?rs=db2pde>

## 10.5 DB2 Query Management Facility

Query Management Facility (QMF) for Windows is a query and reporting tool set for IBM's DB2 relational database management system. The Administrator function is used to define the connection to the DB2 UDB server such as iSeries Server V5.1 or later. The end user can use a graphical version of the query to build the queries without SQL knowledge by using the diagram or prompted interfaces. In addition, the QMF companion product has the conversion utilities available to convert Query/400 or Query Manager/400 into corresponding QMF for Windows objects. You can modernize the existing query reports instead of recreating them from scratch.

### 10.5.1 Migrating existing queries

In the following example, we demonstrate how to convert a query and a query form for iSeries to QMF for Windows objects. We also use QMF for Windows V8.1 to run a query with a query form from those objects.

We have \*QMQRV object, MYQRY, and \*QMFORM object, MYFRM, on the iSeries server. We use MYQRY and MYFRM to create a report summarizing the order amount and sorted by customer name, as shown in Figure 10-22.

Customer name	Order amount
AMPARO RESTREPO	2,445
Deepak	7,050
FEDERIC SMITH	5,100
GLORIA BEDOYA	6,150
HERNANDO BEDOYA	5,250
John F. American	3,300
KRIS PETERSON	3,795
Otto W German	4,950
Pai	3,495
Panky	3,300
Ramiro Zuniga	4,200
Sameer	9,450
SANDRA VELEZ	2,850
SULA	2,250
	63,585

Figure 10-22 Report created by query objects on iSeries server

The steps are:

1. We use the commands RTVQMQRYP and RTVQMFORM to export our query objects, MYQRY and MYFRM, respectively. We also specify the source file that is used to receive the query definitions as a source member.

Here is an example of how to export our query manager object and its source (Figure 10-23).

```
RTVQMQRYP QMQRYP(MYQRY) SRCFILE(QMQRYSRC)
```

```
H QM4 05 Q 01 E V W E R 01 03 04/08/31 17:57
V 1001 050
SELECT
-- Columns
      B.CUSTOMER_NAME, sum(a.order_total) AS TOTAL_ORDER_AMT
-- Tables
      FROM "ITS04710"/"ORDERHDR" A,
          "ITS04710"/"CUSTOMER" B
-- Join Conditions
      WHERE (A.CUSTOMER_NUMBER = B.CUSTOMER_NUMBER)
-- Summary Columns
      GROUP BY B.CUSTOMER_NAME
-- Sort Columns
      ORDER BY B.CUSTOMER_NAME
```

Figure 10-23 Query Management Query source

Since Query Manager/400 uses the system naming convention, which does not comply with QMF, we have to change the SQL statements to use the SQL naming convention instead. You can choose to change either in iSeries before transferring to PC or in QMF for Windows.

```
FROM "ITS04710"."ORDERHDR" A,
     "ITS04710"."CUSTOMER" B
```

The following shows how export a query form object and its source (Figure 10-24 on page 246).

```
RTVQMFORM QMQRYP(MYFRM) SRCFILE(QMFORMSRC)
```

```

H QM4 05 F 01 E V W E R 01 03 04/09/01 13:43
V 1001 050
T 1110 002 005 1114 007 1115 006 1117 005 1118 003 1113 062
R      2      C      1 Customer name
R SUM   2      K0   2 Order amount
V 1201 001 0
V 1202 001 2
T 1210 004 003 1212 004 1213 006 1214 055
R 1  LEFT  &DATE &TIME
R 1  RIGHT Page no. &PAGE
R 2  CENTER My company
R 3  CENTER Summary order amount by customer name
V 1301 001 2
V 1302 001 0
V 1401 002 NO
V 1402 004 1
V 1403 006 0
V 1501 001 1
V 1502 003 YES
V 1505 003 YES
V 1503 003 YES
V 1508 003 YES
V 1507 003 YES
V 1510 003 YES
V 3080 001 1
V 3101 002 NO
V 3102 002 NO
V 3103 001 0
....
....
....
V 3203 006 0
V 3204 001 1

```

Figure 10-24 Query management form source

2. Transfer the Query/400 definition source members to PC. You can use either File Transfer Protocol (FTP) or the iSeries Access Data transfer function. In our example we store the files, MYQRY.QRY and MYFRM.FRM, in the directory c:\temp.

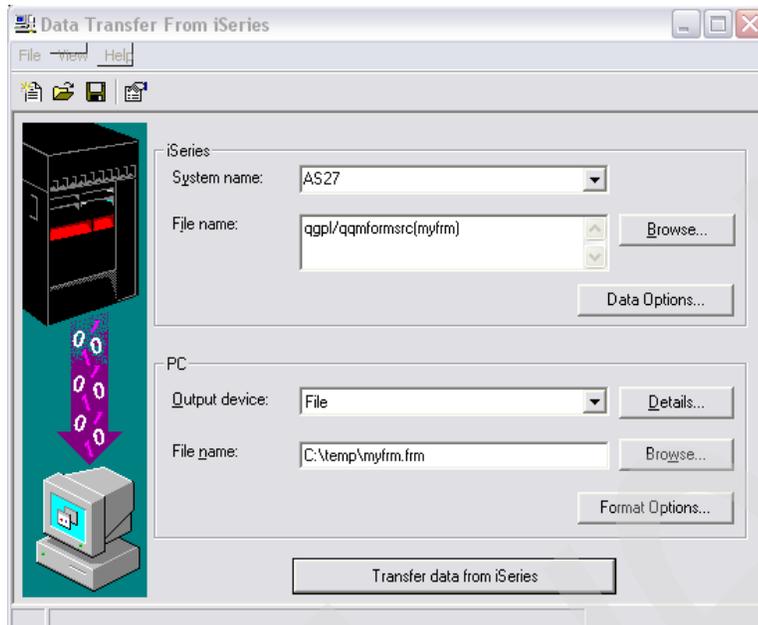


Figure 10-25 Transfer query definition source member to PC

- Use the QMF conversion utility to convert the Query/400 definition source members to the files that can be imported and saved as QMF objects by QMF for Windows. These files will be placed in the target directory QM400 and have a prefix of *QMF\_* added to the name of the original files.

```
C:\Program Files\QM400>qm4_qmf c:\temp\myqry.qry
C:\Program Files\QM400>qm4_qmf c:\temp\myfrm.frm
```

The converted files are QMF\_MYQRY.QRY and QMF\_MYFRM.FRM.

- In the QMF for Windows, select **File** → **Open** to open the converted files. Click **Run Query** in the Query menu and choose **Data Source** as **From open document** in the Form menu. Figure 10-26 on page 248 shows the query result the same as running on iSeries server.

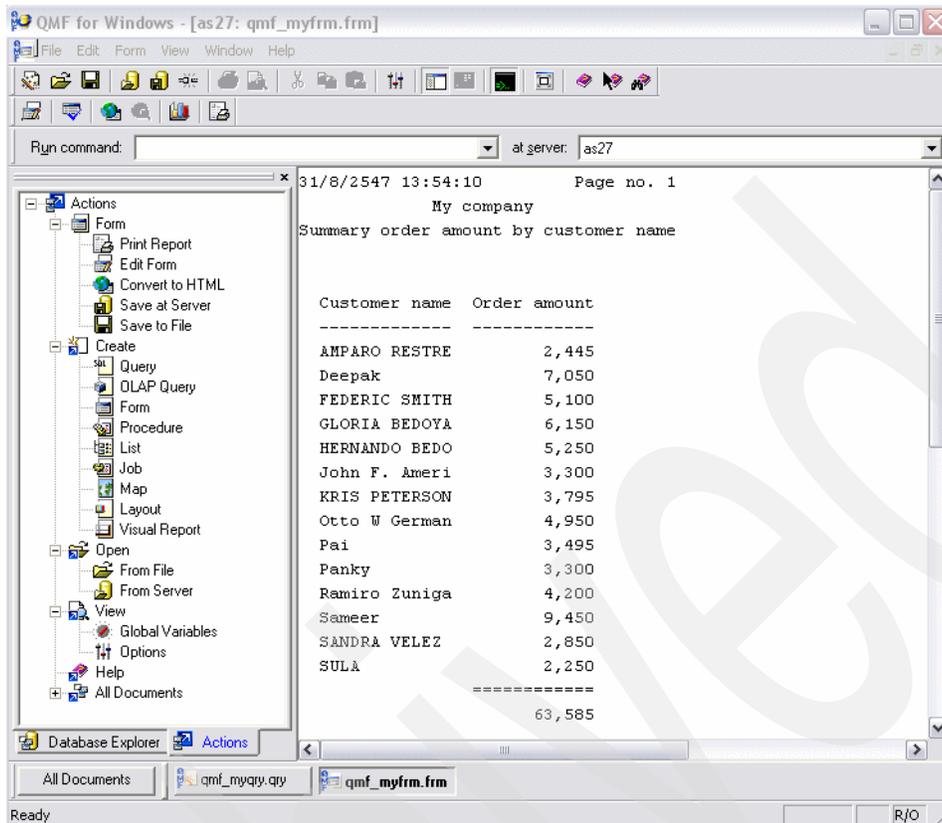


Figure 10-26 QMF for Windows and the query result

The files can be saved as QMF objects, and the end users use the graphical QMF for Windows to continually work with their queries.

## 10.5.2 Creating new queries

This example shows you how to use QMF for Windows V8.1 to create a query definition to summarize the order and sort by customer name. The result is the same as the example shown in the previous section.

1. In the QMF for Windows we create a new query definition by selecting **File** → **New** → **Query**. There are three interfaces to create a query: SQL, Prompted, and Diagram. Prompted is an easy way especially for novice users. Select **View** → **Prompted**. A window, as shown in Figure 10-27 on page 249, will appear.

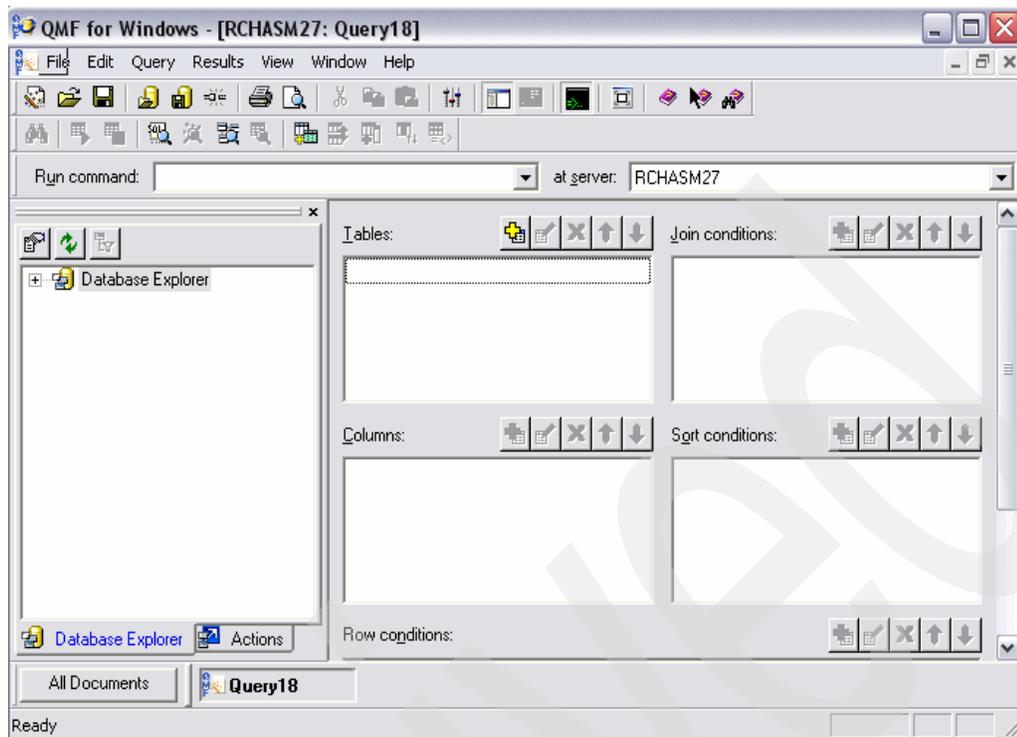


Figure 10-27 QMF for Windows

2. Select **Query** → **Add** → **Table**. The Tables panel, as shown in Figure 10-28 on page 250, appears for entering the schema/library in the Table owner field, and the table name in the Table name field. You can choose to add a table from the list by entering selection criteria (in our example, OR%), and click **Add From List**.

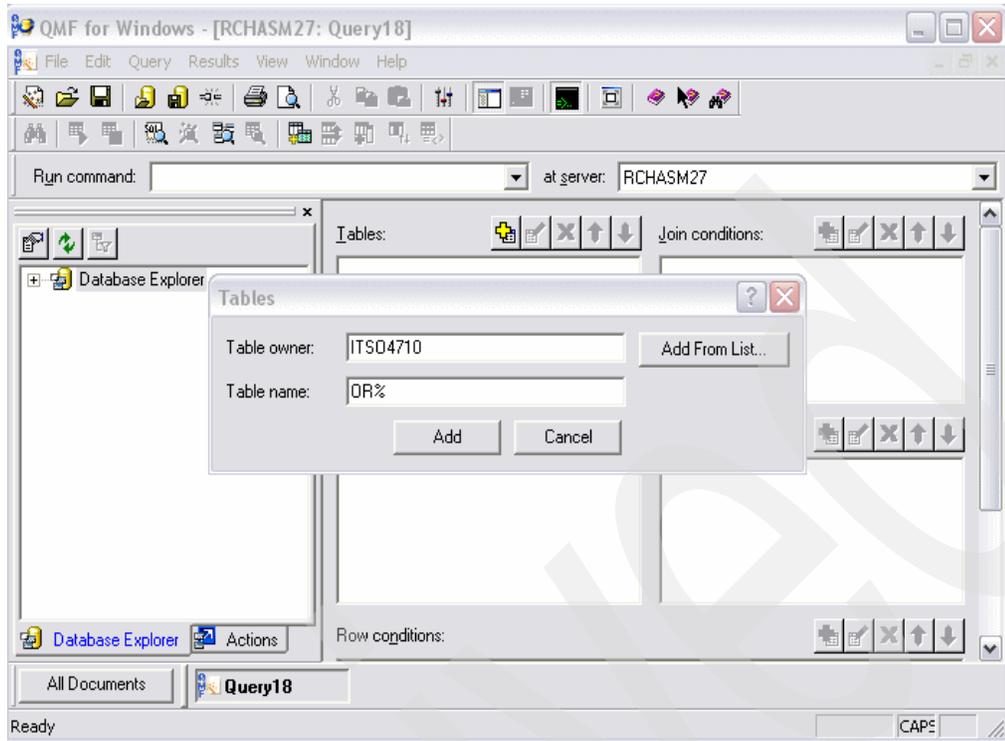


Figure 10-28 Add Table window

3. We select the ORDER\_HEADER file and click **Add**, as shown in Figure 10-29.

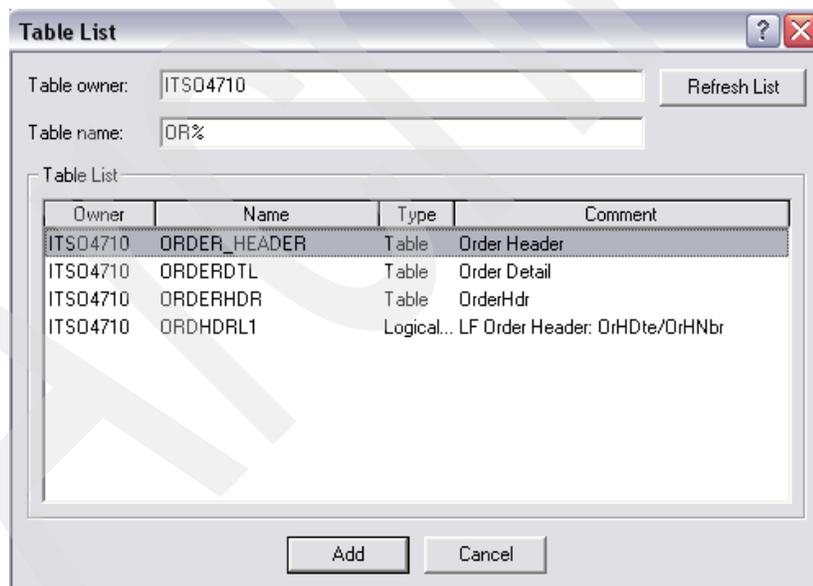


Figure 10-29 Select table

- Repeat steps 2 on page 249 and 3 to select the CUSTOMER file.
- To join the tables, select **Query** → **Add** → **Join Condition**, and select the type of join. We select the Inner join, as shown in Figure 10-30 on page 251. Then click **Continue**.

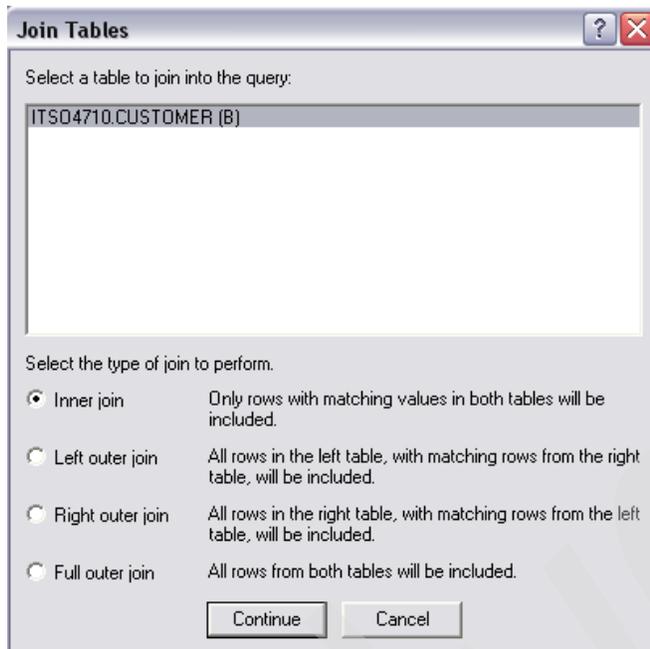


Figure 10-30 Join Tables

6. Select the join columns, as shown in Figure 10-31, select CUSTOMER\_NUMBER, and click **Add**.

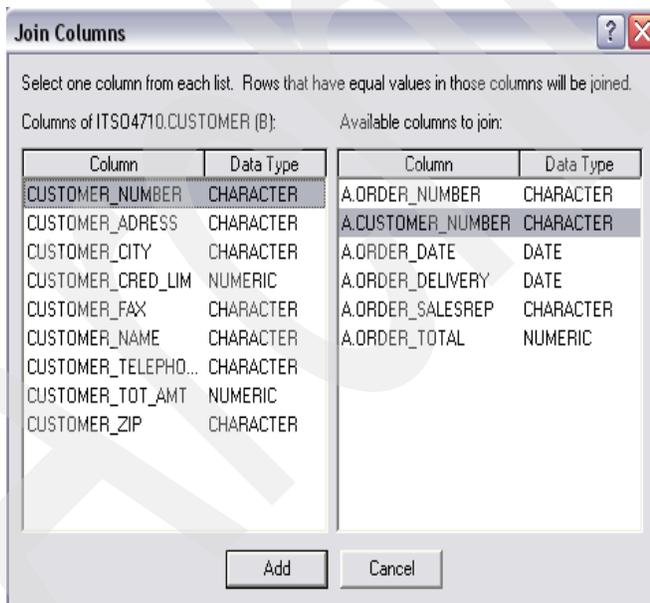


Figure 10-31 Select join columns

7. We select the result columns by selecting **Query** → **Add** → **Column**. In the Columns window, as shown in Figure 10-32 on page 252, we can also add the summary function and change the column name of the selected column. In our example, we select the CUSTOMER\_NAME and the ORDER\_TOTAL column. We specify the SUM function, and also change the column name of ORDER\_TOTAL.

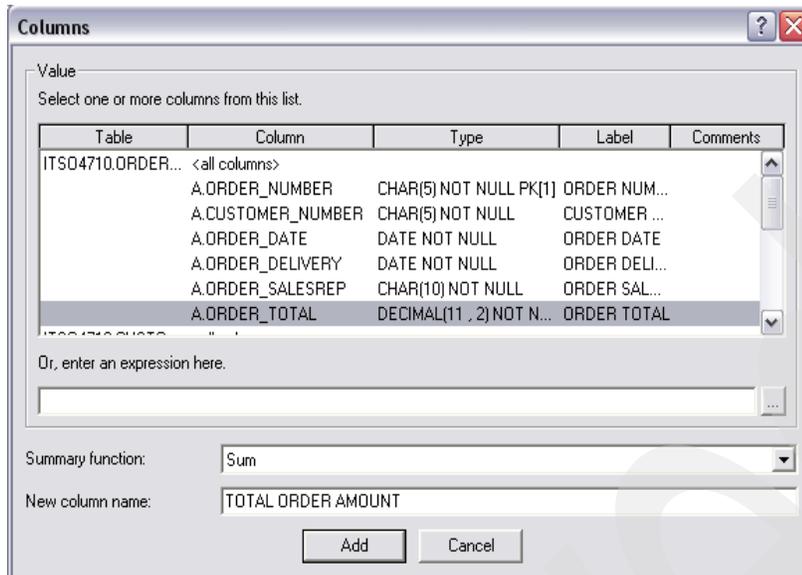


Figure 10-32 Add columns

- For sorting by customer name, select **Query** → **Add** → **Sort condition**, and select the CUSTOMER\_NAME column, as shown in Figure 10-33.

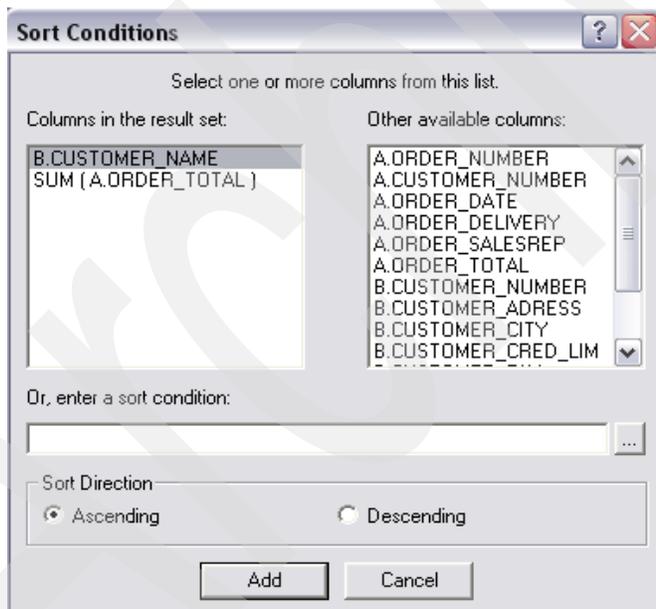


Figure 10-33 Sort Conditions

- Figure 10-34 on page 253 shows the complete SQL prompted for this query.

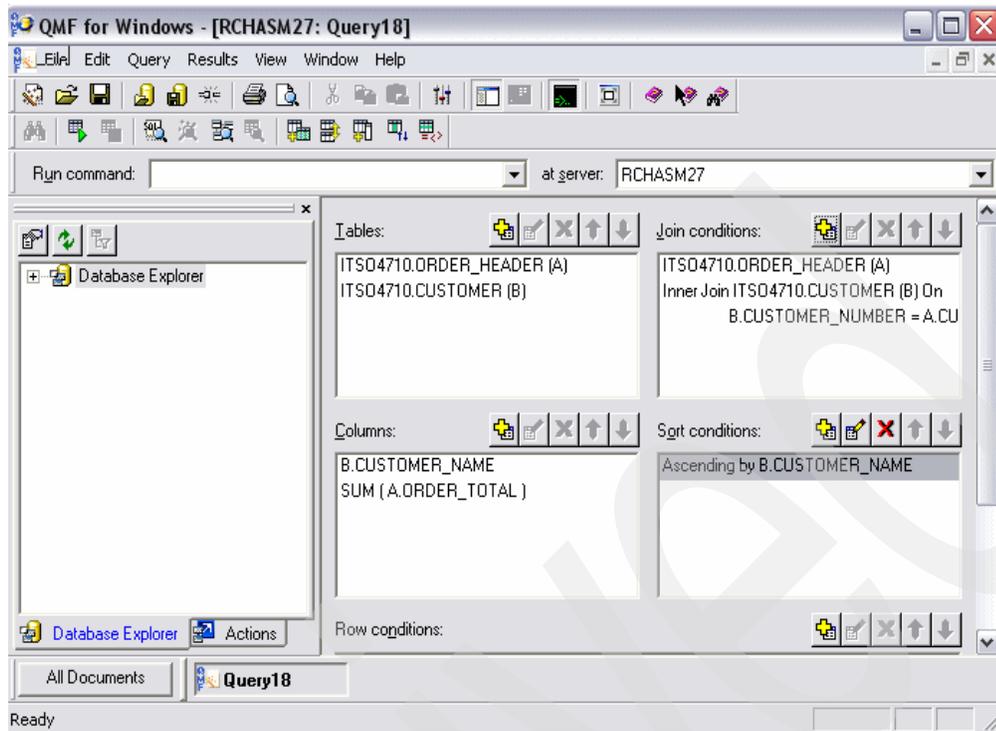


Figure 10-34 Complete SQL prompted

10. You can see the SQL statement, as shown in Figure 10-35, by selecting **View** → **SQL**.

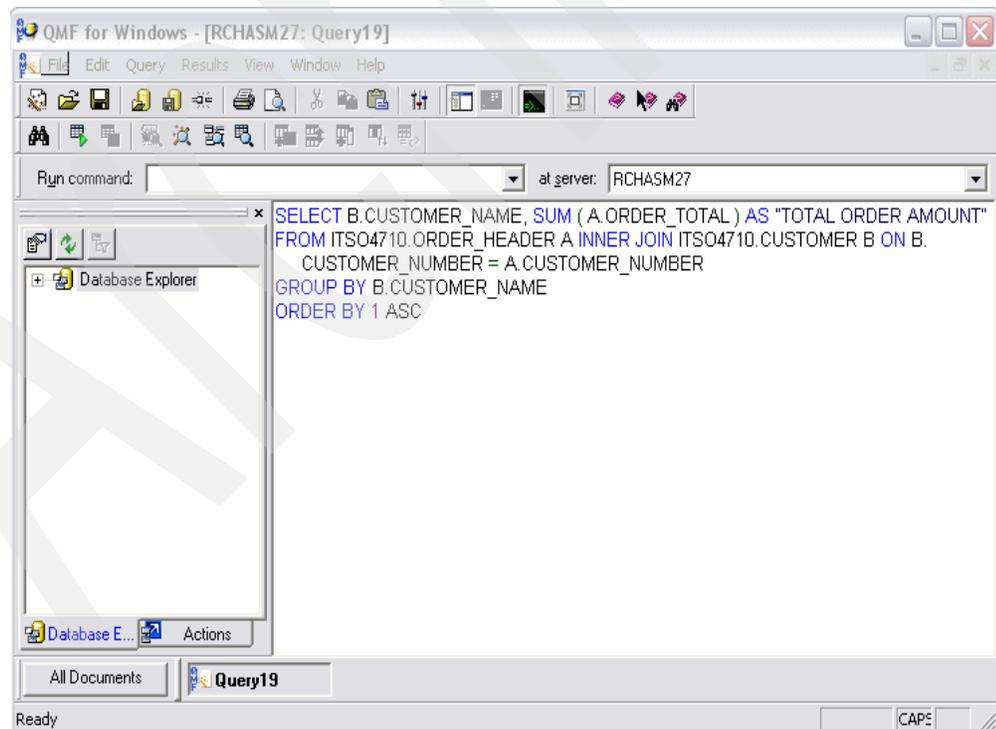


Figure 10-35 SQL statement

11. To run the query, select **Query** → **Run**, and the result will be shown, as in Figure 10-36 on page 254.

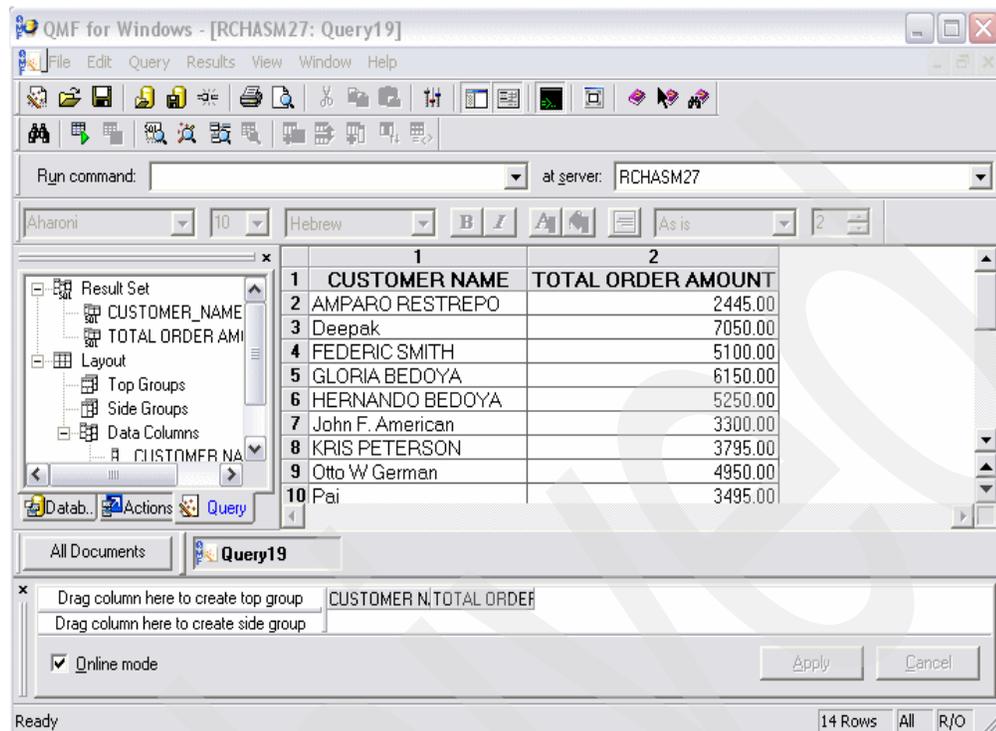


Figure 10-36 The query result

From the query result, you can change the format, print the report, and store as the document file. The end user may use the QMF as the query tool instead of the traditional Query/400.

You can find more information and download the QMF for Windows for evaluation at the Web site:

<http://www-306.ibm.com/software/data/qmf/>

The Query/400 and Query Manager/400 conversion tools are written by the business partner, Rocket Software. The tools can be downloaded from the Web site:

<http://www.rocketsoftware.com/qmf/qmf/cproducts.asp>

For more information on the .Net technology, please visit the Web site:

<http://www.ibm.com/developerworks/db2/downloads/dotnetbeta/>

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 257. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Advanced Functions and Administration on DB2 Universal Database for iSeries*, SG24-4249
- ▶ *Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries*, SG24-6503
- ▶ *Preparing for and Tuning the V5R2 SQL Query Engine on DB2 Universal Database for iSeries*, SG24-6598
- ▶ *IBM WebFacing Tool: Converting 5250 Applications to Browser-based GUIs*, SG24-6801
- ▶ *EJB 2.0 Development with WebSphere Studio Application Developer*, SG24-6819
- ▶ *DB2 Universal Database for iSeries Administration The Graphical Way on V5R3*, SG24-6092
- ▶ *Striving for Optimal Journal Performance*, SG24-6486
- ▶ *DB2 UDB for AS/400 Object Relational Support*, SG24-5409
- ▶ *Striving for Optimal Journal Performance on DB2 Universal Database for iSeries*, SG24-6286

## Other publications

These publications are also relevant as further information sources:

- ▶ *Database Programming*, SC41-5701
- ▶ *SQL Reference*, SC41-5612
- ▶ *DDS Reference*, SC41-5712
- ▶ Conte, Paul. *Database Design and Programming for DB2/400*. 29th Street Press, April 1997. ISBN 1-8824190-65
- ▶ Conte, Paul and Cravitz, Mike. *SQL/400® Developer's Guide*. 29th Street Press, September 2000. ISBN 1-882419-70-7
- ▶ Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional; 3rd edition, September 19, 2003. ISBN 0-321-19368-7

## Online resources

These Web sites and URLs are also relevant as further information sources:

- ▶ Indexing and statistics strategies for DB2 UDB for iSeries whitepaper  
[http://www-1.ibm.com/servers/enable/site/education/abstracts/indxng\\_abs.html](http://www-1.ibm.com/servers/enable/site/education/abstracts/indxng_abs.html)
- ▶ Table Partitioning Strategies for DB2 UDB for iSeries whitepaper  
[http://www-1.ibm.com/servers/enable/site/education/abstracts/2c52\\_abs.html](http://www-1.ibm.com/servers/enable/site/education/abstracts/2c52_abs.html)
- ▶ Information Center  
<http://www.iseries.ibm.com/infocenter>
- ▶ DB2 Universal Database for iSeries  
<http://www.ibm.com/iseries/db2>
- ▶ Additional information on the iSeries Developer Roadmap  
<http://www-1.ibm.com/servers/eserver/iseries/roadmap>
- ▶ Eclipse download and information  
<http://www.eclipse.org>
- ▶ Eclipse plug-ins  
<http://www.eclipse-plugins.info>
- ▶ Information on HATS  
<http://www-306.ibm.com/software/webservers/hats>
- ▶ iSeries Access for Web information  
<http://www-1.ibm.com/servers/eserver/iseries/access/web>
- ▶ Apache's Struts  
<http://struts.apache.org>
- ▶ Struts and the MVC design pattern  
<http://publib.boulder.ibm.com/infocenter/iadthelp/index.jsp?topic=/com.ibm.etools.struts.doc/html/cstruse0001.htm>
- ▶ UML  
<http://www.uml.org>
- ▶ OMG  
<http://www.omg.org>
- ▶ *DB2 Universal Database for iSeries SQL Reference* in the iSeries Information Center  
<http://publib.boulder.ibm.com/infocenter/iseries/v5r3/ic2924/index.htm?info/db2/rbafzmst.html>
- ▶ Managing DB2 UDB for iSeries Schemas and Journals, written by Kent Milligan  
<http://www7b.boulder.ibm.com/dmdd/library/techarticle/0305milligan/0305milligan.html>
- ▶ Download the QMF for Windows for evaluation  
<http://www-306.ibm.com/software/data/qmf/>
- ▶ Query/400 and Query Manager/400 conversion tools  
<http://www.rocketsoftware.com/qmf/qmf/cproducts.asp>
- ▶ .Net technology  
<http://www.ibm.com/developerworks/db2/downloads/dotnetbeta/>

## How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

## Help from IBM

IBM Support and downloads

[ibm.com/support](http://ibm.com/support)

IBM Global Services

[ibm.com/services](http://ibm.com/services)

Archived

Archived

# Index

## A

Activation group 80  
Add Physical File Trigger 129  
ADDPFTRG 129  
ALIAS 33  
ALTER SEQUENCE 71  
Application Developer 5  
Application Developer - Integration Edition 5  
audit trail 118  
Auto-incrementing of keys 16  
Automatic key generation 70  
automatic statistic 236

## B

Binary Data types 179  
Binary Large Objects (BLOBs) 16  
Binding directories 80  
bitmap indexes 49  
BLOB 16  
business rules 25

## C

CALL 167  
CASE expressions 49  
catalog tables 41  
Catalogs 50  
Character Data types 173  
Character Large Objects (CLOBs) 16  
CHECK constraint 17  
Check constraints 67  
Classic Query Engine (CQE) 13  
CLI 13  
CLOB 16  
Close SQL Cursor 81  
CLOSE statement 99  
CODE Designer 7  
Collection 22  
Column 22  
Column level security 69  
column statistics 235  
Column-level triggers 16  
Commitment Control 17  
commitment control 30  
Commitment control level 22  
Compound statement 164  
Condition Handler 168  
condition handler 82  
Constraint coexistence 68  
Constraints 17, 67  
Control Statements 164  
CREATE FUNCTION 159  
CREATE PROCEDURE 59, 139, 144  
CREATE SCHEMA 41

CREATE TABLE 32  
CREATE TRIGGER 132  
CRTBNDRPG 79  
CRTRPGMOD 79  
CRTSQLRPGI 80  
CRTSRVPGM 79  
Cursor Stability 103

## D

Data Definition Language (DDL) 12  
data dictionary 66  
data domains 66  
Data Manipulation Language 12  
Database Navigator 234  
Database normalization 66  
DataLink 16  
Datalink 16, 74  
Date Format 81  
Date Separator Character 81  
Date/Time functions 49  
DATFMT 81  
DATSEP 81  
DB2 Development Center 243  
DB2 schema 40  
DB2 SMP 243  
DB2 SMP (parallel database processing) 17  
DB2GENERAL 147  
DB2ROW 134  
DB2SQL 134, 147  
DBCLOB 16  
DDS data definitions 23  
DDS keyed logical 40  
DECLARE CURSOR 98, 113  
DECLARE HANDLER 168  
DECLARE statement 103  
Decryption 70  
decryption functions 16  
DLCOMMENT 75  
DLLINKTYPE 75  
DLURLCOMPLETE 75  
DLURLPATH 75  
DLURLPATHONLY 75  
DLURLSCHEME 75  
DLURLSERVER 75  
DLVALUE 75  
Double-Byte Character Large Objects (DBCLOBs) 16  
DROP FUNCTION 162  
DROP PROCEDURE 151  
DYNAMIC RESULT SETS 145  
Dynamic SQL 24, 109

## E

Eclipse plug-ins 6  
Eclipse technology 5, 226

EJB 9  
Embedded SQL 77  
Encoded Vector Indices 17, 23  
Encryption 16, 70  
ENDJRNPf 47  
Enterprise Developer 5  
Enterprise Generation Language (EGL) 5  
Enterprise Java Beans 226  
Enterprise JavaBeans 5, 8  
EXCEPTION JOIN 77  
EXECUTE IMMEDIATE 110  
Extended dynamic SQL 24  
EXTERNAL NAME 147  
External Stored Procedures 139  
External Triggers 119  
External triggers 122  
Externalizing I/O operations 62  
externally described 30

## F

FETCH statement 98  
Field 22  
Field Reference files 66  
First Normal Form 66  
Floating point numeric data type 183  
FOR 165  
FOR COLUMN clause 32  
FOREIGN KEY 17  
FREE LOCATOR 76

## G

GENERAL 146  
GENERAL WITH NULLS 146  
Generate SQL 34  
GET DIAGNOSTICS 83  
GOTO 166  
Graphical iSeries System Debugger 238

## H

Hash partitioning 53  
history of SQL 12  
Host variables 86

## I

I/O modules 25  
Identity column attribute 71  
ILE procedure 59  
Index 22  
index maintenance 49  
Initialization Subroutine 98  
iSeries Access for Web 6  
iSeries Developer Roadmap 1  
iSeries Navigator 30, 71, 234  
Isolation Level 22  
ITERATE 166  
Iterative Control 165

## J

JAVA 147  
Java Virtual Machine 8  
JDBC 13  
JDBC catalog 51  
JDFTVAL 38  
JDUPSEQ 38  
Journal 22, 41  
journal receiver 41  
Journaling 17, 30

## K

key generation 70  
Keyed logical file 22

## L

Large Object Support 75  
LEAVE 166  
LEFT OUTER JOIN 38, 77  
legacy programs 25  
Library 22  
library 41  
LOB columns 23  
Log 22  
logical file 17  
logical files 49  
logical page size 49  
long name support 32  
LOOP 165  
LVLCHK 39

## M

Mathematical functions 92  
Message Driven Beans 5  
Metadata 50  
MNGRCV 42  
Model-View-Controller 8  
module object 80  
multi-member files 23  
Multiple format logical file 40  
Multiple format logical files 34  
Multiple member files 48

## N

naming conventions 58  
No Commit 103  
Non keyed logical file 22  
non-relational data 72  
Normalization 66  
Numeric data types 178

## O

Object Management Group 8  
Object-Oriented programming 2  
ODBC 13, 51  
OPEN statement 98, 103  
OPNQRYF 64

OPNQRYP command 13  
Overloading 148  
overloading 148  
OVRDBF 23

## P

Packed numeric data type 179  
Parameter-Style 146  
Partitioned tables 53  
Personal Digital Assistants 8  
Physical File 22  
physical file object 22  
physical files 48  
pointers 127  
PRIMARY KEY 17  
Program described files. 30  
program object 80  
Programming Development Manager (PDM) 4  
prototype 59  
prototype call 63  
PRTSQLINF 243

## Q

QSYS2 50  
Query Management Facility 244  
Query Manager 13  
query performance 68  
Query/400 13

## R

Range partitioning 53  
Read Stability 103  
Receive Program Message API 69  
Record 22  
Record format level check 39  
Record format name 39  
Redbooks Web site 257  
    Contact us xi  
Referential integrity 17, 67  
Registering an external trigger 129  
Registering external triggers 129  
Registering Stored Procedures 140  
Remote System Explorer (RSE) 6  
RENAME INDEX 32  
RENAME TABLE 32  
Reorganize Physical File Member 37  
REPEAT 165  
Repeatable Read 103  
Report Layout Utility (RLU) 4  
RESIGNAL 169  
RETURN 167  
REUSEDLT 37  
REVOKE 69  
RGZPFM (Reorganize Physical File Member) 37  
ROLLBACK 77  
Row 22  
ROWID data type 71  
RTVQMFORM 245

RTVQMORY 245  
Run SQL Scripts 235  
RUNSQLSTM 13  
RUNSQLSTM command 35

## S

S/36 data 72  
SAVEPOINT 77  
SAVLIB 47  
Schema 22  
Screen Design Aid (SDA) 4  
Scroll Cursor 107  
Second Normal Form 66  
Select/Omit filtering 49  
SEQUEL 12  
Sequence object 71  
Serial Cursor 107  
service program 62  
service program object 80  
service programs 58  
Set Option Statement 82  
Set Statement 93  
SIGNAL 169  
Site Developer 5  
Softcoding the trigger buffer 129  
Source Entry Utility (SEU) 4  
SQE Optimizer 14  
SQL Alias 23  
SQL Assist 235  
SQL communication area 83  
SQL data types 49, 172  
SQL Descriptor Area 114  
SQL GRANT 69  
SQL index 17  
SQL indexes 49  
SQL Query Engine (SQE) 13  
SQL SET TRANSACTION 126  
SQL Stored procedures 139  
SQL tables 48  
SQL Triggers 119, 131  
SQL Views 40  
SQL views 26, 49  
SQLCA 82  
SQLCODE 83  
SQLCOLPRIVILEGES 51  
SQLCOLUMNS 51  
SQLFOREIGNKEYS 51  
SQLPRIMARYKEYS 51  
SQLPROCEDURECOLS 51  
SQLPROCEDURES 52  
SQLSCHEMAS 52  
SQLSPECIALCOLUMNS 52  
SQLSTATE 83  
SQLSTATISTICS 52  
SQLTABLEPRIVILEGES 52  
SQLTABLES 52  
SQLTYPEINFO 52  
SQLUDTS 52  
Static 24  
Static SQL 24

- Statistics Manager 235
- Stored Procedures 138
- STRCMTCTL 122, 131
- STRDBMON 243
- String functions 91
- Struts 8
- Symmetric Multiprocessing 17
- SYSCATALOGS 50
- SYSCHKCST 50
- SYSCOLUMNS 50
- SYSCST 50
- SYSCSTCOL 50
- SYSCSTDEP 50
- SYSFUNCS 50
- SYSINDEXES 51
- SYSJARCONTENTS 51
- SYSJAROBJECTS 51
- SYSKEYCST 51
- SYSKEYS 51
- SYSPACKAGE 51
- SYSPARMS 51
- SYSPROCS 51
- SYSREFCST 51
- SYSROUTINEDEP 51
- SYSROUTINES 51
- SYSSEQUENCES 51
- SYSTABLEDEP 51
- SYSTABLES 51
- system catalog tables 31
- system catalogs 32
- SYSTRIGCOL 51, 138
- SYSTRIGDEP 138
- SYSTRIGGERS 51, 138
- SYSTRIGUPD 51, 138
- SYSTYPES 51
- SYSVIEWDEP 51
- SYSVIEWS 51

## T

- Table 22
- Third Normal Form 66
- time data types 185
- Time Format 81
- Time functions 92
- Time Separator Character 81
- TIMFMT 81
- TIMSEP 81
- traditional 5250 tools 4
- Transaction files 30
- Trigger 118
- Trigger Body 135
- Trigger Buffer 122–123
- Trigger Event 132
- Trigger event 119
- Trigger events 121
- Trigger Name 132
- Trigger Time 132
- Trigger time 119
- Triggers 18
- Trigonometric functions 92

## U

- UDTFs 72
- Uncommitted Read 103
- Unified Modeling Language 8
- Uniform Resource Locator 74
- unique identifiers 70
- unique key constraints 67
- Unsupported DDS keywords 38
- User Defined Functions 152
- User Defined Scalar Functions 159
- User Defined Table Functions 160
- User Defined Types 23
- User-defined Distinct Type 152

## V

- validation list object 70
- VARBINARY 75
- VARCHAR 75
- VARGRAPHIC 75
- View 22
- VisualAge Generator 5

## W

- Web service 4
- WebFacing Tool 6–7
- WebSphere Application Server 7
- WebSphere Application Server - Express 6
- WebSphere Development Studio Client for iSeries 226
- WebSphere Host Access Transformation Services (HATS) 6
- WebSphere Studio 5
- WHILE 165
- Wrapper program 59

## Z

- Zoned numeric data type 178



**Modernizing IBM @server iSeries Application Data Access - A Roadmap Cornerstone**

Archived







# Modernizing IBM *e*server iSeries Application Data Access - A Roadmap Cornerstone



**Learn how to move  
your data definition of  
your applications  
from DDS to SQL**

**Discover the ways to  
enhance your data  
access using SQL**

**Understand the  
iSeries developers  
roadmap**

In 1978 IBM introduced the System/38 as part of its midrange platform hardware base. One of the many outstanding features of this system was the built-in Relational Database Management System (RDMS) support. The system included a utility for defining databases, screens, and reports. This utility used a form named Data Description Specifications (DDS) to define the database physical (PF) and logical (LF) files (base tables, views, and indexes).

In 1988, IBM announced the AS/400. The OS/400 operating system also contained a built-in RDMS, however, IBM offered Structured Query Language (SQL) as an alternative to DDS for creating databases. In addition, SQL Data Manipulation Language (DML) statements were made available as an ad hoc query language tool. These statements could also be embedded and compiled within high level language (HLL) programs.

SQL Data Definition Language (DDL) has become the industry standard for defining RDMS databases.

Many customers are in the process of modernizing their database definition and the database access. This IBM Redbook will help you understand how to reverse engineer a DDS-created database along, and provides you with tips and techniques for modernizing applications to use SQL as the database access method.

**INTERNATIONAL  
TECHNICAL  
SUPPORT  
ORGANIZATION**

**BUILDING TECHNICAL  
INFORMATION BASED ON  
PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:  
[ibm.com/redbooks](http://ibm.com/redbooks)**

SG24-6393-00

ISBN 0738492132