**SQL: The Complete Reference** 

by James R. Groff and Paul N. Weinberg ISBN: 0072118458 Osborne/McGraw-Hill © 1999, 994 pages

An encyclopedic reference guide to the SQL database

language for both technical and non-technical readers.

Table of Contents Colleague Comments Back Cover

**Synopsis** by Dean Andrews

What is SQL and where did it come from? How do the SQL tools vary across database applications from different vendors? How will SQL change in the future? You'll find the answers to all these questions and many more in *SQL: The Complete Reference*. Much more than just a listing of SQL commands and their parameters, this encyclopedic reference guide explains the concepts and constructs of SQL programming such that non-technical readers will understand them and technical readers won't be bored.

**Table of Contents**

SQL - The Complete Reference - 4

Preface - 6

**Part I An Overview of SQL**

Chapter 1 - Introduction - 8

Chapter 2 - A Quick Tour of SQL - 15

Chapter 3 - SQL In Perspective - 22

Chapter 4 - Relational Databases - 38

**Part II Retrieving Data**

Chapter 5 - SQL Basics - 51

Chapter 6 - Simple Queries - 69

Chapter 7 - Multi-Table Queries (Joins) - 101

Chapter 8 - Summary Queries - 136

Chapter 9 - Subqueries and Query Expressions - 158

**Part III Updating Data**

Chapter 10 - Database Updates - 196

Chapter 11 - Data Integrity - 211

Chapter 12 - Transaction Processing - 236

**Part IV Database Structure**

Chapter 13 - Creating a Database - 256

Chapter 14 - Views - 290

Chapter 15 - SQL Security - 304

- 2 -

Chapter 16 - The System Catalog - 321

**Part V Programming with SQL**

Chapter 17 - Embedded SQL - 344

Chapter 18 - Dynamic SQL\* - 387

Chapter 19 - SQL APIs - 430

**Part VI SQL Today and Tomorrow**

Chapter 20 - Database Processing and Stored Procedures - 435 Chapter 21 - SQL and Data Warehousing - 535

Chapter 22 - SQL Networking and Distributed Databases - 546 Chapter 23 - SQL and Objects - 575

Chapter 24 - The Future of SQL - 602

**Part VII Appendices**

Appendix A - The Sample Database - 612

Appendix B - Database Vendor Profiles - 616

Appendix C - Company and Product List - 629

Appendix D - SQL Syntax Reference - 634

Appendix E - SQL Call Level Interface - 635

Appendix F - SQL Information Schema Standard - 651

Appendix G - CD-ROM Installation Guide - 667

**Back Cover**

Gain the working knowledge of SQL and relational databases essential for today's information systems professionals. Relational databases built on the SQL database language are the foundation for modern enterprise data processing and are also a force behind many of today's important technology trends.

*SQL: The Complete Reference* provides an in-depth discussion of SQL fundamentals, modern SQL products, and SQL's role in trends such as data warehousing, "thin-client" architectures, and Internet-based e-commerce. This book is your one-stop resource for all you need to know about SQL. It will help you:

• Learn the key concepts and latest developments in relational database technology

• Master the industry-standard SQL language for managing database information

• Understand the differences among all the leading brands of DBMS systems

• Set up and manage SQL-based databases and write programs to access them

• Understand how to use and program SQL databases with application servers and the Web to create e-commerce applications

• Find out more about the proposed SQL3 standard and the key trends in object technologies, 64-bit architectures, distributed databases, 3- tier Internet applications, and more

**About the Authors**

James R. Groff and Paul N. Weinberg were the co-founders of Network Innovations Corporation, an early developer of SQL-based networking

- 3 -

software that links personal computers to corporate databases. Groff is currently CEO of TimesTen Performance Software, developer of an ultra-high performance main-memory SQL database for communications and Internet applications. Weinberg is vice president of A2i, Inc., developer of a database driven, cross-media catalog publishing system that supports printed and electronic output from a single data source.

**SQL: The Complete Reference**

**James R. Groff**

**Paul N. Weinberg**

**Publisher**

Brandon A Nordin

**Associate Publisher and Editor-in-Chief**

Scott Rogers

**Senior Acquisitions Editor**

Wendy Rinaldi

**Acquisitions Editor**

Jane K. Brownlow

**Project Editor**

Heidi Poulin

**Editorial Assistant**

Monica Faltiss

**Copy Editor**

Nancy Crumpton

**Proofreader**

Rhonda Holmes

**Indexer**

Valerie Robbins

**Computer Designer**

Jani Beckwith

Michelle Galicia

**Illustrators**

Robert Hansen

Brian Wells

Beth Young

- 4 -

Osborne/**McGraw-Hill**

2600 Tenth Street

Berkeley, California 94710

U.S.A.

For information on translations or book distributors outside the U.S.A., or to arrange bulk purchase discounts for sales promotions, premiums, or fund-raisers, please contact Osborne/**McGraw-Hill** at the above address.

Copyright © 1999 by The McGraw-Hill Companies. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

Licensed Materials - Property of IBM

IBM® DB2® Universal Database Personal Edition, Version 5.2, for the Windows® Operating Environments© Copyright IBM Corp. 1993, 1998. All Rights Reserved. U.S. Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP schedule Contract with IBM Corp.

© 1999 Informix Corporation. All rights reserved. Informix® is a trademark of Informix Corporation or its affiliates and is registered in the U.S. and some other jurisdictions.

Microsoft® SQL Server ™ 7.0 Evaluation Edition. Copyright Microsoft Corporation, 1997- 98. All rights reserved.

Oracle8 Personal Edition© 1996,1998, Oracle Corporation. All rights reserved. Copyright © 1996-1998, Sybase, Inc. All rights reserved.

1234567890 DOC DOC 90198765432109

ISBN 0-07-211845-8

Information has been obtained by Osborne/**McGraw-Hill** from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Osborne/**McGraw-Hill**, or others, Osborne/**McGraw-Hill** does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information.

**Acknowledgments**

Special thanks to Matan Arazi for doing such an exceptional job assembling the Bonus CD-ROM. He pulled off a real miracle to squeeze all five SQL, DBMS products onto a single CD, a technical feat that would not have been possible without his diligent tenacity

Thanks also to everyone at Osborne for pulling it all together, including Jane Brownlow and Wendy Rinaldi for doing tag-team duty as our acquisitions editors, and to Heidi Poulin for her meticulous attention to detail.

- 5 -

**Preface**

**Overview**

*SQL: The Complete Reference* provides a comprehensive, in-depth treatment of the SQL language for both technical and non-technical users, programmers, data processing professionals, and managers who want to understand the impact of SQL in the computer market. This book offers a conceptual framework for understanding and using SQL, describes the history of SQL and SQL standards, and explains the role of SQL in the computer industry today. It will show you, step-by-step, how to use SQL features, with many illustrations and realistic examples to clarify SQL concepts. The book also compares SQL products from leading DBMS vendors  describing their advantages, benefits, and trade-offs  to help you select the right product for your application. The accompanying CD contains actual trial versions of five leading SQL databases, so you can try them for yourself and gain actual experience in using major database products from Oracle, Microsoft, Sybase, Informix, an IBM.

In some of the chapters in this book, the subject matter is explored at two different levels— a fundamental description of the topic, and an advanced discussion intended for computer professionals who need to understand some of the "internals" behind SQL. The more advanced information is covered in sections marked with an asterisk (\*). You do not need to read these sections to obtain an understanding of what SQL is and what it does.

**How this Book Is Organized**

The book is divided into six parts that cover various aspects of the SQL language:

• Part One, "An Overview of SQL," provides an introduction to SQL and a market perspective of its role as a database language. Its four chapters describe the history of

SQL, the evolution of SQL standards, and how SQL relates to the relational data

model and to earlier database technologies. Part One also contains a quick tour of SQL that briefly illustrates its most important features and provides you with an overview of the entire language early in the book.

• Part Two, "Retrieving Data," describes the features of SQL that allow you to perform database queries. The first chapter in this part describes the basic structure of the

SQL language. The next four chapters start with the simplest SQL queries, and progressively build to more complex queries, including multi-table queries, summary queries, and queries that use subqueries.

• Part Three, "Updating Data," shows how you can use SQL to add new data to a database, delete data from a database, and modify existing database data. It also

describes the database integrity issues that arise when data is updated, and how SQL addresses these issues. The last of the three chapters in this part discusses the SQL transaction concept and SQL support for multi-user transaction processing.

• Part Four, "Database Structure," deals with creating and administering a SQL-based database. Its four chapters tell you how to create the tables, views, and indexes that form the structure of a relational database. It also describes the SQL security scheme

that prevents unauthorized access to data, and the SQL system catalog that describes the structure of a database. This part also discusses the significant differences between the database structures supported by various SQL-based DBMS products.

• Part Five, "Programming with SQL," describes how application programs use SQL for database access. It discusses the embedded SQL specified by the ANSI standard and

used by IBM, Oracle, Ingres, Informix, and most other SQL-based DBMS products. It also describes the dynamic SQL interface that is used to build general-purpose database tools, such as report writers and database browsing programs. Finally, this

- 6 -

part describes the popular SQL APIs, including ODBC, the ISO-standard Call-Level Interface, and Oracle Call Interface, and contrasts them with the embedded SQL interface.

• Part Six, "SQL Today and Tomorrow," examines the state of SQL-based DBMS products today, major database trends, the "hot" new applications, and the directions that SQL will take over the next few years. It describes the intense current activity in

SQL networking and distributed databases, and the evolution of special features to support SQL-based OLTP, and SQL-based data warehousing. This part also discusses the impact of object technology on SQL and relational databases, and the emergence of hybrid, object-relational database models.

**Conventions Used in this Book**

*SQL: The Complete Reference* describes the SQL features and functions that are available in the most popular SQL-based DBMS products and those that are described in the ANSI/ISO SQL standards. Whenever possible, the SQL statement syntax described in the book and used in the examples applies to all dialects of SQL. When the dialects differ, the differences are pointed out in the text, and the examples follow the most common practice. In these cases, you may have to modify the SQL statements in the examples slightly to suit your particular brand of DBMS.

Throughout the book, technical terms appear in italics the first time that they are used and defined. SQL language elements, including SQL keywords, table and column names, and sample SQL statements appear in an uppercase monospace font. SQL API function names appear in a lowercase monospace font. Program listings also appear in monospace font, and use the normal case conventions for the particular programming language (uppercase for COBOL and FORTRAN, lowercase for C). Note that these conventions are used solely to improve readability; most SQL implementations will accept either uppercase or lowercase statements. Many of the SQL examples include query results, which appear immediately following the SQL statement as they would in an interactive SQL session. In some cases, long query results are truncated after a few rows; this is indicated by a vertical ellipsis (. . .) following the last row of query results.

**Why this Book Is for You**

*SQL: The Complete Reference* is the right book for anyone who wants to understand and learn SQL, including database users, data processing professionals, programmers, students, and managers. It describes—in simple, understandable language liberally illustrated with figures and examples—what SQL is, why it is important, and how you use it. This book is not specific to one particular brand or dialect of SQL. Rather, it describes the standard, central core of the SQL language and then goes on to describe the differences among the most popular SQL products, including Oracle, Microsoft SQL Server, IBM's DB2, Informix Universal Server, Sybase Adaptive Server, and others. It also explains the importance of SQL-based standards, such as ODBC and the ANSI/ISO SQL2 and evolving SQL3 standards.

If you are a new user of SQL, this book offers comprehensive, step-by-step treatment of the language, building from simple queries to more advanced concepts. The structure of the book will allow you to quickly start using SQL, but the book will continue to be valuable as you begin to use more complex features of the language. You can use the SQL software on the companion CD to try out the examples and build your SQL skills.

If you are a data processing professional or a manager, this book will give you a perspective on the impact that SQL is having in every segment of the computer market— from personal computers, to mainframes, to online transaction processing systems and data warehousing applications. The early chapters describe the history of SQL, its role in the market, and its evolution from earlier database technologies. The final chapters describe the future of SQL and the development of new database technologies such as distributed databases, business intelligence databases, and object-relational database capabilities.

- 7 -

If you are a programmer, this book offers a very complete treatment of programming with SQL. Unlike the reference manuals of many DBMS products, it offers a conceptual framework for SQL programming, explaining the why as well as the how of developing a SQL-based application. It contrasts the SQL programming interfaces offered by all of the leading SQL products, including embedded SQL, dynamic SQL, ODBC and proprietary APIs such as the Oracle Call Interface, providing a perspective not found in any other book.

If you are selecting a DBMS product, this book offers a comparison of the SQL features, advantages, and benefits offered by the various DBMS vendors. The differences between the leading DBMS products are explained, not only in technical terms, but also in terms of their impact on applications and their competitive position in the marketplace. The DBMS software on the companion CD can be used to try out these features in a prototype of your own application.

In short, both technical and non-technical users can benefit from this book. It is the most comprehensive source of information available about the SQL language, SQL features and benefits, popular SQL-based products, the history of SQL, and the impact of SQL on the future direction of the computer market.

**Part I: An Overview of SQL**

**Chapter List**

**Chapter  1:**

**Chapter 2:**

**Chapter 3:**

**Chapter 4:**

**Introduction**

**A Quick Tour of SQL SQL In Perspective Relational Databases**

**Chapter 1: Introduction**

**Overview**

The SQL language and relational database systems based on it are one of the most important foundation technologies in the computer industry today. Over the last decade, the popularity of SQL has exploded, and it stands today as *the* standard computer database language. Literally hundreds of database products now support SQL, running on computer systems from mainframes to personal computers and even handheld devices. An official international SQL standard has been adopted and expanded twice. Virtually every major enterprise software product relies on SQL for its data management, and SQL is at the core of the database products from Microsoft and Oracle, two of the largest software companies in the world. From its obscure beginnings as an IBM research project, SQL has leaped to prominence as both an important computer technology and a powerful market force.

What, exactly, is SQL? Why is it important? What can it do, and how does it work? If SQL is really a standard, why are there so many different versions and dialects? How do popular SQL products like SQL Server, Oracle, Informix, Sybase, and DB2 compare? How

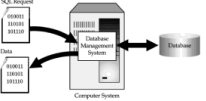
- 8 -

does SQL relate to Microsoft standards, such as ODBC and COM? How does JDBC link SQL to the world of Java and object technology? Does SQL really scale from mainframes to handheld devices? Has it really delivered the performance needed for high-volume transaction processing? How will SQL impact the way you use computers, and how can you get the most out of this important data management tool?

**The SQL Language**

SQL is a tool for organizing, managing, and retrieving data stored by a computer database. The name "SQL" is an abbreviation for *Structured Query Language*. For historical reasons, SQL is usually pronounced "sequel," but the alternate pronunciation "S.Q.L." is also used. As the name implies, SQL is a computer *language* that you use to interact with a database. In fact, SQL works with one specific type of database, called a *relational database*.

Figure 1-1 shows how SQL works. The computer system in the figure has a *database* that stores important information. If the computer system is in a business, the database might store inventory, production, sales, or payroll data. On a personal computer, the database might store data about the checks you have written, lists of people and their phone numbers, or data extracted from a larger computer system. The computer program that controls the database is called a *database management system*, or DBMS.



**Figure 1-1:** Using SQL for database access

When you need to retrieve data from a database, you use the SQL language to make the request. The DBMS processes the SQL request, retrieves the requested data, and returns it to you. This process of requesting data from a database and receiving back the results is called a database *query*—hence the name Structured *Query* Language.

The name Structured Query Language is actually somewhat of a misnomer. First of all, SQL is far more than a query tool, although that was its original purpose and retrieving data is still one of its most important functions. SQL is used to control all of the functions that a DBMS provides for its users, including:

• *Data definition*. SQL lets a user define the structure and organization of the stored data and relationships among the stored data items.

• *Data retrieval*. SQL allows a user or an application program to retrieve stored data from the database and use it.

• *Data manipulation*. SQL allows a user or an application program to update the

database by adding new data, removing old data, and modifying previously stored data.

• *Access control*. SQL can be used to restrict a user's ability to retrieve, add, and modify data, protecting stored data against unauthorized access.

• *Data sharing*. SQL is used to coordinate data sharing by concurrent users, ensuring that they do not interfere with one another.

- 9 -

• *Data integrity*. SQL defines integrity constraints in the database, protecting it from  corruption due to inconsistent updates or system failures.

SQL is thus a comprehensive language for controlling and interacting with a database management system.

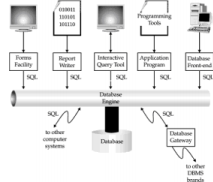
Second, SQL is not really a complete computer language like COBOL, C, C++, or Java. SQL contains no IF statement for testing conditions, and no GOTO, DO, or FOR statements for program flow control. Instead, SQL is a database *sublanguage,* consisting of about forty statements specialized for database management tasks. These SQL statements can be *embedded* into another language, such as COBOL or C, to extend that language for use in database access. Alternatively, they can be explicitly sent to a database management system for processing, via a *call level interface* from a language such as C, C++, or Java.

Finally, SQL is not a particularly structured language, especially when compared to highly structured languages such as C, Pascal, or Java. Instead, SQL statements resemble English sentences, complete with "noise words" that don't add to the meaning of the statement but make it read more naturally. There are quite a few inconsistencies in the SQL language, and there are also some special rules to prevent you from constructing SQL statements that look perfectly legal, but don't make sense.

Despite the inaccuracy of its name, SQL has emerged as *the* standard language for using relational databases. SQL is both a powerful language and one that is relatively easy to learn. The quick tour of SQL in the next chapter will give you a good overview of the language and its capabilities.

**The Role of SQL**

SQL is not itself a database management system, nor is it a stand-alone product. You cannot go into a computer store and "buy SQL." Instead, SQL is an integral part of a database management system, a language and a tool for communicating with the DBMS. Figure 1-2 shows some of the components of a typical DBMS, and how SQL acts as the "glue" that links them together.



**Figure 1-2:** Components of a typical database management system

The *database engine* is the heart of the DBMS, responsible for actually structuring, storing, and retrieving the data in the database. It accepts SQL requests from other DBMS components, such as a forms facility, report writer, or interactive query facility, from user-written application programs, and even from other computer systems. As the

- 10 -

figure shows, SQL plays many different roles:

• SQL is an *interactive query language*. Users type SQL commands into an interactive

SQL program to retrieve data and display it on the screen, providing a convenient, easy-to-use tool for ad hoc database queries.

• SQL is a *database programming language*. Programmers embed SQL commands into their application programs to access the data in a database. Both user-written

programs and database utility programs (such as report writers and data entry tools) use this technique for database access.

• SQL is a *database administration language*. The database administrator responsible

for managing a minicomputer or mainframe database uses SQL to define the database structure and control access to the stored data.

• SQL is a *client/server language*. Personal computer programs use SQL to

communicate over a network with database servers that store shared data. This client/server architecture has become very popular for enterprise-class applications.

• SQL is an *Internet data access language*. Internet web servers that interact with

corporate data and Internet applications servers all use SQL as a standard language for accessing corporate databases.

• SQL is a *distributed database language*. Distributed database management systems use SQL to help distribute data across many connected computer systems. The

DBMS software on each system uses SQL to communicate with the other systems, sending requests for data access.

• SQL is a *database gateway language*. In a computer network with a mix of different

DBMS products, SQL is often used in a *gateway* that allows one brand of DBMS to communicate with another brand.

SQL has thus emerged as a useful, powerful tool for linking people, computer programs, and computer systems to the data stored in a relational database.

**SQL Features and Benefits**

SQL is both an easy-to-understand language and a comprehensive tool for managing data. Here are some of the major features of SQL and the market forces that have made it successful:

• Vendor independence

• Portability across computer systems • SQL standards

• IBM endorsement (DB2)

• Microsoft commitment (ODBC and ADO) • Relational foundation

• High-level, English-like structure • Interactive, ad hoc queries

- 11 -

• Programmatic database access

• Multiple views of data

• Complete database language

• Dynamic data definition

• Client/server architecture

• Extensibility and object technology

• Internet database access

• Java integration (JDBC)

These are the reasons why SQL has emerged as the standard tool for managing data on personal computers, minicomputers, and mainframes. They are described in the sections that follow.

**Vendor Independence**

SQL is offered by all of the leading DBMS vendors, and no new database product over the last decade has been highly successful without SQL support. A SQL-based database and the programs that use it can be moved from one DBMS to another vendor's DBMS with minimal conversion effort and little retraining of personnel. PC database tools, such as query tools, report writers, and application generators, work with many different brands of SQL databases. The vendor independence thus provided by SQL was one of the most important reasons for its early popularity and remains an important feature today.

**Portability Across Computer Systems**

SQL-based database products run on computer systems ranging from mainframes and midrange systems to personal computers, workstations, and even handheld devices. They operate on stand-alone computer systems, in departmental local area networks, and in enterprise-wide or Internet-wide networks. SQL-based applications that begin on single-user systems can be moved to larger server systems as they grow. Data from corporate SQL-based databases can be extracted and downloaded into departmental or personal databases. Finally, economical personal computers can be used to prototype a SQL-based database application before moving it to an expensive multi-user system.

**SQL Standards**

An official standard for SQL was initially published by the American National Standards Institute (ANSI) and the International Standards Organization (ISO) in 1986, and was expanded in 1989 and again in 1992. SQL is also a U.S. Federal Information Processing Standard (FIPS), making it a key requirement for large government computer contracts. Over the years, other international, government, and vendor groups have pioneered the standardization of new SQL capabilities, such as call-level interfaces or object-based extensions. Many of these new initiatives have been incorporated into the ANSI/ISO standard over time. The evolving standards serve as an official stamp of approval for SQL and have speeded its market acceptance.

**IBM Endorsement (DB2)**

- 12 -

SQL was originally invented by IBM researchers and has since become a strategic product for IBM based on its flagship DB2 database. SQL support is available on all major IBM product families, from personal computers through midrange systems (AS/400 and RS/6000) to IBM mainframes running both the MVS and VM operating systems. IBM's initial work provided a clear signal of IBM's direction for other database and system vendors to follow early in the development of SQL and relational databases. Later, IBM's commitment and broad support speeded the market acceptance of SQL.

**Microsoft Commitment (ODBC and ADO)**

Microsoft has long considered database access a key part of its Windows personal computer software architecture. Both desktop and server versions of Windows provide standardized relational database access through Open Database Connectivity (ODBC), a SQL-based call-level API. Leading Windows software applications (spreadsheets, word processors, databases, etc.) from Microsoft and other vendors support ODBC, and all leading SQL databases provide ODBC access. Microsoft has enhanced ODBC support with higher-level, more object-oriented database access layers as part of its Object Linking and Embedding technology (OLE DB), and more recently as part of Active/X (Active/X Data Objects, or ADO).

**Relational Foundation**

SQL is a language for relational databases, and it has become popular along with the relational database model. The tabular, row/column structure of a relational database is intuitive to users, keeping the SQL language simple and easy to understand. The relational model also has a strong theoretical foundation that has guided the evolution and implementation of relational databases. Riding a wave of acceptance brought about by the success of the relational model, SQL has become *the* database language for relational databases.

**High-Level, English-Like Structure**

SQL statements look like simple English sentences, making SQL easy to learn and understand. This is in part because SQL statements describe the *data* to be retrieved, rather than specifying *how* to find the data. Tables and columns in a SQL database can have long, descriptive names. As a result, most SQL statements "say what they mean" and can be read as clear, natural sentences.

**Interactive, Ad Hoc Queries**

SQL is an interactive query language that gives users ad hoc access to stored data. Using SQL interactively, a user can get answers even to complex questions in minutes or seconds, in sharp contrast to the days or weeks it would take for a programmer to write a custom report program. Because of SQL's ad hoc query power, data is more accessible and can be used to help an organization make better, more informed decisions. SQL's ad hoc query capability was an important advantage over nonrelational databases early in its evolution and more recently has continued as a key advantage over pure object-based databases.

**Programmatic Database Access**

SQL is also a database language used by programmers to write applications that access a database. The same SQL statements are used for both interactive and programmatic access, so the database access parts of a program can be tested first with interactive SQL and then embedded into the program. In contrast, traditional databases provided one set of tools for programmatic access and a separate query facility for ad hoc requests, without any synergy between the two modes of access.

**Multiple Views of Data**

- 13 -

Using SQL, the creator of a database can give different users of the database different *views* of its structure and contents. For example, the database can be constructed so that each user sees data for only their department or sales region. In addition, data from several different parts of the database can be combined and presented to the user as a simple row/column table. SQL views can thus be used to enhance the security of a database and tailor it to the particular needs of individual users.

**Complete Database Language**

SQL was first developed as an ad hoc query language, but its powers now go far beyond data retrieval. SQL provides a complete, consistent language for creating a database, managing its security, updating its contents, retrieving data, and sharing data among many concurrent users. SQL concepts that are learned in one part of the language can be applied to other SQL commands, making users more productive.

**Dynamic Data Definition**

Using SQL, the structure of a database can be changed and expanded dynamically, even while users are accessing database contents. This is a major advance over static data definition languages, which prevented access to the database while its structure was being changed. SQL thus provides maximum flexibility, allowing a database to adapt to changing requirements while on-line applications continue uninterrupted.

**Client/Server Architecture**

SQL is a natural vehicle for implementing applications using a distributed, client/server architecture. In this role, SQL serves as the link between "front-end" computer systems optimized for user interaction and "back-end" systems specialized for database management, allowing each system to do what it does best. SQL also allows personal computers to function as front-ends to network servers or to larger minicomputer and mainframe databases, providing access to corporate data from personal computer applications.

**Extensibility and Object Technology**

The major challenge to SQL's continued dominance as a database standard has come from the emergence of object-based programming, and the introduction of object-based databases as an extension of the broad market trend toward object-based technology. SQL-based database vendors have responded to this challenge by slowly expanding and enhancing SQL to include object features. These "object/relational" databases, which continue to be based on SQL, have emerged as a more popular alternative to "pure object" databases and may insure SQL's continuing dominance for the next decade.

**Internet Database Access**

With the exploding popularity of the Internet and the World Wide Web, and their standards-based foundation, SQL found a new role in the late 1990s as an Internet data access standard. Early in the development of the Web, developers needed a way to retrieve and present database information on web pages and used SQL as a common language for database gateways. More recently, the emergence of three-tiered Internet architectures with distinct thin client, application server and database server layers, have established SQL as the standard link between the application and database tiers.

**Java Integration (JDBC)**

One of the major new areas of SQL development is the integration of SQL with Java. Seeing the need to link the Java language to existing relational databases, Sun

- 14 -

Microsystems (the creator of Java) introduced Java Data Base Connectivity (JDBC), a standard API that allows Java programs to use SQL for database access. Many of the leading database vendors have also announced or implemented Java support *within* their database systems, allowing Java to be used as a language for stored procedures and business logic within the database itself. This trend toward integration between Java and SQL will insure the continued importance of SQL in the new era of Java-based programming.

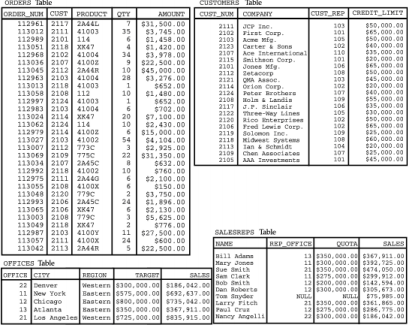
**Chapter 2: A Quick Tour of SQL**

**Overview**

Before diving into the details of SQL, it's a good idea to develop an overall perspective on the language and how it works. This chapter contains a quick tour of SQL that illustrates its major features and functions. The goal of the quick tour is not to make you proficient in writing SQL statements; that is the goal of Part II of this book. Rather, by the time you've finished this chapter, you will have a basic familiarity with the SQL language and an overview of its capabilities.

**A Simple Database**

The examples in the quick tour are based on a simple relational database for a small distribution company. The database, shown in Figure 2-1, stores the information needed to implement a small order processing application. Specifically, it stores the following information:

**Figure 2-1:** A simple relational database

• the *customers* who buy the company's products,

- 15 -

• the *orders* placed by those customers,

• the *salespeople* who sell the products to customers, and

• the *sales offices* where those salespeople work.

This database, like most others, is a model of the "real world." The data stored in the database represents real entities—customers, orders, salespeople, and offices. There is a separate table of data for each different kind of entity. Database requests that you make using the SQL language parallel real-world activities, as customers place, cancel, and change orders, as you hire and fire salespeople, and so on. Let's see how you can use SQL to manipulate data.

**Retrieving Data**

First, let's list the sales offices, showing the city where each one is located and its year to-date sales. The SQL statement that retrieves data from the database is called SELECT. This SQL statement retrieves the data you want:

SELECT CITY, OFFICE, SALES

FROM OFFICES

CITY OFFICE SALES

------------ ------ -----------

Denver 22 $186,042.00

New York 11 $692,637.00

Chicago 12 $735,042.00

Atlanta 13 $367,911.00

Los Angeles 21 $835,915.00

The SELECT statement asks for three pieces of data—the city, the office number, and the sales—for each office. It also specifies that the data comes from the OFFICES table, which stores data about sales offices. The results of the query appear, in tabular form, immediately after the request.

The SELECT statement is used for all SQL queries. For example, here is a query that lists the names and year-to-date sales for each salesperson in the database. It also shows the quota (sales target) and the office number where each person works. In this case, the data comes from SALESREPS table:

SELECT NAME, REP\_OFFICE, SALES, QUOTA

FROM SALESREPS

NAME REP\_OFFICE SALES QUOTA -------------- ---------- ----------- ----------- Bill Adams 13 $367,911.00 $350,000.00 Mary Jones 11 $392,725.00 $300,000.00 Sue Smith 21 $474,050.00 $350,000.00 Sam Clark 11 $299,912.00 $275,000.00 Bob Smith 12 $142,594.00 $200,000.00 Dan Roberts 12 $305,673.00 $300,000.00 Tom Snyder NULL $75,985.00 NULL Larry Fitch 21 $361,865.00 $350,000.00 Paul Cruz 12 $286,775.00 $275,000.00

- 16 -

Nancy Angelli 22 $186,042.00 $300,000.00

SQL also lets you ask for calculated results. For example, you can ask SQL to calculate the amount by which each salesperson is over or under quota:

SELECT NAME, SALES, QUOTA, (SALES - QUOTA)

FROM SALESREPS

NAME SALES QUOTA (SALES-QUOTA) -------------- ----------- ----------- -------------- Bill Adams $367,911.00 $350,000.00 $17,911.00 Mary Jones $392,725.00 $300,000.00 $92,725.00 Sue Smith $474,050.00 $350,000.00 $124,050.00 Sam Clark $299,912.00 $275,000.00 $24,912.00 Bob Smith $142,594.00 $200,000.00 -$57,406.00 Dan Roberts $305,673.00 $300,000.00 $5,673.00 Tom Snyder $75,985.00 NULL NULL Larry Fitch $361,865.00 $350,000.00 $11,865.00 Paul Cruz $286,775.00 $275,000.00 $11,775.00 Nancy Angelli $186,042.00 $300,000.00 -$113,958.00

The requested data (including the calculated difference between sales and quota for each salesperson) once again appears in a row/column table. Perhaps you would like to focus on the salespeople whose sales are less than their quotas. SQL lets you retrieve that kind of selective information very easily, by adding a mathematical comparison to the previous request:

SELECT NAME, SALES, QUOTA, (SALES - QUOTA)

FROM SALESREPS

WHERE SALES < QUOTA

NAME SALES QUOTA (SALES-QUOTA) -------------- ----------- ----------- -------------- Bob Smith $142,594.00 $200,000.00 -$57,406.00 Nancy Angelli $186,042.00 $300,000.00 -$113,958.00

The same technique can be used to list large orders in the database and find out which customer placed the order, what product was ordered, and in what quantity. You can also ask SQL to sort the orders based on the order amount:

SELECT ORDER\_NUM, CUST, PRODUCT, QTY, AMOUNT

FROM ORDERS

WHERE AMOUNT > 25000.00

ORDER BY AMOUNT

ORDER\_NUM CUST PRODUCT QTY AMOUNT

---------- ------------ ---- ----------

112987 2103 4100Y 11 $27,500.00

113069 2109 775C 22 $31,350.00

112961 2117 2A44L 7 $31,500.00

113045 2112 2A44R 10 $45,000.00

**Summarizing Data**

- 17 -

SQL not only retrieves data from the database, it can be used to summarize the database contents as well. What's the average size of an order in the database? This request asks SQL to look at all the orders and find the average amount:

SELECT AVG(AMOUNT)

FROM ORDERS

AVG(AMOUNT)

------------

$8,256.37

You could also ask for the average amount of all the orders placed by a particular customer:

SELECT AVG(AMOUNT)

FROM ORDERS

WHERE CUST = 2103

AVG(AMOUNT)

-----------

$8,895.50

Finally, let's find out the total amount of the orders placed by each customer. To do this, you can ask SQL to group the orders together by customer number and then total the orders for each customer:

SELECT CUST, SUM(AMOUNT)

FROM ORDERS

GROUP BY CUST

CUST SUM(AMOUNT)

----- ------------

2101 $1,458.00

2102 $3,978.00

2103 $35,582.00

2106 $4,026.00

2107 $23,132.00

2108 $7,255.00

2109 $31,350.00

2111 $6,445.00

2112 $47,925.00

2113 $22,500.00

2114 $22,100.00

2117 $31,500.00

2118 $3,608.00

2120 $3,750.00

2124 $3,082.00

**Adding Data to the Database**

SQL is also used to add new data to the database. For example, suppose you just opened a new Western region sales office in Dallas, with target sales of $275,000. Here's the INSERT statement that adds the new office to the database, as office number 23:

- 18 -

INSERT INTO OFFICES (CITY, REGION, TARGET, SALES, OFFICE) VALUES ('Dallas', 'Western', 275000.00, 0.00, 23)

1 row inserted.

Similarly, if Mary Jones (employee number 109) signs up a new customer, Acme Industries, this INSERT statement adds the customer to the database as customer number 2125 with a $25,000 credit limit:

INSERT INTO CUSTOMERS (COMPANY, CUST\_REP, CUST\_NUM, CREDIT\_LIMIT) VALUES ('Acme Industries', 109, 2125, 25000.00)

1 row inserted.

**Deleting Data**

Just as the SQL INSERT statement adds new data to the database, the SQL DELETE statement removes data from the database. If Acme Industries decides a few days later to switch to a competitor, you can delete them from the database with this statement:

DELETE FROM CUSTOMERS

WHERE COMPANY = 'Acme Industries'

1 row deleted.

And if you decide to terminate all salespeople whose sales are less than their quotas, you can remove them from the database with this DELETE statement:

DELETE FROM SALESREPS

WHERE SALES < QT<R

2 rows deleted.

**Updating the Database**

The SQL language is also used to modify data that is already stored in the database. For example, to increase the credit limit for First Corp. to $75,000, you would use the SQL UPDATE statement:

UPDATE CUSTOMERS

SET CREDIT\_LIMIT = 75000.00

WHERE COMPANY = 'First Corp.'

1 row updated.

The UPDATE statement can also make many changes in the database at once. For example, this UPDATE statement raises the quota for all salespeople by $15,000:

UPDATE SALESREPS

SET QUOTA = QUOTA + 15000.00

8 rows updated.

**Protecting Data**

- 19 -

An important role of a database is to protect the stored data from access by unauthorized users. For example, suppose your assistant, named Mary, was not previously authorized to insert data about new customers into the database. This SQL statement grants her that permission:

GRANT INSERT

ON CUSTOMERS

TO MARY

Privilege granted.

Similarly, the following SQL statement gives Mary permission to update data about customers and to retrieve customer data with the SELECT statement:

GRANT UPDATE, SELECT

ON CUSTOMERS

TO MARY

Privilege granted.

If Mary is no longer allowed to add new customers to the database, this REVOKE statement will disallow it:

REVOKE INSERT

ON CUSTOMERS

FROM MARY

Privilege revoked.

Similarly, this REVOKE statement will revoke all of Mary's privileges to access customer data in any way:

REVOKE ALL

ON CUSTOMERS

FROM MARY

Privilege revoked.

**Creating a Database**

Before you can store data in a database, you must first define the structure of the data. Suppose you want to expand the sample database by adding a table of data about the products sold by your company. For each product, the data to be stored includes:

• a three-character manufacturer ID code,

• a five-character product ID code,

• a description of up to thirty characters,

• the price of the product, and

• the quantity currently on hand.

- 20 -

This SQL CREATE TABLE statement defines a new table to store the products data:

CREATE TABLE PRODUCTS

(MFR\_ID CHAR(3),

PRODUCT\_ID CHAR(5),

DESCRIPTION VARCHAR(20),

PRICE MONEY,

QTY\_ON\_HAND INTEGER)

Table created.

Although more cryptic than the previous SQL statements, the CREATE TABLE statement is still fairly straightforward. It assigns the name PRODUCTS to the new table and specifies the name and type of data stored in each of its five columns.

Once the table has been created, you can fill it with data. Here's an INSERT statement for a new shipment of 250 size 7 widgets (product ACI-41007), which cost $225.00 apiece:

INSERT INTO PRODUCTS (MFR\_ID, PRODUCT\_ID, DESCRIPTION, PRICE, QTY\_ON\_HAND)

VALUES ('ACI', '41007', 'Size 7 Widget', 225.00, 250)

1 row inserted.

Finally, if you discover later that you no longer need to store the products data in the database, you can erase the table (and all of the data it contains) with the DROP TABLE statement:

DROP TABLE PRODUCTS

Table dropped.

**Summary**

This quick tour of SQL showed you what SQL can do and illustrated the style of the SQL language, using eight of the most commonly used SQL statements. To summarize:

• SQL is used to *retrieve* data from the database, using the SELECT statement. You can

retrieve all or part of the stored data, sort it, and ask SQL to summarize the data, using totals and averages.

• SQL is used to *update* the database, by adding new data with the INSERT statement,

deleting data with the DELETE statement, and modifying existing data with the UPDATE statement.

• SQL is used to *control access* to the database, by granting and revoking specific privileges for specific users with the GRANT and REVOKE statements.

• SQL is used to *create* the database by defining the structure of new tables and dropping tables when they are no longer needed, using the CREATE and DROP statements.

**Chapter 3: SQL In Perspective**

- 21 -

**Overview**

SQL is both a *de facto* and an official standard language for database management. What does it mean for SQL to be a standard? What role does SQL play as a database language? How did SQL become a standard, and what impact is the SQL standard having on personal computers, local area networks, minicomputers, and mainframes? To answer these questions, this chapter traces the history of SQL and describes its current role in the computer market.

**SQL and Database Management**

One of the major tasks of a computer system is to store and manage data. To handle this task, specialized computer programs known as *database management systems* began to appear in the late 1960s and early 1970s. A database management system, or DBMS, helped computer users to organize and structure their data and allowed the computer system to play a more active role in managing the data. Although database management systems were first developed on large mainframe systems, their popularity has spread to minicomputers, personal computers, workstations, and specialized server computers.

Database management also plays a key role in the explosion of computer networking and the Internet. Early database systems ran on laarge, monolithic computer systems, where the data, the database management software, and the user or application program accessing the database all operated on the same system. The 1980s and 1990s saw the explosion of a new, client/server model for database access, in which a user on a personal computer or an application program accessed a database on a separate computer system using a network. In the late 1990s, the increasing popularity of the Internet and the World Wide Web intertwined the worlds of networking and data management even further. Now users require little more than a web browser to access and interact with databases, not only within their own organizations, but around the world.

Today, database management is very big business. Independent software companies and computer vendors ship billions of dollars worth of database management products every year. Computer industry experts say that mainframe and minicomputer database products each account for about 10 to 20 percent of the database market, and personal computer and server-based database products account for 50 percent or more. Database servers are one of the fastest-growing segments of the computer systems market, driven by database installations on Unix and Windows NT-based servers. Database management thus touches every segment of the computer market.

Since the late 1980s a specific type of DBMS, called a *relational* database management system (RDBMS), has become so popular that it is *the* standard database form. Relational databases organize data in a simple, tabular form and provide many advantages over earlier types of databases. SQL is specifically a relational database language used to work with relational databases.

**A Brief History of SQL**

The history of the SQL language is intimately intertwined with the development of relational databases. Table 3-1 shows some of the milestones in its 30-year history. The relational database concept was originally developed by Dr. E.F. "Ted" Codd, an IBM researcher. In June 1970 Dr. Codd published an article entitled "A Relational Model of Data for Large Shared Data Banks" that outlined a mathematical theory of how data could be stored and manipulated using a tabular structure. Relational databases and SQL trace their origins to this article, which appeared in the *Communications of the Association for Computing Machinery*.

**Table 3-1: Milestones in the Development of SQL**

- 22 -

**Date Event**

1970 Codd defines relational database model

1974 IBM's System/R project begins

1974 First article describing the SEQUEL language

1978 System/R customer tests

1979 Oracle introduces first commercial RDBMS

1981 Relational Technology introduces Ingres

1981 IBM announces SQL/DS

1982 ANSI forms SQL standards committee

1983 IBM announces DB2

1986 ANSI SQL1 standard ratified

1986 Sybase introduces RDBMS for transaction processing

1987 ISO SQL1 standard ratified

1988 Ashton-Tate and Microsoft announce SQL Server for OS/2 1989 First TPC benchmark (TPC-A) published

1990 TPC-B benchmark published

1991 SQL Access Group database access specification published 1992 Microsoft publishes ODBC specification

1992 ANSI SQL2 standard ratified

1992 TPC-C (OLTP) benchmark published

1993 First shipment of specialized SQL data warehousing systems 1993 First shipment of ODBC products

1994 TPC-D (decision support) benchmark published

1994 Commercial shipment of parallel database server technology 1996 Publication of standard API for OLAP database access and OLAP benchmark

- 23 -

1997 IBM DB2 UDB unifies DB2 architecture across IBM and other vendor platforms 1997 Major DBMS vendors announce Java integration strategies

1998 Microsoft SQL Server 7 provides enterprise-level database support for Windows NT

1998 Oracle 8i provides database/Internet integration and moves away from client/server model

**The Early Years**

Codd's article triggered a flurry of relational database research, including a major research project within IBM. The goal of the project, called System/R, was to prove the workability of the relational concept and to provide some experience in actually implementing a relational DBMS. Work on System/R began in the mid-1970s at IBM's Santa Teresa laboratories in San Jose, California.

In 1974 and 1975 the first phase of the System/R project produced a minimal prototype of a relational DBMS. In addition to the DBMS itself, the System/R project included work on database query languages. One of these languages was called SEQUEL, an acronym for Structured English Query Language. In 1976 and 1977 the System/R research prototype

was rewritten from scratch. The new implementation supported multi-table queries and allowed several users to share access to the data.

The System/R implementation was distributed to a number of IBM customer sites for evaluation in 1978 and 1979. These early customer sites provided some actual user experience with System/R and its database language, which, for legal reasons, had been renamed SQL, or Structured Query Language. Despite the name change, the SEQUEL pronunciation remained and continues to this day. In 1979 the System/R research project came to an end, with IBM concluding that relational databases were not only feasible, but could be the basis for a useful commercial product.

**Early Relational Products**

The System/R project and its SQL database language were well-chronicled in technical journals during the 1970s. Seminars on database technology featured debates on the merits of the new and "heretical" relational model. By 1976 it was apparent that IBM was becoming enthusiastic about relational database technology and that it was making a major commitment to the SQL language.

The publicity about System/R attracted the attention of a group of engineers in Menlo Park, California, who decided that IBM's research foreshadowed a commercial market for relational databases. In 1977 they formed a company, Relational Software, Inc., to build a relational DBMS based on SQL. The product, named Oracle, shipped in 1979 and became the first commercially available relational DBMS. Oracle beat IBM's first product to market by a full two years and ran on Digital's VAX minicomputers, which were less expensive than IBM mainframes. Today the company, renamed Oracle Corporation, is a leading vendor of relational database management systems, with annual sales of many billions of dollars.

Professors at the University of California's Berkeley computer laboratories were also researching relational databases in the mid-1970s. Like the IBM research team, they built a prototype of a relational DBMS and called their system Ingres. The Ingres project included a query language named QUEL that, although more "structured" than SQL, was less English-like. Many of today's database experts trace their involvement with relational

- 24 -

databases back to the Berkeley Ingres project, including the founders of Sybase and many of the object-oriented database startup companies.

In 1980 several professors left Berkeley and founded Relational Technology, Inc., to build a commercial version of Ingres, which was announced in 1981. Ingres and Oracle quickly became arch-rivals, but their rivalry helped to call attention to relational database technology in this early stage. Despite its technical superiority in many areas, Ingres became a clear second-place player in the market, competing against the SQL-based capabilities (and the aggressive marketing and sales strategies) of Oracle. The original QUEL query language was effectively replaced by SQL in 1986, a testimony to the market power of the SQL standard. By the mid-1990s, the Ingres technology had been sold to Computer Associates, a leading mainframe software vendor.

**IBM Products**

While Oracle and Ingres raced to become commercial products, IBM's System/R project had also turned into an effort to build a commercial product, named SQL/Data System (SQL/DS). IBM announced SQL/DS in 1981 and began shipping the product in 1982. In 1983 IBM announced a version of SQL/DS for VM/CMS, an operating system that is frequently used on IBM mainframes in corporate "information center" applications.

In 1983 IBM also introduced Database 2 (DB2), another relational DBMS for its mainframe systems. DB2 operated under IBM's MVS operating system, the workhorse operating system used in large mainframe data centers. The first release of DB2 began shipping in 1985, and IBM officials hailed it as a strategic piece of IBM software technology. DB2 has since become IBM's flagship relational DBMS, and with IBM's weight behind it, DB2's SQL language became the *de facto* standard database language. DB2 technology has now migrated across all IBM product lines, from personal computers to network servers to mainframes. In 1997, IBM took the DB2 cross-platform strategy even farther, by announcing DB2 versions for computer systems made by Sun Microsystems, Hewlett-Packard, and other IBM hardware competitors.

**Commercial Acceptance**

During the first half of the 1980s, the relational database vendors struggled for commercial acceptance of their products. The relational products had several disadvantages when compared to the traditional database architectures. The performance of relational databases was seriously inferior to that of traditional databases. Except for the IBM products, the relational databases came from small "upstart" vendors. And, except for the IBM products, the relational databases tended to run on minicomputers rather than on IBM mainframes.

The relational products did have one major advantage, however. Their relational query languages (SQL, QUEL, and others) allowed users to pose *ad hoc* queries to the database— and get immediate answers—without writing programs. As a result, relational databases began slowly turning up in information center applications as decision-support tools. By May 1985 Oracle proudly claimed to have "over 1,000" installations. Ingres was installed in a comparable number of sites. DB2 and SQL/DS were also being slowly accepted and counted their combined installations at slightly over 1,000 sites.

During the last half of the 1980s, SQL and relational databases were rapidly accepted as the database technology of the future. The performance of the relational database products improved dramatically. Ingres and Oracle, in particular, leapfrogged with each new version claiming superiority over the competitor and two or three times the performance of the previous release. Improvements in the processing power of the underlying computer hardware also helped to boost performance.

Market forces also boosted the popularity of SQL in the late 1980s. IBM stepped up its evangelism of SQL, positioning DB2 as the data management solution for the 1990s. Publication of the ANSI/ISO standard for SQL in 1986 gave SQL "official" status as a

- 25 -

standard. SQL also emerged as a standard on Unix-based computer systems, whose popularity accelerated in the 1980s. As personal computers became more powerful and were linked in local area networks, they needed more sophisticated database management. PC database vendors embraced SQL as the solution to these needs, and minicomputer database vendors moved "down market" to compete in the emerging PC local area network market. Through the early 1990s, steadily improving SQL implementations and dramatic improvements in processor speeds made SQL a practical solution for transaction processing applications. Finally, SQL became a key part of the client/server architecture that used PCs, local area networks, and network servers to build much lower cost information processing systems.

SQL's supremacy in the database world has not gone unchallenged. By the early 1990s, object-oriented programming had emerged as the method of choice for applications development, especially for personal computers and their graphical user interfaces. The object model, with its model of objects, classes, methods, and inheritance, did not provide an ideal fit with relational model of tables, rows, and columns of data. A new generation of venture capital-backed "object database" companies sprang up, hoping to make relational databases and their vendors obsolete, just as SQL had done to the earlier, nonrelational vendors. However, SQL and the relational model have more than withstood the challenge to date. Annual revenues for object-oriented databases are measured in the hundreds of millions of dollars, at best, while SQL and relational database systems, tools, and services produce tens of billions of dollars.

As SQL grew to address an ever-wider variety of data management tasks, the "one-size fits-all" approach showed serious strain. By the late 1990s, "database management" was no longer a monolithic market. Specialized database systems sprang up to support different market needs. One of the fastest-growing segments was "data warehousing," where databases were used to search through huge amounts of data to discover underlying trends and patterns. A second major trend was the incorporation of new data types (such as multimedia data) and object-oriented principles into SQL. A third important segment was "mobile databases" for portable personal computers that could operate when sometimes connected to, and sometimes disconnected from, a centralized database system. Despite the emergence of database market subsegments, SQL has remained a common denominator across them all. As the computer industry prepares for the next century, SQL's dominance as *the* database standard is as strong as ever.

**SQL Standards**

One of the most important developments in the market acceptance of SQL is the emergence of SQL standards. References to "the SQL standard" usually mean the official standard adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). However, there are other important SQL standards, including the *de facto* standard SQL defined by IBM's DB2 product family.

**The ANSI/ISO Standards**

Work on the official SQL standard began in 1982, when ANSI charged its X3H2 committee with defining a standard relational database language. At first the committee debated the merits of various proposed database languages. However, as IBM's commitment to SQL increased and SQL emerged as a *de facto* standard in the market, the committee selected SQL as their relational database language and turned their attention to standardizing it.

The resulting ANSI standard for SQL is largely based on DB2 SQL, although it contains some major differences from DB2. After several revisions, the standard was officially adopted as ANSI standard X3.135 in 1986, and as an ISO standard in 1987. The ANSI/ISO standard has since been adopted as a Federal Information Processing Standard (FIPS) by the U.S. government. This standard, slightly revised and expanded in 1989, is usually called the "SQL-89" or "SQL1" standard.

- 26 -

Many of the ANSI and ISO standards committee members were representatives from database vendors who had existing SQL products, each implementing a slightly different SQL dialect. Like dialects of human languages, the SQL dialects were generally very similar to one another but were incompatible in their details. In many areas the committee simply sidestepped these differences by omitting some parts of the language from the standard and specifying others as "implementor-defined." These decisions allowed existing SQL implementations to claim broad adherence to the resulting ANSI/ISO standard but made the standard relatively weak.

To address the holes in the original standard, the ANSI committee continued its work, and drafts for a new more rigorous SQL2 standard were circulated. Unlike the 1989 standard, the SQL2 drafts specified features considerably beyond those found in current commercial SQL products. Even more far-reaching changes were proposed for a follow on SQL3 standard. In addition, the draft standards attempted to officially standardize parts of the SQL language where different "proprietary standards" had long since been set by the various major DBMS brands. As a result, the proposed SQL2 and SQL3 standards were a good deal more controversial than the initial SQL standard. The SQL2 standard weaved its way through the ANSI approval process and was finally approved in October, 1992. While the original 1986 standard took less than 100 pages, the SQL2 standard (officially called "SQL-92") takes nearly 600 pages.

The SQL2 standards committee acknowledged the large step from SQL1 to SQL2 by explicitly creating three levels of SQL2 standards compliance. The lowest compliance level ("Entry-Level") requires only minimal additional capability beyond the SQL-89 standard. The middle compliance level ("Intermediate-Level") was created as an achievable major step beyond SQL-89, but one that avoids the most complex and most system-dependent and DBMS brand-dependent issues. The third compliance level ("Full") requires a full implementation of all SQL2 capabilities. Throughout the 600 pages of the standard, each description of each feature includes a definition of the specific aspects of that feature which must be supported in order to achieve Entry, Intermediate, or Full compliance.

Despite the existence of a SQL2 standard, no commercial SQL product available today implements all of its features, and no two commercial SQL products support exactly the same SQL dialect. Moreover, as database vendors introduce new capabilities, they are expanding their SQL dialects and moving them even further apart. The central core of the

SQL language has become fairly standardized, however. Where it could be done without hurting existing customers or features, vendors have brought their products into conformance with the SQL-89 standard, and the same will slowly happen with SQL2. In the meantime, work continues on standards beyond SQL2. The "SQL3" effort effectively fragmented into separate standardization efforts and focused on different extensions to SQL. Some of these, such as stored procedure capabilities, are already found in many commercial SQL products and pose the same standardization challenges faced by SQL2. Others, such as proposed object extensions to SQL, are not yet widely available or fully implemented, but have generated a great deal of controversy. With most vendors far from fully implementing SQL2 capabilities, and with the diversity of SQL extensions now available in commercial products, work on SQL3 has taken on less commercial importance.

The "real" SQL standard, of course, is the SQL implemented in products that are broadly accepted by the marketplace. For the most part, programmers and users tend to stick with those parts of the language that are fairly similar across a broad range of products. The innovation of the database vendors continues to drive the invention of new SQL capabilities; some products remain years later only for backward compatibility, and some find commercial success and move into the mainstream.

**Other SQL Standards**

Although it is the most widely recognized, the ANSI/ISO standard is not the only standard for SQL. X/OPEN, a European vendor group, has also adopted SQL as part of its suite of standards for a "portable application environment" based on Unix. The X/OPEN

- 27 -

standards play a major role in the European computer market, where portability among computer systems from different vendors is a key concern. Unfortunately, the X/OPEN standard differs from the ANSI/ISO standard in several areas.

IBM also included SQL in the specification of its bold Systems Application Architecture (SAA) blueprint, promising that all of its SQL products would eventually move to this SAA SQL dialect. Although SAA failed to achieve its promise of unifying the IBM product line, the momentum toward a unified IBM SQL continued. With its mainframe DB2 database as the flagship, IBM introduced DB2 implementations for OS/2, its personal computer operating system, and for its RS/6000 line of Unix-based workstations and servers. By 1997, IBM had moved DB2 beyond its own product line and shipped versions of DB2- Universal Database for systems made by rival manufacturers Sun Microsystems, Hewlett-Packard, and Silicon Graphics, and for Windows NT. With IBM's historical leadership in relational database technology, the SQL dialect supported by DB2 version is a very powerful *de facto* standard.

**ODBC and the SQL Access Group**

An important area of database technology not addressed by official standards is *database interoperability*—the methods by which data can be exchanged among different databases, usually over a network. In 1989, a group of vendors formed the SQL Access Group to address this problem. The resulting SQL Access Group specification for Remote Database Access (RDA) was published in 1991. Unfortunately, the RDA specification is closely tied to the OSI protocols, which have not been widely accepted, so it has had little impact. Transparent interoperability among different vendors' databases remains an elusive goal.

A second standard from the SQL Access Group has had far more market impact. At Microsoft's urging and insistence, SQL Access Group expanded its focus to include a call-level interface for SQL. Based on a draft from Microsoft, the resulting Call-Level Interface (CLI) specification was published in 1992. Microsoft's own Open Database Connectivity (ODBC) specification, based on the CLI standard, was published the same year. With the market power of Microsoft behind it, and the "open standards" blessing of SQL Access Group, ODBC has emerged as the *de facto* standard interface for PC access to SQL databases. Apple and Microsoft announced an agreement to support ODBC on Macintosh and Windows in the spring of 1993, giving ODBC "standard" status in both popular graphical user interface environments. ODBC implementations for Unix based systems soon followed.

Today, ODBC is in its fourth major revision as a cross-platform database access standard. ODBC support is available for all major DBMS brands. Most packaged application programs that have database access as an important part of their capabilities support ODBC, range from multi-million dollar enterprise class applications like Enterprise Resource Planning (ERP) and Supply Chain Management (SCM) to PC applications such as spreadsheets, query tools, and reporting programs. Microsoft's focus has moved beyond ODBC to higher-level interfaces (such as OLE/DB) and more recently to ADO (Active Data Objects), but these new interfaces are layered on top of ODBC for relational database access, and it remains a key cross-platform database access technology.

**The Portability Myth**

The existence of published SQL standards has spawned quite a few exaggerated claims about SQL and applications portability. Diagrams such as the one in Figure 3-1 are frequently drawn to show how an application using SQL can work interchangeably with any SQL-based database management system. In fact, the holes in the SQL-89 standard and the current differences between SQL dialects are significant enough that an application must *always* be modified when moved from one SQL database to another. These differences, many of which were eliminated by the SQL2 standard but have not yet implemented in commercial products, include:

- 28 -



**Figure 3-1:** The SQL portability myth

• *Error codes.* The SQL-89 standard does not specify the error codes to be returned

when SQL detects an error, and all of the commercial implementations use their own set of error codes. The SQL2 standard specifies standard error codes.

• *Data types.* The SQL-89 standard defines a minimal set of data types, but it omits some of the most popular and useful types, such as variable-length character strings,

dates and times, and money data. The SQL2 standard addresses these, but not "new" data types such as graphics and multimedia objects.

• *System tables.* The SQL-89 standard is silent about the system tables that provide information regarding the structure of the database itself. Each vendor has its own

structure for these tables, and even IBM's four SQL implementations differ from one another. The tables are standardized in SQL2, but only at the higher levels of compliance, which are not yet provided by most vendors.

• *Interactive SQL.* The standard specifies only the *programmatic* SQL used by an application program, not interactive SQL. For example, the SELECT statement used to

query the database in interactive SQL is absent from the SQL-89 standard. Again, the SQL2 standard addressed this issue, but long after all of the major DBMS vendors had well-established interactive SQL capabilities.

• *Programmatic interface.* The original standard specifies an abstract technique for using SQL from within an applications program written in COBOL, C, FORTRAN, and other programming languages. No commercial SQL product uses this technique, and

there is considerable variation in the actual programmatic interfaces used. The SQL2 standard specifies an embedded SQL interface for popular programming languages but not a call-level interface.

• *Dynamic SQL.* The SQL-89 standard does not include the features required to develop general-purpose database front-ends, such as query tools and report writers. These features, known as *dynamic SQL*, are found in virtually all SQL database systems, but

they vary significantly from product to product. SQL2 includes a standard for dynamic SQL, but with hundreds of thousands of existing applications dependent on backward compatibility, DBMS vendors have not implemented it.

• *Semantic differences.* Because the standards specify certain details as "implementor defined," it's possible to run the same query against two different conforming SQL

implementations and produce two different sets of query results. These differences occur in the handling of NULL values, column functions, and duplicate row elimination.

• *Collating sequences.* The SQL-89 standard does not address the collating (sorting) sequence of characters stored in the database. The results of a sorted query will be different if the query is run on a personal computer (with ASCII characters) and a

mainframe (with EBCDIC characters). The SQL2 standard includes an elaborate specification for how a program or a user can request a specific collating sequence, but it is an advanced-level feature that is not typically supported in commercial products.

- 29 -

• *Database structure.* The SQL-89 standard specifies the SQL language to be used once a particular database has been opened and is ready for processing. The details

of database naming and how the initial connection to the database is established vary widely and are not portable. The SQL2 standard creates more uniformity but cannot completely mask these details.

Despite these differences, commercial database tools boasting portability across several different brands of SQL databases began to emerge in the early 1990s. In every case, however, the tools require a special adapter for each supported DBMS, which generates the appropriate SQL dialect, handles data type conversion, translates error codes, and so on. Transparent portability across different DBMS brands based on standard SQL is the major goal of SQL2 and ODBC, and significant progress has been made. Today, virtually all programs that support multiple databases include specific "drivers" for communicating with each of the major DBMS brands, and usually include an ODBC driver for accessing the others.

**SQL and Networking**

The dramatic growth of computer networking over the past decade has had a major impact on database management and given SQL a new prominence. As networks became more common, applications that traditionally ran on a central minicomputer or mainframe moved to local area networks of desktop workstations and servers. In these networks SQL plays a crucial role as the link between an application running on a desktop workstation with a graphical user interface and the DBMS that manages shared data on a cost-effective server. More recently, the exploding popularity of the Internet and the World Wide Web has reinforced the network role for SQL. In the emerging "three-tier" Internet architecture, SQL once again provides the link between the application logic (now running in the "middle tier," on an application server or web server) and the database residing in the "back-end" tier. The next few sections in this chapter discuss the evolution of database network architectures and the role of SQL in each one.

**Centralized Architecture**

The traditional database architecture used by DB2, SQL/DS, and the original minicomputer databases such as Oracle and Ingres is shown in Figure 3-2. In this architecture the DBMS and the physical data both reside on a central minicomputer or mainframe system, along with the application program that accepts input from the user's terminal and displays data on the user's screen. The application program communicates with the DBMS using SQL.



**Figure 3-2:** Database management in a centralized architecture

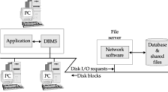
Suppose that the user types a query that requires a sequential search of a database, such as a request to find the average amount of merchandise of all orders. The DBMS receives the query, scans through the database fetching each record of data from the disk, calculates the average, and displays the result on the terminal screen. Both the application processing and the database processing occur on the central computer, so execution of this type of query (and in fact, all kinds of queries) is very efficient.

The disadvantage of the centralized architecture is scalability. As more and more users are added, each of them adds application processing workload to the system. Because the system is shared, each user experiences degraded performance as the system becomes more heavily loaded.

- 30 -

**File Server Architecture**

The introduction of personal computers and local area networks led to the development of the *file server* architecture, shown in Figure 3-3. In this architecture, an application running on a personal computer can transparently access data located on a file server, which stores shared files. When a PC application requests data from a shared file, the networking software automatically retrieves the requested block of the file from the server. Early PC databases, such as dBASE and later Microsoft's Access, supported this file server approach, with each personal computer running its own copy of the DBMS software.



**Figure 3-3:** Database management in a file server architecture

For typical queries that retrieve only one row or a few rows from the database, this architecture provides excellent performance, because each user has the full power of a personal computer running its own copy of the DBMS. However, consider the query made in the previous example. Because the query requires a sequential scan of the database, the DBMS repeatedly requests blocks of data from the database, which is physically located across the network on the server. Eventually *every* block of the file will be requested and sent across the network. Obviously this architecture produces very heavy network traffic and slow performance for queries of this type.

**Client/Server Architecture**

Figure 3-4 shows the next stage of network database evolution—the *client/server* database architecture. In this scheme, personal computers are combined in a local area network with a *database server* that stores shared databases. The functions of the DBMS are split into two parts. Database "front-ends," such as interactive query tools, report writers, and application programs, run on the personal computer. The back-end database engine that stores and manages the data runs on the server. As the client/server architecture grew in popularity during the 1990s, SQL became the standard database language for communication between the front-end tools and the back-end engine in this architecture.



**Figure 3-4:** Database management in a client/server architecture

Consider once more the query requesting the average order size. In the client/server architecture, the query travels across the network to the database server as a SQL

- 31 -

request. The database engine on the server processes the request and scans the database, which also resides on the server. When the result is calculated, the database engine sends it back across the network as a single reply to the initial request, and the front-end application displays it on the PC screen.

The client/server architecture reduces the network traffic and splits the database workload. User-intensive functions, such as handling input and displaying data, are concentrated on the user's PC. Data-intensive functions, such as file I/O and query processing, are concentrated in the database server. Most importantly, the SQL language provides a well-defined interface between the front-end and back-end systems, communicating database access requests in an efficient manner.

By the mid-1990s, these advantages made the client/server architecture the most popular scheme for implementing new applications. All of the most popular DBMS products— Oracle, Informix, Sybase, SQL Server, DB2, and many more—offered client/server capability. The database industry grew to include many companies offering tools for building client/server applications. Some of these came from the database companies themselves; others came from independent companies.

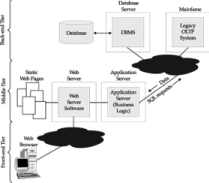
Like all architectures, client/server had its disadvantages. The most serious of these was the problem of managing the applications software that was now distributed across hundreds or thousands of desktop PCs instead of running on a central minicomputer or mainframe. To update an application program in a large company, the information systems department had to update thousands of PC systems, one at a time. The situation was even worse if changes to the application program had to be synchronized with changes to other applications, or to the DBMS system itself. In addition, with personal computers on user's desks, users tended to add new personal software of their own or to change the configuration of their systems. Such changes often disrupted existing applications, adding to the support burden. Companies developed strategies to deal with these issues, but by the late 1990s there was growing concern about the manageability of client/server applications on large, distributed PC networks.

**Multi-Tier Architecture**

With the emergence of the Internet and especially the World Wide Web, network database architecture has taken another step. At first, the Web was used to access ("browse") static documents and evolved outside of the database world. But as the use of web browsers became widespread, it wasn't long before companies thought about using them as a simple way to provide access to corporate databases as well. For example, suppose a company starts using the Web to provide product information to its customers, by making product descriptions and graphics available on its web site. A natural next step is to give customers access to current product availability information through the same web browser interface. This requires linking the web server to the database system that stores the (constantly changing) current product inventory levels.

The methods used to link web servers and DBMS systems have evolved rapidly over the last several years and have converged on the three-tier network architecture shown in Figure 3-5. The user interface is a web browser running on a PC or some other "thin client" device in the "front" tier. It communicates with a web server in the "middle tier." When the user request is for something more complex than a simple web page, the web server passes the request to an *application server* whose role is to handle the business logic required to process the request. Often the request will involve access to an existing ("legacy") application running on a mainframe system or to a corporate database. These systems run in the "back" tier of the architecture. As with the client/server architecture, SQL is solidly entrenched as the standard database language for communicating between the application server and back-end databases. All of the packaged application server products provide a SQL-based callable API for database access.

- 32 -



**Figure 3-5:** Database management in a three-tier Internet architecture **The Proliferation of SQL**

As the standard for relational database access, SQL has had a major impact on all parts of the computer market. IBM has adopted SQL as a unifying database technology for its product line. SQL-based databases dominate the market for Unix-based computer systems. In the PC market, SQL databases on Windows NT are mounting a serious challenge to the dominance of Unix as a database processing platform, especially for departmental applications. SQL is accepted as a technology for online transaction processing, fully refuting the conventional wisdom of the 1980s that relational databases would never offer performance good enough for transaction processing applications. SQL-based data warehousing and data mining applications are helping companies to discover customer purchase patterns and offer better products and services. On the Internet, SQL-based databases are the foundation of more personalized products, services, and information services that are a key benefit of electronic commerce.

**SQL and IBM's Unified Database Strategy**

SQL plays a key role as the database access language that unifies IBM's multiple incompatible computer families. Originally, this role was part of IBM's Systems Application Architecture (SAA) strategy, announced in March 1987. Although IBM's grand goals for SAA were not achieved, the unifying role of SQL has grown even more important over time. The DB2 database system, IBM's flagship SQL-based DBMS, now runs on a broad range of IBM and non-IBM computer systems, including:

• *Mainframes.* DB2 started as the SQL standard-bearer for IBM mainframes running

MVS and has now replaced SQL/DS as the relational system for the VM and VSE mainframe operating systems.

• *AS/400.* This SQL implementation runs on IBM's family of midrange business systems, targeted at small- and medium-sized businesses and server applications.

• *RS/6000.* DB2 runs under the Unix operating system on IBM's family of RISC-based

workstations and servers, for engineering and scientific applications and as IBM's own Unix database server platform.

• *Other Unix platforms.* IBM supports DB2 on Unix-based server platforms from Sun

Microsystems and Hewlett-Packard, the two largest Unix system vendors, and on Unix-based workstations from Silicon Graphics.

• *OS/2.* A smaller-scale version of DB2 runs on this IBM-proprietary operating system for Intel-based personal computers

- 33 -

• *Windows NT*. A PC-LAN server version of DB2 competes with Microsoft SQL Server,  Oracle, and others on this fast-growing database server platform.

**SQL on Minicomputers**

Minicomputers were one of the most fertile early markets for SQL-based database systems. Oracle and Ingres were both originally marketed on Digital's VAX/VMS minicomputer systems. Both products have since been ported to many other platforms. Sybase, a later database system specialized for online transaction processing, also targeted the VAX as one of its primary platforms.

Through the 1980s, the minicomputer vendors also developed their own proprietary relational databases featuring SQL. Digital considered relational databases so important that it bundled a run-time version of its Rdb/VMS database with every VAX/VMS system. Hewlett-Packard offered Allbase, a database that supported both its HPSQL dialect and a nonrelational interface. Data General's DG/SQL database replaced its older nonrelational databases as DG's strategic data management tool. In addition, many of the minicomputer vendors resold relational databases from the independent database software vendors. These efforts helped to establish SQL as an important technology for midrange computer systems.

Today, the minicomputer vendors' SQL products have largely disappeared, beaten in the marketplace by multi-platform software from Oracle, Informix, Sybase, and others. Accompanying this trend, the importance of proprietary minicomputer operating systems has faded as well, replaced by widespread use of Unix on midrange systems. Yesterday's minicomputer SQL market has effectively become today's market for Unix based database servers based on SQL.

**SQL on Unix-Based Systems**

SQL has firmly established itself as the data management solution of choice for Unix based computer systems. Originally developed at Bell Laboratories, Unix became very popular in the 1980s as a vendor-independent, standard operating system. It runs on a wide range of computer systems, from workstations to mainframes, and has become the standard operating system for scientific and engineering applications.

In the early 1980s four major databases were already available for Unix systems. Two of them, Ingres and Oracle, were Unix versions of the products that ran on DEC's proprietary minicomputers. The other two, Informix and Unify, were written specifically for Unix. Neither of them originally offered SQL support, but by 1985 Unify offered a SQL query language, and Informix had been rewritten as Informix-SQL, with full SQL support.

Today, Oracle, Informix, and Sybase dominate the Unix-based database market and are available on all of the leading Unix systems. Unix-based database servers are a mainstream building block for both client/server and three-tier Internet architectures. The constant search for higher SQL database performance has driven some of the most important trends in Unix system hardware. These include the emergence of symmetric multiprocessing (SMP) as a mainstream server architecture, and the use of RAID (Redundant Array of Independent Disk) technology to boost I/O performance.

**SQL on Personal Computers**

Databases have been popular on personal computers since the early days of the IBM PC. Ashton-Tate's dBASE product reached an installed base of over one million MS DOS-based PCs. Although these early PC databases often presented data in tabular form, they lacked the full power of a relational DBMS and a relational database language such as SQL. The first SQL-based PC databases were versions of popular minicomputer products that barely fit on personal computers. For example, Professional Oracle for the IBM PC, introduced in 1984, required two megabytes of memory—well above the typical

- 34 -

640KB PC configuration of the day.

The real impact of SQL on personal computers began with the announcement of OS/2 by IBM and Microsoft in April 1987. In addition to the standard OS/2 product, IBM announced a proprietary OS/2 Extended Edition (OS/2 EE) with a built-in SQL database and communications support. With the introduction, IBM again signaled its strong commitment to SQL, saying in effect that SQL was so important that it belonged in the computer's operating system.

OS/2 Extended Edition presented Microsoft with a problem. As the developer and distributor of standard OS/2 to other personal computer manufacturers, Microsoft needed an alternative to the Extended Edition. Microsoft responded by licensing the Sybase DBMS, which had been developed for VAX, and began porting it to OS/2. In January 1988, in a surprise move, Microsoft and Ashton-Tate (the PC database leader at the time with its dBASE product) announced that they would jointly sell the resulting OS/2-based product, renamed SQL Server. Microsoft would sell SQL Server with OS/2 to computer manufacturers; Ashton-Tate would sell the product through retail channels to PC users. In September 1989, Lotus Development (the other member of the "big three" of PC software at the time) added its endorsement of SQL Server by investing in Sybase. Later that year, Ashton-Tate relinquished its exclusive retail distribution rights and sold its investment to Lotus.

SQL Server for OS/2 met with only limited success. But in typical Microsoft fashion, Microsoft continued to invest heavily in SQL Server development and ported it to its Windows NT operating system. For a while, Microsoft and Sybase remained partners, with Sybase focused on the minicomputer and Unix-based server markets and Microsoft focused on PC local area networks (LANs) and Windows NT. As Windows NT and Unix systems became more and more competitive as database server operating system platforms, the relationship became less cooperative and more competitive. Eventually, Sybase and Microsoft went their separate ways. The common heritage of Sybase's and Microsoft's SQL products can still be seen in product capabilities and some common SQL extensions (for example, stored procedures), but the product lines have already diverged significantly.

Today SQL Server is a major database system on Windows NT. SQL Server 7.0, which shipped in late 1998, provided a significant step up in the size and scale of database applications that SQL Server can support. In addition to SQL Server's impact, the availability of Oracle, Informix, DB2, and other mainstream DBMS products has helped Windows NT to steadily make inroads into Unix's dominance as a database server platform. While Unix continues to dominate the largest database server installations, Windows NT and the Intel architecture systems on which it runs have achieved credibility in the midrange market.

**SQL and Transaction Processing**

SQL and relational databases originally had very little impact in online transaction processing (OLTP) applications. With their emphasis on queries, relational databases were confined to decision support and low volume online applications, where their slower performance was not a disadvantage. For OLTP applications, where hundreds of users needed online access to data and subsecond response times, IBM's nonrelational Information Management System (IMS) reigned as the dominant DBMS.

In 1986 a new DBMS vendor, Sybase, introduced a new SQL-based database especially designed for OLTP applications. The Sybase DBMS ran on VAX/VMS minicomputers and Sun workstations and focused on maximum online performance. Oracle Corporation and

Relational Technology followed shortly with announcements that they, too, would offer OLTP versions of their popular Oracle and Ingres database systems. In the Unix market, Informix announced an OLTP version of its DBMS, named Informix-Turbo.

In 1988 IBM jumped on the relational OLTP bandwagon with DB2 Version 2, with benchmarks showing the new version operating at over 250 transactions per second on

- 35 -

large mainframes. IBM claimed that DB2 performance was now suitable for all but the most demanding OLTP applications, and encouraged customers to consider it as a serious alternative to IMS. OLTP benchmarks have now become a standard sales tool for relational databases, despite serious questions about how well the benchmarks actually measure performance in real applications.

The suitability of SQL for OLTP improved dramatically through the 1990s, with advances in relational technology and more powerful computer hardware both leading to ever higher transaction rates. DBMS vendors started to position their products based on their OLTP performance, and for a few years database advertising focused almost entirely on these "performance benchmark wars." A vendor-independent organization, the Transaction Processing Council, jumped into the benchmarking fray with a series of vendor-independent benchmarks (TPC-A, TPC-B, and TPC-C), which only served to intensify the performance focus of the vendors.

By the late 1990s, SQL-based relational databases on high-end Unix-based database servers had passed the 1,000 transactions per second mark. Client/server systems using SQL databases have become the accepted architecture for implementing OLTP applications. From a position as "unsuitable for OLTP," SQL has grown to be the industry standard foundation for building OLTP applications.

**SQL and Workgroup Databases**

The dramatic growth of PC LANs through the 1980s and 1990s created a new opportunity for departmental or "workgroup" database management. The original database systems focused on this market segment ran on IBM's OS/2 operating system. In fact, SQL Server, now a key part of Microsoft's Windows strategy, originally made its debut as an OS/2 database product. In the mid-1990s, Novell also made a concentrated effort to make its NetWare operating system an attractive workgroup database server platform. From the earliest days of PC LANs, NetWare had become established as the dominant network operating system for file and print servers. Through deals with Oracle and others, Novell sought to extend this leadership to workgroup database servers as well.

The arrival of Windows NT on the workgroup computing scene was the catalyst that caused the workgroup database market to really take off. While NetWare offered a clear performance advantage over NT as a workgroup file server, NT had a more robust, general-purpose architecture, more like the minicomputer operating systems. Microsoft successfully positioned NT as a more attractive platform for running workgroup applications (as an "application server") and workgroup databases. Microsoft's own SQL Server product was marketed (and often bundled) with NT as a tightly integrated workgroup database platform. Corporate information systems departments were at first very cautious about using relatively new and unproven technology, but the NT/SQL Server combination allowed departments and non-IS executives to undertake smaller scale, workgroup-level projects on their own, without corporate IS help. This phenomenon, like the grass roots support for personal computers a decade earlier, fueled the early growth of the workgroup database segment.

Today, SQL is well established as a workgroup database standard. Microsoft's SQL Server has been joined by Oracle, Informix, Sybase, DB2, and many other DBMS brands running on the Windows NT/Windows 2000 platform. Windows-based SQL databases are the second largest segment of the DBMS market and are the fastest growing. From this solid dominance in the workgroup segment, Windows-based server systems are mounting a continued assault on enterprise-class database applications, slowly but surely eating into low-end Unix-based database deployments.

**SQL and Data Warehousing**

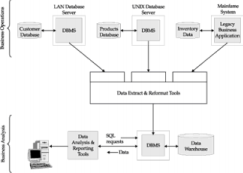
For several years, the effort to make SQL a viable technology for OLTP applications shifted the focus away from the original relational database strengths of query processing and decision making. Performance benchmarks and competition among the major DBMS

- 36 -

brands focused on simple transactions like adding a new order to the database or determining a customer's account balance. Because of the power of the relational database model, the databases that companies used to handle daily business operations could also be used to analyze the growing amounts of data that were being accumulated. A frequent theme of conferences and trade show speeches for IS managers was that a corporation's accumulated data (stored in SQL databases, of course) should be treated as a valuable "asset" and used to help improve the quality of business decision-making.

Although relational databases could, in theory, easily perform both OLTP and decision making applications, there were some very significant practical problems. OLTP workloads consisted of many short database transactions, and the response time for users was very important. In contrast, decision-support queries could involve sequential scans of large database tables to answer questions like "What is the average order size by sales region?" or "How do inventory trends compare with the same time a year ago?" These queries could take minutes or hours. If a business analyst tried to run one of these queries during a time when business transaction volumes reached their peak, it could cause serious degradation in OLTP performance. Another problem was that the data to answer useful questions about business trends was often spread across many different databases, typically involving different DBMS vendors and different computer platforms.

The desire to take advantage of accumulated business data, and the practical performance problems it caused for OLTP applications, led to a new database trend called "data warehousing." The idea of the data warehouse is shown in Figure 3-6. Business data is extracted from OLTP systems, reformatted and validated as necessary, and then placed into a separate database that is dedicated to decision-making queries (the "warehouse"). The data extraction and transformation can be scheduled for off-hours batch processing. Ideally, only new or changed data can be extracted, minimizing the amount of data to be processed in the monthly, weekly, or daily warehouse "refresh" cycle. With this scheme, the time-consuming business analysis queries use the data warehouse, not the OLTP database, as their source of data.



**Figure 3-6:** The data warehousing concept

SQL-based relational databases were a clear choice for the warehouse data store because of their flexible query processing. A series of new companies was formed to build the data extraction, transformation, and database query tools needed by the data warehouse model. In addition, DBMS vendors started to focus on the kinds of database queries that customers tended to run in the data warehouse. These queries tended to be large and complex—such as analyzing tens or hundreds of millions of individual cash register receipts to look for product purchase patterns. They often involved time-series data—for example, analyzing product sales or market share data over time. They also tended to involve statistical summaries of data—total sales, average order volume, percent growth, and so on—rather than the individual data items themselves.

- 37 -

To address the specialized needs of data warehousing applications (often called "Online Analytical Processing" or OLAP), specialized databases began to appear. These databases were optimized for OLAP workloads in several different ways. Their performance was tuned for complex, read-only query access. They supported advanced statistical and other data functions, such as built-in time-series processing. They supported precalculation of database statistical data, so that retrieving averages and totals could be dramatically faster. Some of these specialized databases did not use SQL, but many did (leading to the companion term "ROLAP," for Relational Online Analytic Processing). As with so many segments of the database market, SQL's advantages as a standard proved to be a powerful force. Data warehousing has become a one-billion-dollar plus segment of the database market, and SQL-based databases are firmly entrenched as the mainstream technology for building data warehouses.

**Summary**

This chapter described the development of SQL and its role as a standard language for relational database management:

• SQL was originally developed by IBM researchers, and IBM's strong support of SQL is a key reason for its success.

• There are official ANSI/ISO SQL standards and several other SQL standards, each slightly different from the ANSI/ISO standards.

• Despite the existence of standards, there are many small variations among commercial SQL dialects; no two SQLs are exactly the same.

• SQL has become the standard database management language across a broad range

of computer systems and applications areas, including mainframes, workstations,

personal computers, OLTP systems, client/server systems, data warehousing, and the Internet.

**Chapter 4: Relational Databases**

**Overview**

Database management systems organize and structure data so that it can be retrieved and manipulated by users and application programs. The data structures and access techniques provided by a particular DBMS are called its *data model*. A data model determines both the "personality" of a DBMS and the applications for which it is particularly well suited.

SQL is a database language for relational databases that uses the *relational data model*. What exactly is a relational database? How is data stored in a relational database? How do relational databases compare to earlier technologies, such as hierarchical and network databases? What are the advantages and disadvantages of the relational model? This chapter describes the relational data model supported by SQL and compares it to earlier strategies for database organization.

**Early Data Models**

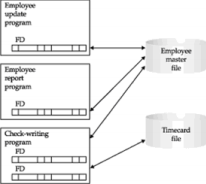
As database management became popular during the 1970s and 1980s, a handful of popular data models emerged. Each of these early data models had advantages and disadvantages that played key roles in the development of the relational data model. In many ways the relational data model represented an attempt to streamline and simplify the earlier data models. In order to understand the role and contribution of SQL and the relational model, it is useful to briefly examine some data models that preceded the development of SQL.

- 38 -

**File Management Systems**

Before the introduction of database management systems, all data permanently stored on a computer system, such as payroll and accounting records, was stored in individual files. A *file management system*, usually provided by the computer manufacturer as part of the computer's operating system, kept track of the names and locations of the files. The file management system basically had no data model; it knew nothing about the internal contents of files. To the file management system, a file containing a word processing document and a file containing payroll data appeared the same.

Knowledge about the contents of a file—what data it contained and how the data was organized—was embedded in the application programs that used the file, as shown in Figure 4-1. In this payroll application, each of the COBOL programs that processed the employee master file contained a *file description* (FD) that described the layout of the data in the file. If the structure of the data changed—for example, if an additional item of data was to be stored for each employee—every program that accessed the file had to be modified. As the number of files and programs grew over time, more and more of a data processing department's effort went into maintaining existing applications rather than developing new ones.



**Figure 4-1:** A payroll application using a file management system

The problems of maintaining large file-based systems led in the late 1960s to the development of database management systems. The idea behind these systems was simple: take the definition of a file's content and structure out of the individual programs, and store it, together with the data, in a database. Using the information in the database, the DBMS that controlled it could take a much more active role in managing both the data and changes to the database structure.

**Hierarchical Databases**

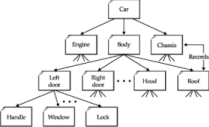
One of the most important applications for the earliest database management systems was production planning for manufacturing companies. If an automobile manufacturer decided to produce 10,000 units of one car model and 5,000 units of another model, it

needed to know how many parts to order from its suppliers. To answer the question, the product (a car) had to be decomposed into assemblies (engine, body, chassis), which were decomposed into subassemblies (valves, cylinders, spark plugs), and then into sub subassemblies, and so on. Handling this list of parts, known as a *bill of materials*, was a job tailor-made for computers.

The bill of materials for a product has a natural hierarchical structure. To store this data,

- 39 -

the *hierarchical* data model, illustrated in Figure 4-2, was developed. In this model, each *record* in the database represented a specific part. The records had *parent/child relationships*, linking each part to its subpart, and so on.



**Figure 4-2:** A hierarchical bill-of-materials databse

To access the data in the database, a program could:

• find a particular part by number (such as the left door),

• move "down" to the first child (the door handle),

• move "up" to its parent (the body), or

• move "sideways" to the next child (the right door).

Retrieving the data in a hierarchical database thus required *navigating* through the records, moving up, down, and sideways one record at a time.

One of the most popular hierarchical database management systems was IBM's Information Management System (IMS), first introduced in 1968. The advantages of IMS and its hierarchical model follow.

• *Simple structure.* The organization of an IMS database was easy to understand. The database hierarchy paralleled that of a company organization chart or a family tree.

• *Parent/child organization.* An IMS database was excellent for representing parent/child relationships, such as "A is a part of B" or "A is owned by B."

• *Performance.* IMS stored parent/child relationships as physical pointers from one data

record to another, so that movement through the database was rapid. Because the

structure was simple, IMS could place parent and child records close to one another on the disk, minimizing disk input/output.

IMS is still a very widely used DBMS on IBM mainframes. Its raw performance makes it the database of choice in high-volume transaction processing applications such as processing bank ATM transactions, verifying credit card numbers, and tracking the delivery of overnight packages. Although relational database performance has improved dramatically over the last decade, the performance requirements of applications such as these have also increased, insuring a continued role for IMS.

**Network Databases**

The simple structure of a hierarchical database became a disadvantage when the data had a more complex structure. In an order-processing database, for example, a single

- 40 -

order might participate in three *different* parent/child relationships, linking the order to the customer who placed it, the salesperson who took it, and the product ordered, as shown in Figure 4-3. The structure of this type of data simply didn't fit the strict hierarchy of IMS.



**Figure 4-3:** Multiple parent/child relationships

To deal with applications such as order processing, a new *network* data model was developed. The network data model extended the hierarchical model by allowing a record to participate in multiple parent/child relationships, as shown in Figure 4-4. These relationships were known as *sets* in the network model. In 1971 the Conference on Data Systems Languages published an official standard for network databases, which became known as the CODASYL model. IBM never developed a network DBMS of its own, choosing instead to extend IMS over the years. But during the 1970s independent software companies rushed to embrace the network model, creating products such as Cullinet's IDMS, Cincom's Total, and the Adabas DBMS that became very popular.



**Figure 4-4:** A network (CODASYL) order-processing database

For a programmer, accessing a network database was very similar to accessing a hierarchical database. An application program could:

• find a specific parent record by key (such as a customer number), • move down to the first child in a particular set (the first order placed by this customer),

• move sideways from one child to the next in the set (the next order placed by the same customer), or

• move up from a child to its parent in another set (the salesperson who took the order).

Once again the programmer had to navigate the database record-by-record, this time specifying which relationship to navigate as well as the direction.

- 41 -

Network databases had several advantages:

• *Flexibility.* Multiple parent/child relationships allowed a network database to represent data that did not have a simple hierarchical structure.

• *Standardization.* The CODASYL standard boosted the popularity of the network

model, and minicomputer vendors such as Digital Equipment Corporation and Data General implemented network databases.

• *Performance.* Despite their greater complexity, network databases boasted performance approaching that of hierarchical databases. Sets were represented by

pointers to physical data records, and on some systems, the database administrator could specify data clustering based on a set relationship.

Network databases had their disadvantages, too. Like hierarchical databases, they were very rigid. The set relationships and the structure of the records had to be specified in advance. Changing the database structure typically required rebuilding the entire database.

Both hierarchical and network databases were tools for programmers. To answer a question such as "What is the most popular product ordered by Acme Manufacturing?" a programmer had to write a program that navigated its way through the database. The backlog of requests for custom reports often stretched to weeks or months, and by the time the program was written, the information it delivered was often worthless.

The disadvantages of the hierarchical and network models led to intense interest in the new *relational* data model when it was first described by Dr. Codd in 1970. At first the relational model was little more than an academic curiosity. Network databases continued to be important throughout the 1970s and early 1980s, particularly on the minicomputer systems that were surging in popularity. However, by the mid-1980s the relational model was clearly emerging as the "new wave" in data management. By the early 1990s, network databases were clearly declining in importance, and today they no longer play a major role in the database market.

**The Relational Data Model**

The relational model proposed by Dr. Codd was an attempt to simplify database structure. It eliminated the explicit parent/child structures from the database, and instead represented all data in the database as simple row/column tables of data values. Figure 4-5 shows a relational version of the network order-processing database in Figure 4-4.



**Figure 4-5:** A relational order-processing database

Unfortunately, the practical definition of "What is a relational database?" became much less clear-cut than the precise, mathematical definition in Codd's 1970 paper. Early relational database management systems failed to implement some key parts of Codd's model, which are only now finding their way into commercial products. As the relational concept grew in popularity, many databases that were called "relational" in fact were not.

- 42 -

In response to the corruption of the term "relational," Dr. Codd wrote an article in 1985 setting forth 12 rules to be followed by any database that called itself "truly relational." Codd's 12 rules have since been accepted as *the* definition of a truly relational DBMS. However, it's easier to start with a more informal definition:

A relational database is a database where all data visible to the user is organized strictly as tables of data values, and where all database operations work on these tables.

The definition is intended specifically to rule out structures such as the embedded pointers of a hierarchical or network database. A relational DBMS can represent parent/child relationships, but they are represented strictly by the data values contained in the database tables.

**The Sample Database**

Figure 4-6 shows a small relational database for an order-processing application. This sample database is used throughout this book and provides the basis for most of the examples. Appendix A contains a complete description of the database structure and its contents.



**Figure 4-6:** The sample database

The sample database contains five tables. Each table stores information about one particular *kind* of entity:

• The CUSTOMERS table stores data about each customer, such as the company name, credit limit, and the salesperson who calls on the customer.

• The SALESREPS table stores the employee number, name, age, year-to-date sales, and other data about each salesperson.

• The OFFICES table stores data about each of the five sales offices, including the city where the office is located, the sales region to which it belongs, and so on.

• The ORDERS table keeps track of every order placed by a customer, identifying the

salesperson who took the order, the product ordered, the quantity and amount of the order, and so on. For simplicity, each order is for only one product.

• The PRODUCTS table stores data about each product available for sale, such as the manufacturer, product number, description, and price.

- 43 -

**Tables**

The organizing principle in a relational database is the *table,* a rectangular, row/column arrangement of data values. Each table in a database has a unique *table name* that identifies its contents. (Actually, each user can choose their own table names without worrying about the names chosen by other users, as explained in Chapter 5.)

The row/column structure of a table is shown more clearly in Figure 4-7, which is an enlarged view of the OFFICES table. Each horizontal *row* of the OFFICES table represents a single physical entity—a single sales office. Together the five rows of the table represent all five of the company's sales offices. All of the data in a particular row of the table applies to the office represented by that row.

**Figure 4-7:.** The row/column structure of a relational table

Each vertical *column* of the OFFICES table represents one item of data that is stored in the database for each office. For example, the CITY column holds the location of each office. The SALES column contains each office's year-to-date sales total. The MGR column shows the employee number of the person who manages the office.

Each row of a table contains exactly one data value in each column. In the row representing the New York office, for example, the CITY column contains the value "New York." The SALES column contains the value "$692,637.00," which is the year-to-date sales total for the New York office.

For each column of a table, all of the data values in that column hold the same type of data. For example, all of the CITY column values are words, all of the SALES values are money amounts, and all of the MGR values are integers (representing employee numbers). The set of data values that a column can contain is called the *domain* of the column. The domain of the CITY column is the set of all names of cities. The domain of the SALES column is any money amount. The domain of the REGION column is just two data values, "Eastern" and "Western," because those are the only two sales regions the company has!

- 44 -

Each column in a table has a *column name,* which is usually written as a heading at the top of the column. The columns of a table must all have different names, but there is no prohibition against two columns in two different tables having identical names. In fact, frequently used column names, such as NAME, ADDRESS, QTY, PRICE, and SALES, are often found in many different tables of a production database.

The columns of a table have a left-to-right order, which is defined when the table is first created. A table always has at least one column. The ANSI/ISO SQL standard does not specify a maximum number of columns in a table, but almost all commercial SQL products do impose a limit. Usually the limit is 255 columns per table or more.

Unlike the columns, the rows in a table do *not* have any particular order. In fact, if you use two consecutive database queries to display the contents of a table, there is no guarantee that the rows will be listed in the same order twice. Of course you can ask SQL to sort the rows before displaying them, but the sorted order has nothing to do with the actual arrangement of the rows within the table.

A table can have any number of rows. A table of zero rows is perfectly legal and is called an *empty* table (for obvious reasons). An empty table still has a structure, imposed by its columns; it simply contains no data. The ANSI/ISO standard does not limit the number of rows in a table, and many SQL products will allow a table to grow until it exhausts the

available disk space on the computer. Other SQL products impose a maximum limit, but it is always a very generous one—two billion rows or more is common.

**Primary Keys**

Because the rows of a relational table are unordered, you cannot select a specific row by its position in the table. There is no "first row," "last row," or "thirteenth row" of a table. How then can you specify a particular row, such as the row for the Denver sales office?

In a well-designed relational database every table has some column or combination of columns whose values uniquely identify each row in the table. This column (or columns) is called the *primary key* of the table. Look once again at the OFFICES table in Figure 4- 7. At first glance, either the OFFICE column or the CITY column could serve as a primary key for the table. But if the company expands and opens two sales offices in the same city, the CITY column could no longer serve as the primary key. In practice, "ID numbers," such as an office number (OFFICE in the OFFICES table), an employee number (EMPL\_NUM in the SALESREPS table), and customer numbers (CUST\_NUM in the CUSTOMERS table), are often chosen as primary keys. In the case of the ORDERS table there is no choice—the only thing that uniquely identifies an order is its order number (ORDER\_NUM).

The PRODUCTS table, part of which is shown in Figure 4-8, is an example of a table where the primary key must be a *combination* of columns. The MFR\_ID column identifies the manufacturer of each product in the table, and the PRODUCT\_ID column specifies the manufacturer's product number. The PRODUCT\_ID column might make a good primary key, but there's nothing to prevent two different manufacturers from using the same number for their products. Therefore, a combination of the MFR\_ID and PRODUCT\_ID columns must be used as the primary key of the PRODUCTS table. Every product in the table is guaranteed to have a unique combination of data values in these two columns.

- 45 -



**Figure 4-8:** A table with a composite primary key

The primary key has a different unique value for each row in a table, so no two rows of a table with a primary key are exact duplicates of one another. A table where every row is different from all other rows is called a *relation* in mathematical terms. The name "relational database" comes from this term, because relations (tables with distinct rows) are at the heart of a relational database.

Although primary keys are an essential part of the relational data model, early relational database management systems (System/R, DB2, Oracle, and others) did not provide explicit support for primary keys. Database designers usually ensured that all of the tables in their databases had a primary key, but the DBMS itself did not provide a way to identify the primary key of a table. DB2 Version 2, introduced in April 1988, was the first of IBM's commercial SQL products to support primary keys. The ANSI/ISO standard was subsequently expanded to include a definition of primary key support.

**Relationships**

One of the major differences between the relational model and earlier data models is that explicit pointers, such as the parent/child relationships of a hierarchical database, are banned from relational databases. Yet obviously these relationships exist in a relational database. For example, in the sample database, each of the salespeople is assigned to a particular sales office, so there is an obvious relationship between the rows of the OFFICES table and the rows of the SALESREPS table. Doesn't the relational model "lose information" by banning these relationships from the database?

As shown in Figure 4-9, the answer to the question is "no." The figure shows a close-up of a few rows of the OFFICES and SALESREPS tables. Note that the REP\_OFFICE column of the SALESREPS table contains the office number of the sales office where each salesperson works. The domain of this column (the set of legal values it may contain) is *precisely* the set of office numbers found in the OFFICE column of the OFFICES table. In fact, you can find the sales office where Mary Jones works by finding the value in Mary's REP\_OFFICE column (11) and finding the row of the OFFICES table that has a matching value in the OFFICE column (in the row for the New York office). Similarly, to find all the salespeople who work in New York, you could note the OFFICE value for the New York row (11) and then scan down the REP\_OFFICE column of the SALESREPS table looking for matching values (in the rows for Mary Jones and Sam Clark).

- 46 -

**Figure 4-9:** A parent/child relationship in a relational database

The parent/child relationship between a sales office and the people who work there isn't lost by the relational model, it's just not represented by an explicit pointer stored in the database. Instead, the relationship is represented by *common data values* stored in the two tables. All relationships in a relational database are represented this way. One of the main goals of the SQL language is to let you retrieve related data from the database by manipulating these relationships in a simple, straightforward way.

**Foreign Keys**

A column in one table whose value matches the primary key in some other table is called a *foreign key.* In Figure 4-9 the REP\_OFFICE column is a foreign key for the OFFICES table. Although REP\_OFFICE is a column in the SALESREPS table, the values that this column contains are office numbers. They match values in the OFFICE column, which is the primary key for the OFFICES table. Together, a primary key and a foreign key create a parent/child relationship between the tables that contain them, just like the parent/child relationships in a hierarchical database.

Just as a combination of columns can serve as the primary key of a table, a foreign key can also be a combination of columns. In fact, the foreign key will *always* be a compound (multi-column) key when it references a table with a compound primary key. Obviously, the number of columns and the data types of the columns in the foreign key and the primary key must be identical to one another.

A table can contain more than one foreign key if it is related to more than one other table. Figure 4-10 shows the three foreign keys in the ORDERS table of the sample database:

- 47 -

**Figure 4-10:** Multiple parent/child relationships in a relational database

• The CUST column is a foreign key for the CUSTOMERS table, relating each order to the customer who placed it.

• The REP column is a foreign key for the SALESREPS table, relating each order to the salesperson who took it.

• The MFR and PRODUCT columns together are a composite foreign key for the PRODUCTS table, relating each order to the product being ordered.

The multiple parent/child relationships created by the three foreign keys in the ORDERS table may seem familiar to you, and they should. They are precisely the same relationships as those in the network database of Figure 4-4. As the example shows, the relational data model has all of the power of the network model to express complex relationships.

Foreign keys are a fundamental part of the relational model because they create relationships among tables in the database. Unfortunately, as with primary keys, foreign key support was missing from early relational database management systems. They were added to DB2 Version 2, have since been added to the ANSI/ISO standard, and now appear in many commercial products.

**Codd's Twelve Rules \***

In his 1985 *Computerworld* article, Ted Codd presented 12 rules that a database must obey if it is to be considered truly relational. Codd's 12 rules, shown in the following list, have since become a semi-official definition of a relational database. The rules come out of Codd's theoretical work on the relational model and actually represent more of an ideal goal than a definition of a relational database.

1. *The information rule.* All information in a relational database is represented explicitly at the logical level and in exactly one way—by values in tables.

2. *Guaranteed access rule.* Each and every datum (atomic value) in a relational

database is guaranteed to be logically accessible by resorting to a combination of table name, primary key value, and column name.

3. *Systematic treatment of null values*. Null values (distinct from an empty character string or a string of blank characters and distinct from zero or any other number) are

- 48 -

supported in a fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of the data type.

4. *Dynamic online catalog based on the relational model*. The database description is represented at the logical level in the same way as ordinary data, so that authorized

users can apply the same relational language to its interrogation as they apply to the regular data.

5. *Comprehensive data sublanguage rule*. A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks

mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings, and that is comprehensive in supporting all of the following items:

• Data definition

• View definition

• Data manipulation (interactive and by program)

• Integrity constraints

• Authorization

• Transaction boundaries (begin, commit, and rollback)

6. *View updating rule*. All views that are theoretically updateable are also updateable by the system.

7. *High-level insert*, *update*, *and delete*. The capability of handling a base relation or a

derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update, and deletion of data.

8. *Physical data independence*. Application programs and terminal activities remain

logically unimpaired whenever any changes are made in either storage representations or access methods.

9. *Logical data independence*. Application programs and terminal activities remain

logically unimpaired when information preserving changes of any kind that theoretically permit unimpairment are made to the base tables.

10. *Integrity independence*. Integrity constraints specific to a particular relational

database must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.

11. *Distribution independence.* A relational DBMS has distribution independence. 12. *Nonsubversion rule.* If a relational system has a low-level (single record at a time)

language, that low level cannot be used to subvert or bypass the integrity rules and

constraints expressed in the higher-level relational language (multiple records at a time).

During the early 1990s, it became popular practice to compile "scorecards" for commercial DBMS products, showing how well they satisfy each of the rules. Unfortunately, the rules are subjective so the scorecards were usually full of footnotes and qualifications, and didn't reveal a great deal about the products. Today, the basis of competition for database vendors tends to revolve around performance, new features, the availability of development tools, the quality of vendor support, and other issues,

- 49 -

rather than conformance to Codd's rules. Nonetheless, they are an important part of the history of the relational model.

Rule 1 is basically the informal definition of a relational database presented at the beginning of this section.

Rule 2 stresses the importance of primary keys for locating data in the database. The table name locates the correct table, the column name finds the correct column, and the primary key value finds the row containing an individual data item of interest. Rule 3 requires support for missing data through NULL values, which are described in Chapter 5.

Rule 4 requires that a relational database be self-describing. In other words, the database must contain certain *system tables* whose columns describe the structure of the database itself. These tables are described in Chapter 16.

Rule 5 mandates using a relational database language, such as SQL, although SQL is not specifically required. The language must be able to support all the central functions of a DBMS—creating a database, retrieving and entering data, implementing database security, and so on.

Rule 6 deals with views, which are *virtual tables* used to give various users of a database different views of its structure. It is one of the most challenging rules to implement in practice, and no commercial product fully satisfies it today. Views and the problems of updating them are described in Chapter 14.

Rule 7 stresses the set-oriented nature of a relational database. It requires that rows be treated as sets in insert, delete, and update operations. The rule is designed to prohibit implementations that only support row-at-a-time, navigational modification of the database.

Rule 8 and Rule 9 insulate the user or application program from the low-level implementation of the database. They specify that specific access or storage techniques used by the DBMS, and even changes to the structure of the tables in the database, should not affect the user's ability to work with the data.

Rule 10 says that the database language should support integrity constraints that restrict the data that can be entered into the database and the database modifications that can be made. This is another of the rules that is not supported in most commercial DBMS products.

Rule 11 says that the database language must be able to manipulate distributed data located on other computer systems. Distributed data and the challenges of managing it are described in Chapter 20.

Finally, Rule 12 prevents "other paths" into the database that might subvert its relational structure and integrity.

**Summary**

SQL is based on the relational data model that organizes the data in a database as a collection of tables:

• Each table has a table name that uniquely identifies it.

• Each table has one or more named columns, which are arranged in a specific, left-to right order.

• Each table has zero or more rows, each containing a single data value in each column. The rows are unordered.

- 50 -

• All data values in a given column have the same data type, and are drawn from a set  of legal values called the domain of the column.

Tables are related to one another by the data they contain. The relational data model uses primary keys and foreign keys to represent these relationships among tables:

• A primary key is a column or combination of columns in a table whose value(s) uniquely identify each row of the table. A table has only one primary key.

• A foreign key is a column or combination of columns in a table whose value(s) are a

primary key value for some other table. A table can contain more than one foreign key, linking it to one or more other tables.

• A primary key/foreign key combination creates a parent/child relationship between the tables that contain them.

**Part II: Retrieving Data**

**Chapter List**

**Chapter  5:**

**Chapter 6:**

**Chapter 7:**

**Chapter 8:**

**Chapter 9:**

**SQL Basics**

**Simple Queries**

**Multi-Table Queries (Joins) Summary Queries**

**Subqueries and Query Expressions**

**Chapter 5: SQL Basics**

**Overview**

This chapter begins a detailed description of the features of SQL. It describes the basic structure of a SQL statement and the basic elements of the language, such as keywords, data types, and expressions. The way that SQL handles missing data through NULL values is also described. Although these are basic features of SQL, there are some subtle differences in the way they are implemented by various popular SQL products, and in many cases the SQL products provide significant extensions to the capabilities specified in the ANSI/ISO SQL standard. These differences and extensions are also described in this chapter.

**Statements**

The main body of the SQL language consists of about 40 statements, which are summarized in Table 5-1. Each statement requests a specific action from the DBMS, such as creating a new table, retrieving data, or inserting new data into the database. All SQL statements have the same basic form, illustrated in Figure 5-1.

- 51 -

**Figure 5-1:** The structure of a SQL statement

**Table 5-1: Major SQL Statements**

**Statement Description**

***Data Manipulation***

SELECT Retrieves data from the database INSERT Adds new rows of data to the database DELETE Removes rows of data from the database UPDATE Modifies existing database data ***Data Definition***

CREATE TABLE Adds a new table to the database DROP TABLE Removes a table from the database ALTER TABLE Changes the structure of an existing table CREATE VIEW Adds a new view to the database DROP VIEW Removes a view from the database CREATE INDEX Builds an index for a column DROP INDEX Removes the index for a column CREATE SCHEMA Adds a new schema to the database DROP SCHEMA Removes a schema from the database CREATE DOMAIN Adds a new data value domain

- 52 -

ALTER DOMAIN Changes a domain definition DROP DOMAIN Removes a domain from the database ***Access Control***

GRANT Grants user access privileges REVOKE Removes user access privileges ***Control***

***Transaction***

COMMIT Ends the current transaction ROLLBACK Aborts the current transaction

SET

TRANSACTION

Defines data access characteristics of the current transaction

***SQL***

***Programmatic***

DECLARE Defines a cursor for a query

EXPLAIN Describes the data access plan for a query

OPEN Opens a cursor to retrieve query results

FETCH Retrieves a row of query results

CLOSE Closes a cursor

PREPARE Prepares a SQL statement for dynamic execution EXECUTE Executes a SQL statement dynamically

DESCRIBE Describes a prepared query

Every SQL statement begins with a *verb*, a keyword that describes what the statement does. CREATE, INSERT, DELETE, and COMMIT are typical verbs. The statement continues with one or more *clauses*. A clause may specify the data to be acted upon by the statement or provide more detail about what the statement is supposed to do. Every clause also begins with a keyword, such as WHERE, FROM, INTO, and HAVING. Some clauses are optional; others are required. The specific structure and content vary from one clause to another. Many clauses contain table or column names; some may contain additional keywords, constants, or expressions.

The ANSI/ISO SQL standard specifies the SQL keywords that are used as verbs and in statement clauses. According to the standard, these keywords cannot be used to name database objects, such as tables, columns, and users. Many SQL implementations relax this restriction, but it's generally a good idea to avoid the keywords when you name your

- 53 -

tables and columns. Table 5-2 lists the keywords included in the ANSI/ISO SQL2 standard, which roughly tripled the number of keywords reserved by the earlier SQL1 standard. The SQL2 standard also includes a list of "potential keywords" that are candidates for becoming keywords in future revisions of the standard. These keywords are listed in Table 5-3.

**Table 5-2: ANSI/ISO SQL2 Keywords**

ABSOLUTE CROSS GET NEXT SPACE ACTION CURRENT GLOBAL NO SQL ADD CURRENT\_DATE GO NOT SQLCODE ALL CURRENT\_TIME GOTO NULL SQLERROR ALLOCATE CURRENT\_TIMESTAMP GRANT OCTET\_LENGTH SQLSTATE ALTER CURRENT\_USER GROUP OF SUBSTRING AND CURSOR HAVING ON SUM ANY DATE HOUR ONLY SYSTEM\_USER ARE DAY IDENTITY OPEN TABLE AS DEALLOCATE IMMEDIATE OPTION TEMPORARY ASC DEC IN OR THEN ASSERTION DECIMAL INDICATOR ORDER TIME AT DECLARE INITIALLY OUTER TIMESTAMP AUTHORIZATION DEFAULT INNER OUTPUT TIMEZONE\_HOUR AVG DEFERRABLE INPUT OVERLAPS TIMEZONE\_MINUTE BEGIN DEFERRED INSENSITIVE PAD TO BETWEEN DELETE INSERT PARTIAL TRAILING BIT DESC INT POSITION TRANSACTION BIT\_LENGTH DESCRIBE INTEGER PRECISION TRANSLATE BOTH DESCRIPTOR INTERSECT PREPARE TRANSLATION BY DIAGNOSTICS INTERVAL PRESERVE TRIM CASCADE DISCONNECT INTO PRIMARY TRUE

- 54 -

CASCADED DISTINCT IS PRIOR UNION CASE DOMAIN ISOLATION PRIVILEGES UNIQUE CAST DOUBLE JOIN PROCEDURE UNKNOWN CATALOG DROP KEY PUBLIC UPDATE CHAR ELSE LANGUAGE READ UPPER CHARACTER END LAST REAL USAGE CHAR\_LENGTH END-EXEC LEADING REFERENCES USER

CHARACTER\_LENGTHESCAPE LEFT RELATIVE USING CHECK EXCEPT LEVEL RESTRICT VALUE CLOSE EXCEPTION LIKE REVOKE VALUES COALESCE EXEC LOCAL RIGHT VARCHAR COLLATE EXECUTE LOWER ROLLBACK VARYING COLLATION EXISTS MATCH ROWS VIEW COLUMN EXTERNAL MAX SCHEMA WHEN COMMIT EXTRACT MIN SCROLL WHENEVER CONNECT FALSE MINUTE SECOND WHERE CONNECTION FETCH MODULE SECTION WITH CONSTRAINT FIRST MONTH SELECT WORK CONSTRAINTS FLOAT NAMES SESSION WRITE CONTINUE FOR NATIONAL SESSION\_USER YEAR CONVERT FOREIGN NATURAL SET ZONE CORRESPONDING FOUND NCHAR SIZE COUNT FROM NULLIF SMALLINT CREATE FULL NUMERIC SOME

**Table 5-3: ANSI/ISO SQL2 Potential Keywords**

- 55 -

AFTER EQUALS OLD RETURN TEST ALIAS GENERAL OPERATION RETURNS THERE ASYNC IF OPERATORS ROLE TRIGGER BEFORE IGNORE OTHERS ROUTINE TYPE BOOLEAN LEAVE PARAMETERS ROW UNDER BREADTH LESS PENDANT SAVEPOINT VARIABLE COMPLETION LIMIT PREORDER SEARCH VIRTUAL CALL LOOP PRIVATE SENSITIVE VISIBLE CYCLE MODIFY PROTECTED SEQUENCE WAIT DATA NEW RECURSIVE SIGNAL WHILE DEPTH NONE REF SIMILAR WITHOUT DICTIONARY OBJECT REFERENCING SQLEXCEPTION EACH OFF REPLACE SQLWARNING ELSEIF OID RESIGNAL STRUCTURE

Throughout this book, the acceptable forms of a SQL statement are illustrated by a syntax diagram, such as the one shown in Figure 5-2. A valid SQL statement or clause is constructed by "following the line" through the syntax diagram to the dot that marks the end of the diagram. Keywords in the syntax diagram and in the examples (such as DELETE and FROM in Figure 5-2) are always shown in UPPERCASE, but almost all SQL implementations accept both uppercase and lowercase keywords, and it's often more convenient to actually type them in lowercase.



**Figure 5-2:** A sample syntax diagram

Variable items in a SQL statement (such as the table name and search condition in Figure 5-2) are shown in *lowercase italics*. It's up to you to specify the appropriate item each time the statement is used. Optional clauses and keywords, such as the WHERE clause in Figure 5-2, are indicated by alternate paths through the syntax diagram. When a choice of optional keywords is offered, the default choice (that is, the behavior of the statement if no

- 56 -

keyword is specified) is UNDERLINED.

**Names**

The objects in a SQL-based database are identified by assigning them unique names. Names are used in SQL statements to identify the database object on which the statement should act. The most fundamental named objects in a relational database are table names (which identify tables), column names (which identify columns), and user names (which identify users of the database); conventions for naming these objects were specified in the original SQL1 standard. The ANSI/ISO SQL2 standard significantly expanded the list of named entities, to include schemas (collections of tables), constraints (restrictions on the contents of tables and their relationships), domains (sets of legal values that may be assigned to a column), and several other types of objects. Many SQL implementations support additional named objects, such as stored procedures (Sybase and SQL Server), primary key/foreign key relationships (DB2), and data entry forms (Ingres).

The original ANSI/ISO standard specified that SQL names must contain 1 to 18 characters, must begin with a letter, and may not contain any spaces or special punctuation characters. The SQL2 standard increased the maximum to 128 characters. In practice the names supported by SQL-based DBMS products vary significantly. DB2, for example, restricts user names to 8 characters but allows longer table and column names. The various products also differ in the special characters they permit in table names. For portability it's best to keep names relatively short and to avoid the use of special characters.

**Table Names**

When you specify a table name in a SQL statement, SQL assumes that you are referring to one of your own tables (that is, a table that you created). Usually, you will want to choose table names that are short but descriptive. The table names in the sample database (ORDERS, CUSTOMERS, OFFICES, SALESREPS) are good examples. In a personal or departmental database, the choice of table names is usually up to the database developer or designer.

In a larger, shared-use corporate database, there may be corporate standards for naming tables, to insure that table names do not conflict. In addition, most DBMS brands allow different users to create tables with the same name (that is, both Joe and Sam can create a table named BIRTHDAYS). The DBMS uses the appropriate table, depending on which user is requesting data. With the proper permission, you can also refer to tables owned by other users, by using a *qualified table name.* A qualified table name specifies both the name of the table's owner and the name of the table, separated by a period (.). For example, Joe could access the BIRTHDAYS table owned by Sam by using the qualified table name:

SAM.BIRTHDAYS

A qualified table name generally can be used in a SQL statement wherever a table name can appear.

The ANSI/ISO SQL2 standard generalizes the notion of a qualified table name even further. It allows you to create a named collection of tables, called a *schema*. You can refer to a table in a specific schema using a qualified table name. For example, the BIRTHDAYS table in the EMPLOYEEINFO schema would be referenced as:

EMPLOYEEINFO.BIRTHDAYS

Chapter 13 provides more information about schemas, users, and other aspects of SQL database structure.

- 57 -

**Column Names**

When you specify a column name in a SQL statement, SQL can normally determine from the context which column you intend. However, if the statement involves two columns with the same name from two different tables, you must use a *qualified column name* to unambiguously identify the column you intend. A qualified column name specifies both the name of the table containing the column and the name of the column, separated by a period (.). For example, the column named SALES in the SALESREPS table has the qualified column name:

SALESREPS.SALES

If the column comes from a table owned by another user, a qualified table name is used in the qualified column name. For example, the BIRTHDATE column in the BIRTHDAYS table owned by the user SAM is specified by the fully qualified column name:

SAM.BIRTHDAYS.BIRTH\_DATE

Qualified column names can generally be used in a SQL statement wherever a simple (unqualified) column name can appear; exceptions are noted in the descriptions of the individual SQL statements.

**Data Types**

The ANSI/ISO SQL standard specifies the various types of data that can be stored in a SQL-based database and manipulated by the SQL language. The original SQL1 standard specified only a minimal set of data types. The SQL2 standard expanded this list to include variable-length character strings, date and time data, bit strings, and other types. Today's commercial DBMS products can process a rich variety of different kinds of data, and there is considerable diversity in the particular data types supported across different DBMS brands. Typical data types include:

• *Integers*. Columns holding this type of data typically store counts, quantities, ages, and

so on. Integer columns are also frequently used to contain I.D. numbers, such as customer, employee, and order numbers.

• *Decimal numbers*. Columns with this data type store numbers that have fractional

parts and must be calculated exactly, such as rates and percentages. They are also frequently used to store money amounts.

• *Floating point numbers*. Columns with this data type are used to store scientific numbers that can be calculated approximately, such as weights and distances.

Floating point numbers can represent a larger range of values than decimal numbers but can produce round-off errors in computations.

• *Fixed-length character strings*. Columns holding this type of data typically store names of people and companies, addresses, descriptions, and so on.

• *Variable-length character strings*. This data type allows a column to store character

strings that vary in length from row to row, up to some maximum length. (The SQL1

standard permitted only fixed-length character strings, which are easier for the DBMS to process but can waste considerable space.)

• *Money amounts*. Many SQL products support a MONEY or CURRENCY type, which is

usually stored as a decimal or floating point number. Having a distinct money type allows the DBMS to properly format money amounts when they are displayed.

- 58 -

• *Dates and times*. Support for date/time values is also common in SQL products, although the details vary dramatically from one product to another. Various combinations of dates, times, timestamps, time intervals, and date/time arithmetic are

generally supported. The SQL2 standard includes an elaborate specification for DATE, TIME, TIMESTAMP, and INTERVAL data types, including support for time zones and time precision (for example, tenths or hundredths of seconds).

• *Boolean data*. Some SQL products, such as Informix Dynamic Server, support logical

(TRUE or FALSE) values as an explicit type, and some permit logical operations (comparison, AND/OR, and so on) on the stored data within SQL statements.

• *Long text*. Several SQL-based databases support columns that store long text strings (typically up to 32,000 or 65,000 characters, and in some cases even larger). This

allows the database to store entire documents, product descriptions, technical papers, resumes, and similar unstructured text data. The DBMS usually restricts the use of these columns in interactive queries and searches.

• *Unstructured byte streams*. Several DBMS products allow unstructured, variable length sequences of bytes to be stored and retrieved. Columns containing this data

are used to store compressed video images, executable code, and other types of unstructured data. SQL Server's IMAGE data type, for example, can store a stream of up to 2 billion bytes of data.

• *Asian characters*. As databases grow to support global applications, DBMS vendors have added support for fixed-length and variable-length strings of 16-bit characters

used to represent Kanji and other Asian characters. Searching and sorting on these GRAPHIC and VARGRAPHIC types is usually not permitted, however.

Table 5-4 lists the data types specified in the ANSI/ISO SQL standard. **Table 5-4: ANSI/ISO SQL Data Types**

**Data Type Description**

CHAR(*len*)

CHARACTER(*len*)

VARCHAR(*len*)

CHAR VARYING(*len*)

CHARACTER VARYING(*len*)

NCHAR(*len*)

NATIONAL CHAR(*len*)

NATIONAL CHARACTER(*len*)

NCHAR VARYING(*len*)

NATIONAL CHAR VARYING(*len*) NATIONAL CHARACTER

VARYING(*len*)

INTEGER

INT

Fixed-length character strings

Variable-length character strings\* Fixed-length national character strings\* Variable-length national character strings\*

Integer numbers

- 59 -

SMALLINT Small integer numbers BIT(*len*) Fixed-length bit string\*

BIT VARYING(*len*) Variable-length bit string\* NUMERIC(*precision,scale*)

DECIMAL(*precision,scale*) DEC(*precision,scale*)

Decimal numbers

FLOAT(*precision*) Floating point numbers

REAL Low-precision floating point numbers DOUBLE PRECISION High-precision floating point numbers DATE Calendar date\*

TIME(*precision*) Clock time\*

TIMESTAMP(*precision*) Date and time\*

INTERVAL Time interval\*

\*new data type in SQL2

The differences between the data types offered in various SQL implementations is one of the practical barriers to the portability of SQL-based applications. These differences have come about as a result of innovation as relational databases have evolved to include a broader range of capabilities. The typical pattern has been:

• A DBMS vendor adds a new data type that provides useful new capability for a certain group of users.

• Other DBMS vendors add the same or similar data types, adding their own innovations to differentiate their products from the others.

• Over several years, the popularity of the data type grows, and it becomes a part of the "mainstream" set of data types supported by most SQL implementations.

• The standards bodies become involved to try to standardize the new data type and eliminate arbitrary differences between the vendor implementations. The more well

entrenched the data type has become, the more difficult the set of compromises faced by the standards group. Usually this results in an addition to the standard that does not exactly match *any* of the current implementations.

• DBMS vendors slowly add support for the new standardized data type as an option to their systems, but because they have a large installed base that is using the older

(now "proprietary") version of the data type, they must maintain support for this form of the data type as well.

• Over a very long period of time (typically several major releases of the DBMS product),

- 60 -

users migrate to the new, standardized form of the data type, and the DBMS vendor can begin the process of obsoleting the proprietary version.

Date/time data provides an excellent example of this phenomenon and the data type variations it creates. DB2 has long offered support for three different date/time data types:

• DATE, which stores a date like June 30, 1991,

• TIME, which stores a time of day like 12:30 P.M., and

• TIMESTAMP, which is a specific instant in history, with a precision down to the nanosecond.

Specific dates and times can be specified as string constants, and date arithmetic is supported. Here is an example of a valid query using DB2 dates, assuming that the HIREDATE column contains DATE data:

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE HIRE\_DATE >= '05/30/1989' + 15 DAYS

SQL Server provides a single date/time data type, called DATETIME, which closely resembles the DB2 TIMESTAMP data type. If HIRE\_DATE contained DATETIME data, SQL Server could accept this version of the query (without the date arithmetic):

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE HIRE\_DATE >= '06/14/1989'

Since no specific time on June 14, 1989, is specified in the query, SQL Server defaults to midnight on that date. The SQL Server query thus *really* means:

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE HIRE\_DATE >= '06/14/1989 12:00AM'

If a salesperson's hire date was stored in the database as midday on June 14, 1989, the salesperson would not be included in the SQL Server query results but would have been included in the DB2 results (because only the date would be stored). SQL Server also supports date arithmetic through a set of built-in functions. Thus the DB2-style query can also be specified in this way:

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE HIRE\_DATE >= DATEADD(DAY, 15, '05/30/1989')

which, of course, is considerably different from the DB2 syntax.

Oracle also supports date/time data, with a single data type called DATE. Like SQL Server's DATETIME type, an Oracle DATE is, in fact, a timestamp. Also like SQL Server, the time part of an Oracle DATE value defaults to midnight if no time is explicitly specified. The default Oracle date format is different from the DB2 and SQL Server formats, so the Oracle version of the query becomes:

SELECT NAME, HIRE\_DATE

- 61 -

FROM SALESREPS

WHERE HIRE\_DATE >= '14-JUN-89'

Oracle also supports limited date arithmetic, so the DB2-style query can also be specified but without the DAYS keyword:

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE HIRE\_DATE >= '30-MAY-89' + 15

Finally, the ANSI/ISO SQL2 standard added support for date/time data with a set of data types that are based on, but not identical to, the DB2 types. In addition to the DATE, TIME, and TIMESTAMP data types, the standard specifies an INTERVAL data type, which can be used to store a time interval (for example, a timespan measured in days, or a duration measured in hours, minutes, and seconds). The standard also provides a very elaborate and complex method for dealing with date/time arithmetic, specifying the precision of intervals, adjusting for time zone differences, and so on.

As these examples illustrate, the subtle differences in data types among various SQL products lead to some significant differences in SQL statement syntax. They can even cause the same SQL query to produce slightly different results on different database management systems. The widely praised portability of SQL is thus true but only at a general level. An application can be moved from one SQL database to another, and it can be highly portable if it uses only the most mainstream, basic SQL capabilities. However, the subtle variations in SQL implementations mean that data types and SQL statements must almost always be adjusted somewhat if it is to be moved across DBMS brands. The more complex the application, the more likely it is to become dependent on DBMS-specific features and nuances, and the less portable it will become.

**Constants**

In some SQL statements a numeric, character, or date data value must be expressed in text form. For example, in this INSERT statement, which adds a salesperson to the database:

INSERT INTO SALESREPS (EMPL\_NUM, NAME, QUOTA, HIRE\_DATE, SALES) VALUES (115, 'Dennis Irving', 175000.00, '21-JUN-90', 0.00)

the value for each column in the newly inserted row is specified in the VALUES clause. Constant data values are also used in expressions, such as in this SELECT statement:

SELECT CITY

FROM OFFICES

WHERE TARGET > (1.1 \* SALES) + 10000.00

The ANSI/ISO SQL standard specifies the format of numeric and string constants, or *literals,* which represent specific data values. These conventions are followed by most SQL implementations.

**Numeric Constants**

Integer and decimal constants (also called *exact numeric literals*) are written as ordinary decimal numbers in SQL statements, with an optional leading plus or minus sign.

21 -375 2000.00 +497500.8778

- 62 -

You must not put a comma between the digits of a numeric constant, and not all SQL dialects allow the leading plus sign, so it's best to avoid it. For money data, most SQL implementations simply use integer or decimal constants, although some allow the constant to be specified with a currency symbol:

$0.75 $5000.00 $-567.89

Floating point constants (also called *approximate numeric literals*) are specified using the *E* notation commonly found in programming languages such as C and FORTRAN. Here are some valid SQL floating point constants:

1.5E3 -3.14159E1 2.5E-7 0.783926E21

The E is read "times ten to the power of," so the first constant becomes "1.5 times ten to the third power," or 1500.

**String Constants**

The ANSI/ISO standard specifies that SQL constants for character data be enclosed in single quotes ('. . .'), as in these examples:

'Jones, John J.' 'New York' 'Western'

If a single quote is to be included in the constant text, it is written within the constant as two consecutive single quote characters. Thus this constant value:

'I can''t'

becomes the seven-character string "I can't".

Some SQL implementations, such as SQL Server and Informix, accept string constants enclosed in double quotes (". . ."):

"Jones, John J." "New York" "Western"

Unfortunately, the double quotes pose portability problems with other SQL products, including some unique portability problems with SQL/DS. SQL/DS allows column names containing blanks and other special characters (in violation of the ANSI/ISO standard). When these characters appear as names in a SQL statement, they must be enclosed in double quotes. For example, if the NAME column of the SALESREPS table were called "FULL NAME" in a SQL/DS database, this SELECT statement would be valid:

SELECT "FULL NAME", SALES, QUOTA

FROM SALESREPS

WHERE "FULL NAME" = 'Jones, John J.'

The SQL2 standard provides the additional capability to specify string constants from a specific national character set (for example, French or German) or from a user-defined character set. These capabilities have not yet found their way into mainstream SQL implementations.

**Date and Time Constants**

In SQL products that support date/time data, constant values for dates, times, and time intervals are specified as string constants. The format of these constants varies from one DBMS to the next. Even more variation is introduced by the differences in the way dates and times are written in different countries.

- 63 -

DB2 supports several different international formats for date, time, and timestamp constants, as shown in Table 5-5. The choice of format is made when the DBMS is installed. DB2 also supports durations specified as "special" constants, as in this example:

HIRE\_DATE + 30 DAYS

Note that a duration can't be stored in the database, however, because DB2 doesn't have an explicit DURATION data type.

SQL Server also supports date/time data and accepts a variety of different formats for date and time constants. The DBMS automatically accepts all of the alternate formats, and you can intermix them if you like. Here are some examples of legal SQL Server date constants:

March 15, 1990 Mar 15 1990 3/15/1990 3-15-90 1990 MAR 15 and here are some legal time constants:

15:30:25 3:30:25 PM 3:30:25 pm 3 PM

Oracle dates and times are also written as string constants, using this format: 15-MAR-90

You can also use Oracle's built-in TO\_DATE() function to convert date constants written in other formats, as in this example:

SELECT NAME, AGE

FROM SALESREPS

WHERE HIRE\_DATE = TO\_DATE('JUN 14 1989', 'MON DD YYYY')

The SQL2 standard specifies a format for date and time constants, based on the ISO format in Table 5-5, except that time constants are written with colons instead of periods separating the hours, minutes, and seconds.

**Table 5-5: IBM SQL Date and Time Formats**

**Format Name**

**DATE Format DATE Example**

**TIME Format**

**TIME**

**Example**

American mm/dd/yyyy 5/19/1960 hh:mm am/pm

2:18 PM

European dd.mm.yyyy 19.5.1960 hh.mm.ss 14.18.08 Japanese yyyy-mm-dd 1960-5-19 hh:mm:ss 14:18:08

ISO yyyy-mm-dd 1960-5-19 hh.mm.ss 14.18.08

- 64 -

TIMESTAMP format

TIMESTAMP example

hh.mm.ss.nnnnnn yyyy-mm-dd

14.18.08.048632 1960-05-19-

**Symbolic Constants**

In addition to user-supplied constants, the SQL language includes special symbolic constants that return data values maintained by the DBMS itself. For example, in some DBMS brands the symbolic constant CURRENT\_DATE yields the value of the current date and can be used in queries such as the following, which lists the salespeople whose hire date is still in the future.

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE HIRE\_DATE > CURRENT\_DATE

The SQL1 standard specified only a single symbolic constant (the USER constant described in Chapter 15), but most SQL products provide many more. Generally, a symbolic constant can appear in a SQL statement anywhere that an ordinary constant of the same data type could appear. The SQL2 standard adopted the most useful symbolic constants from current SQL implementations and provides for CURRENT\_DATE, CURRENT\_TIME, and CURRENT\_TIMESTAMP (note the underscores!) as well as USER, SESSION\_USER, and SYSTEM\_USER.

Some SQL products, including SQL Server, provide access to system values through built-in functions rather than symbolic constants. The SQL Server version of the preceding query is:

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE HIRE\_DATE > GETDATE()

Built-in functions are described later in this chapter.

**Expressions**

Expressions are used in the SQL language to calculate values that are retrieved from a database and to calculate values used in searching the database. For example, this query calculates the sales of each office as a percentage of its target:

SELECT CITY, TARGET, SALES, (SALES/TARGET) \* 100

FROM OFFICES

and this query lists the offices whose sales are more than $50,000 over target:

SELECT CITY

FROM OFFICES

WHERE SALES > TARGET + 50000.00

The ANSI/ISO SQL standard specifies four arithmetic operations that can be used in expressions: addition (X + Y), subtraction (X – Y), multiplication (X \* Y), and division (X / Y). Parentheses can also be used to form more complicated expressions, like this one:

- 65 -

(SALES \* 1.05) - (TARGET \* .95)

Strictly speaking, the parentheses are not required in this query because the ANSI/ISO standard specifies that multiplication and division have a higher precedence than addition and subtraction. However, you should always use parentheses to make your expressions unambiguous because different SQL dialects may use different rules. The parentheses also increase the readability of the statement and make programmatic SQL statements easier to maintain.

The ANSI/ISO standard also specifies automatic data type conversion from integers to decimal numbers, and from decimal numbers to floating point numbers, as required. You can thus mix these data types in a numeric expression. Many SQL implementations support other operators and allow operations on character and date data. DB2, for example, supports a string concatenation operator, written as two consecutive vertical bar characters (||). If two columns named FIRST\_NAME and LAST\_NAME contain the values "Jim" and "Jackson," then this DB2 expression:

('Mr./Mrs. ' || FIRST\_NAME || ' ' || LAST\_NAME)

produces the string "Mr./Mrs. Jim Jackson." As already mentioned, DB2 also supports addition and subtraction of DATE, TIME, and TIMESTAMP data, for occasions when those operations make sense. This capability has been included in the SQL2 standard.

**Built-in Functions**

Although the SQL1 standard doesn't specify them, most SQL implementations include a number of useful *built-in functions.* These facilities often provide data type conversion facilities. For example, DB2's built-in MONTH() and YEAR() functions take a DATE or TIMESTAMP value as their input and return an integer that is the month or year portion of the value. This query lists the name and month of hire for each salesperson in the sample database:

SELECT NAME, MONTH(HIRE\_DATE)

FROM SALESREPS

and this one lists all salespeople hired in 1988:

SELECT NAME, MONTH(HIRE\_DATE)

FROM SALESREPS

WHERE YEAR(HIRE\_DATE) = 1988

Built-in functions also are often used for data reformatting. Oracle's built-in TO\_CHAR() function, for example, takes a DATE data type and a format specification as its arguments and returns a string containing a formatted version of the date. In the results produced by this query:

SELECT NAME, TO\_CHAR(HIRE\_DATE,'DAY MONTH DD, YYYY') FROM SALESREPS

the hire dates will all have the format "Wednesday June 14, 1989" because of the built-in function.

In general, a built-in function can be specified in a SQL expression anywhere that a constant of the same data type can be specified. The built-in functions supported by popular SQL dialects are too numerous to list here. The IBM SQL dialects include about two dozen built-in functions, Oracle supports a different set of about two dozen built-in

- 66 -

functions, and SQL Server has several dozen. The SQL2 standard incorporated the most useful built-in functions from these implementations, in many cases with slightly different syntax. These functions are summarized in Table 5-6.

**Table 5-6: Built-in SQL2 Functions**

**Function Returns**

BIT\_LENGTH(*string*) Number of bits in a bit string

CAST(*value* AS *data\_type*) The value, converted to the specified data type (e.g., a date converted to a character string)

CHAR\_LENGTH(*string*) Length of a character string

CONVERT(*string* USING *conv*) String converted as specified by a named conversion function

CURRENT\_DATE Current date

CURRENT\_TIME(*precision*) Current time, with the specified *precision* CURRENT\_TIMESTAMP(*precision*)Current date and time, with the specified *precision*

EXTRACT(*part* FROM *source*) Specified part (DAY, HOUR, etc.) from a DATETIME value

LOWER(*string*) String converted to all lowercase letters OCTET\_LENGTH(*string*) Number of 8-bit bytes in a character string

POSITION(*target* IN *source*) Position where the *target* string appears within the *source* string

SUBSTRING(*source* FROM *n*FOR *len)*

TRANSLATE(*string* USING *trans*)

A portion of the *source* string, beginning at the *n*-th character, for a length of *len*

String translated as specified by a named translation function

TRIM(BOTH *char* FROM *string*) String with both leading and trailing occurrences of *char* trimmed off

TRIM(LEADING *char* FROM *string*)

TRIM(TRAILING *char* FROM *string*)

String with any leading occurrences of *char* trimmed off

String with any trailing occurrences of *char* trimmed off

UPPER(*string*) String converted to all uppercase letters

- 67 -

**Missing Data (NULL Values)**

Because a database is usually a model of a real-world situation, certain pieces of data are inevitably missing, unknown, or don't apply. In the sample database, for example, the QUOTA column in the SALESREPS table contains the sales goal for each salesperson. However, the newest salesperson has not yet been assigned a quota; this data is missing for that row of the table. You might be tempted to put a zero in the column for this salesperson, but that would not be an accurate reflection of the situation. The salesperson does not have a zero quota; the quota is just "not yet known."

Similarly, the MANAGER column in the SALESREPS table contains the employee number of each salesperson's manager. But Sam Clark, the Vice President of Sales, has no manager in the sales organization. This column does not apply to Sam. Again, you might think about entering a zero, or a 9999 in the column, but neither of these values would really be the employee number of Sam's boss. No data value is applicable to this row.

SQL supports missing, unknown, or inapplicable data explicitly, through the concept of a *null value.* A null value is an *indicator* that tells SQL (and the user) that the data is missing or not applicable. As a convenience, a missing piece of data is often said to have the value NULL. But the NULL value is not a real data value like 0, 473.83, or "Sam Clark." Instead, it's a signal, or a reminder, that the data value is missing or unknown. Figure 5-3 shows the contents of the SALESREPS table. Note that the QUOTA and REP\_OFFICE values for Tom Snyder's row and the MANAGER value for Sam Clark's row of the table all contain NULL values.



**Figure 5-3:** NULL values in the SALEREPS table

In many situations NULL values require special handling by the DBMS. For example, if the user requests the sum of the QUOTA column, how should the DBMS handle the missing data when computing the sum? The answer is given by a set of special rules that govern NULL value handling in various SQL statements and clauses. Because of these rules, some leading database authorities feel strongly that NULL values should not be used. Others, including Dr. Codd, have advocated the use of multiple NULL values, with distinct indicators for "unknown" and "not applicable" data.

Regardless of the academic debates, NULL values are a well-entrenched part of the ANSI/ISO SQL standard and are supported in virtually all commercial SQL products. They also play an important, practical role in production SQL databases. The special rules that apply to NULL values (and the cases where NULL values are handled inconsistently by various SQL products) are pointed out in the relevant sections of this book.

**Summary**

This chapter described the basic elements of the SQL language. The basic structure of SQL can be summarized as follows:

• The SQL language that is in common use includes about 30 statements, each consisting of a verb and one or more clauses. Each statement performs a single,

- 68 -

specific function.

• SQL-based databases can store various types of data, including text, integers, decimal numbers, floating point numbers, and usually many more vendor-specific data types.

• SQL statements can include expressions that combine column names, constants, and built-in functions, using arithmetic and other vendor-specific operators.

• Variations in data types, constants, and built-in functions make portability of SQL statements more difficult than it may seem at first.

• NULL values provide a systematic way of handling missing or inapplicable data in the SQL language.

**Chapter 6: Simple Queries**

**Overview**

In many ways, queries are the heart of the SQL language. The SELECT statement, which is used to express SQL queries, is the most powerful and complex of the SQL statements. Despite the many options afforded by the SELECT statement, it's possible to start simply and then work up to more complex queries. This chapter discusses the simplest SQL queries—those that retrieve data from a single table in the database.

**The SELECT Statement**

The SELECT statement retrieves data from a database and returns it to you in the form of query results. You have already seen many examples of the SELECT statement in the quick tour presented in Chapter 2. Here are several more sample queries that retrieve information about sales offices:

*List the sales offices with their targets and actual sales.*

SELECT CITY, TARGET, SALES

FROM OFFICES

CITY TARGET SALES

------------- ---------- -----------

Denver $300,000.00 $186,042.00

New York $575,000.00 $692,637.00

Chicago $800,000.00 $735,042.00

Atlanta $350,000.00 $367,911.00

Los Angeles $725,000.00 $835,915.00

*List the Eastern region sales offices with their targets and sales.*

SELECT CITY, TARGET, SALES

FROM OFFICES

WHERE REGION = 'Eastern'

CITY TARGET SALES

------------- ----------- ------------

New York $575,000.00 $692,637.00

Chicago $800,000.00 $735,042.00

- 69 -

Atlanta $350,000.00 $367,911.00

*List Eastern region sales offices whose sales exceed their targets, sorted in alphabetical order by city.*

SELECT CITY, TARGET, SALES

FROM OFFICES

WHERE REGION = 'Eastern'

AND SALES > TARGET

ORDER BY CITY

CITY TARGET SALES

------------- ----------- ----------

Atlanta $350,000.00 $367,911.00

New York $575,000.00 $692,637.00

*What are the average target and sales for Eastern region offices?*

SELECT AVG(TARGET), AVG(SALES)

FROM OFFICES

WHERE REGION = 'Eastern'

AVG(TARGET) AVG(SALES)

------------- ----------

$575,000.00 $598,530.00

For simple queries, the English language request and the SQL SELECT statement are very similar. When the requests become more complex, more features of the SELECT statement must be used to specify the query precisely.

Figure 6-1 shows the full form of the SELECT statement, which consists of six clauses. The SELECT and FROM clauses of the statement are required. The remaining four clauses are optional. You include them in a SELECT statement only when you want to use the functions they provide. The following list summarizes the function of each clause:



**Figure 6-1:** SELECT statement syntax diagram

• The SELECT clause lists the data items to be retrieved by the SELECT statement. The

- 70 -

items may be columns from the database, or columns to be calculated by SQL as it performs the query. The SELECT clause is described in later sections of this chapter.

• The FROM clause lists the tables that contain the data to be retrieved by the query. Queries that draw their data from a single table are described in this chapter. More

complex queries that combine data from two or more tables are discussed in Chapter 7.

• The WHERE clause tells SQL to include only certain rows of data in the query results. A *search condition* is used to specify the desired rows. The basic uses of the WHERE

clause are described later in this chapter. Those that involve subqueries are discussed in Chapter 9.

• The GROUP BY clause specifies a summary query. Instead of producing one row of query results for each row of data in the database, a summary query groups together

similar rows and then produces one summary row of query results for each group. Summary queries are described in Chapter 8.

• The HAVING clause tells SQL to include only certain groups produced by the GROUP

BY clause in the query results. Like the WHERE clause, it uses a search condition to specify the desired groups. The HAVING clause is described in Chapter 8.

• The ORDER BY clause sorts the query results based on the data in one or more

columns. If it is omitted, the query results are not sorted. The ORDER BY clause is described later in this chapter.

**The SELECT Clause**

The SELECT clause that begins each SELECT statement specifies the data items to be retrieved by the query. The items are usually specified by a *select list,* a list of *select items* separated by commas. Each select item in the list generates a single column of query results, in left-to-right order. A select item can be:

• a *column name*, identifying a column from the table(s) named in the FROM clause. When a column name appears as a select item, SQL simply takes the value of that

column from each row of the database table and places it in the corresponding row of query results.

• a *constant*, specifying that the same constant value is to appear in every row of the query results.

• a *SQL expression*, indicating that SQL must calculate the value to be placed into the query results, in the style specified by the expression.

Each type of select item is described later in this chapter.

**The FROM Clause**

The FROM clause consists of the keyword FROM, followed by a list of table specifications separated by commas. Each table specification identifies a table containing data to be retrieved by the query. These tables are called the *source tables* of the query (and of the SELECT statement) because they are the source of all of the data in the query results. All of the queries in this chapter have a single source table, and every FROM clause contains a single table name.

**Query Results**

- 71 -

The result of a SQL query is always a table of data, just like the tables in the database. If you type a SELECT statement using interactive SQL, the DBMS displays the query results in tabular form on your computer screen. If a program sends a query to the DBMS using programmatic SQL, the table of query results is returned to the program. In either case, the query results always have the same tabular, row/column format as the actual tables in the database, as shown in Figure 6-2. Usually the query results will be a table with several columns and several rows. For example, this query produces a table of three columns (because it asks for three items of data) and ten rows (because there are ten salespeople):



**Figure 6-2:** The tabular structure of SQL query results

*List the names, offices, and hire dates of all salespeople.*

SELECT NAME, REP\_OFFICE, HIRE\_DATE

FROM SALESREPS

NAME REP\_OFFICE HIRE\_DATE

--------------- ---------- ---------

Bill Adams 13 12-FEB-88

Mary Jones 11 12-OCT-89

Sue Smith 21 10-DEC-86

Sam Clark 11 14-JUN-88

Bob Smith 12 19-MAY-87

Dan Roberts 12 20-OCT-86

Tom Snyder NULL 13-JAN-90

Larry Fitch 21 12-OCT-89

Paul Cruz 12 01-MAR-87

Nancy Angelli 22 14-NOV-88

In contrast, the following query produces a single row because only one salesperson has the requested employee number. Even though this single row of query results looks less "tabular" than the multi-row results, SQL still considers it to be a table of three columns and one row.

*What are the name, quota, and sales of employee number 107?*

SELECT NAME, QUOTA, SALES

FROM SALESREPS

WHERE EMPL\_NUM = 107

- 72 -

NAME QUOTA SALES

-------------- ---------- -----------

Nancy Angelli $300,000.00 $186,042.00

In some cases the query results can be a single value, as in the following example: *What are the average sales of our salespeople?*

SELECT AVG(SALES)

FROM SALESREPS

AVG(SALES)

------------

$289,353.20

These query results are still a table, although it's a very small one consisting of one column and one row.

Finally, it's possible for a query to produce *zero* rows of query results, as in this example: *List the name and hire date of anyone with sales over $500,000.*

SELECT NAME, HIRE\_DATE

FROM SALESREPS

WHERE SALES > 500000.00

NAME HIRE\_DATE

----------- ---------

Even in this situation, the query results are still a table. This one is an empty table with two columns and zero rows.

Note that SQL's support for missing data extends to query results as well. If a data item in the database has a NULL value, the NULL value appears in the query results when the data item is retrieved. For example, the SALESREPS table contains NULL values in its QUOTA and MANAGER columns. The following query returns these NULL values in the second and third columns of query results:

*List the salespeople, their quotas, and their managers.*

SELECT NAME, QUOTA, MANAGER

FROM SALESREPS

NAME QUOTA MANAGER

-------------- ------------ ----------

Bill Adams $350,000.00 104

Mary Jones $300,000.00 106

Sue Smith $350,000.00 108

Sam Clark $275,000.00 NULL

Bob Smith $200,000.00 106

Dan Roberts $300,000.00 104

Tom Snyder NULL 101

Larry Fitch $350,000.00 106

- 73 -

Paul Cruz $275,000.00 104

Nancy Angelli $300,000.00 108

The fact that a SQL query always produces a table of data is very important. It means that the query results can be stored back into the database as a table. It means that the results of two similar queries can be combined to form a larger table of query results. Finally, it means that the query results can themselves be the target of further queries. A relational database's tabular structure thus has a very synergistic relationship with the relational query facilities of SQL. Tables can be queried, and queries produce tables.

**Simple Queries**

The simplest SQL queries request columns of data from a single table in the database. For example, this query requests three columns from the OFFICES table:

*List the location, region, and sales of each sales office.*

SELECT CITY, REGION, SALES

FROM OFFICES

CITY REGION SALES

-------------- ------- ------------

Denver Western $186,042.00

New York Eastern $692,637.00

Chicago Eastern $735,042.00

Atlanta Eastern $367,911.00

Los Angeles Western $835,915.00

The SELECT statement for simple queries like this one includes only the two required clauses. The SELECT clause names the requested columns; the FROM clause names the table that contains them.

Conceptually, SQL processes the query by going through the table named in the FROM clause, one row at a time, as shown in Figure 6-3. For each row, SQL takes the values of the columns requested in the select list and produces a single row of query results. The query results thus contain one row of data for each row in the table.



**Figure 6-3:** Simple query processing (no WHERE clause)

**Calculated Columns**

- 74 -

In addition to columns whose values come directly from the database, a SQL query can include *calculated columns* whose values are calculated from the stored data values. To request a calculated column, you specify a SQL expression in the select list. As discussed in Chapter 5, SQL expressions can involve addition, subtraction, multiplication, and division. You can also use parentheses to build more complex expressions. Of course the columns referenced in an arithmetic expression must have a numeric type. If you try to add, subtract, multiply, or divide columns containing text data, SQL will report an error.

This query shows a simple calculated column:

List the city, region, and amount over/under target for each office.

SELECT CITY, REGION, (SALES - TARGET)

FROM OFFICES

CITY REGION (SALES-TARGET)

-------------- ------ ----------------

Denver Western -$113,958.00

New York Eastern $117,637.00

Chicago Eastern -$64,958.00

Atlanta Eastern $17,911.00

Los Angeles Western $110,915.00

To process the query, SQL goes through the offices, generating one row of query results for each row of the OFFICES table, as shown in Figure 6-4. The first two columns of query results come directly from the OFFICES table. The third column of query results is calculated, row-by-row, using the data values from the current row of the OFFICES table.



**Figure 6-4:** Query processing with a calculated column

Here are other examples of queries that use calculated columns:

*Show the value of the inventory for each product.*

SELECT MFR\_ID, PRODUCT\_ID, DESCRIPTION, (QTY\_ON\_HAND \* PRICE) FROM PRODUCTS

MFR\_ID PRODUCT\_ID DESCRIPTION (QTY\_ON\_HAND\*PRICE) ------ ---------- ---------------- ------------------ REI 2A45C Ratchet Link $16,590.00

- 75 -

ACI 4100Y Widget Remover $68,750.00 QSA XK47 Reducer $13,490.00 BIC 41672 Plate $0.00 IMM 779C 900-lb Brace $16,875.00 ACI 41003 Size 3 Widget $22,149.00 ACI 41004 Size 4 Widget $16,263.00 BIC 41003 Handle $1,956.00

*Show me the result if I raised each salesperson's quota by 3 percent of their year-to-date sales.*

SELECT NAME, QUOTA, (QUOTA + (.03\*SALES))

FROM SALESREPS

NAME QUOTA (QUOTA+(.03\*SALES)) --------------- ----------- ------------------- Bill Adams $350,000.00 $361,037.33 Mary Jones $300,000.00 $311,781.75 Sue Smith $350,000.00 $364,221.50 Sam Clark $275,000.00 $283,997.36 Bob Smith $200,000.00 $204,277.82 Dan Roberts $300,000.00 $309,170.19 Tom Snyder NULL NULL Larry Fitch $350,000.00 $360,855.95 Paul Cruz $275,000.00 $283,603.25 Nancy Angelli $300,000.00 $305,581.26

As mentioned in Chapter 5, many SQL products provide additional arithmetic operations, character string operations, and built-in functions that can be used in SQL expressions. These can appear in select list expressions, as in this DB2 example:

*List the name and month and year of hire for each salesperson.*

SELECT NAME, MONTH(HIRE\_DATE), YEAR(HIRE\_DATE)

FROM SALESREPS

SQL constants can also be used by themselves as items in a select list. This can be useful for producing query results that are easier to read and interpret, as in the example on the bottom of the next page.

List the sales for each city.

SELECT CITY, 'has sales of', SALES

FROM OFFICES

CITY HAS SALES OF SALES

------------ ----------- --------------

Denver has sales of $186,042.00

New York has sales of $692,637.00

Chicago has sales of $735,042.00

Atlanta has sales of $367,911.00

Los Angeles has sales of $835,915.00

The query results appear to consist of a separate "sentence" for each office, but they're

- 76 -

really a table of three columns. The first and third columns contain values from the OFFICES table. The second column always contains the same 12-character text string. This distinction is subtle when the query results are displayed on a screen, but it is crucial in programmatic SQL, when the results are being retrieved into a program and used for calculations.

**Selecting All Columns (SELECT \*)**

Sometimes it's convenient to display the contents of all the columns of a table. This can be particularly useful when you first encounter a new database and want to get a quick understanding of its structure and the data it contains. As a convenience, SQL lets you use an asterisk (\*) in place of the select list as an abbreviation for "all columns":

*Show me all the data in the OFFICES table.*

SELECT \*

FROM OFFICES

OFFICE CITY REGION MGR TARGET SALES ------------------- -------- --- ----------- ----------- 22 Denver Western 108 $300,000.00 $186,042.00 11 New York Eastern 106 $575,000.00 $692,637.00 12 Chicago Eastern 104 $800,000.00 $735,042.00 13 Atlanta Eastern 105 $350,000.00 $367,911.00 21 Los Angeles Western 108 $725,000.00 $835,915.00

The query results contain all six columns of the OFFICES table, in the same left-to-right order as in the table itself.

The ANSI/ISO SQL standard specifies that a SELECT statement can have either an all column selection or a select list, but not both, as shown in Figure 6-1. However, many SQL implementations treat the asterisk (\*) as just another element of the select list. Thus the query:

SELECT \*, (SALES - TARGET)

FROM OFFICES

is legal in most commercial SQL dialects (for example in DB2, Oracle, and SQL Server), but it is not permitted by the ANSI/ISO standard.

The all-columns selection is most appropriate when you are using interactive SQL casually. It should be avoided in programmatic SQL, because changes in the database structure can cause a program to fail. For example, suppose the OFFICES table were dropped from the database and then re-created with its columns rearranged and a new seventh column added. SQL automatically takes care of the database-related details of such changes, but it cannot modify your application program for you. If your program expects a SELECT \* FROM OFFICES query to return six columns of query results with certain data types, it will almost certainly stop working when the columns are rearranged and a new one is added.

These difficulties can be avoided if you write the program to request the columns it needs by name. For example, the following query produces the same results as SELECT \* FROM OFFICES. It is also immune to changes in the database structure, as long as the named columns continue to exist in the OFFICES table:

SELECT OFFICE, CITY, REGION, MGR, TARGET, SALES

FROM OFFICES

- 77 -

**Duplicate Rows (DISTINCT)**

If a query includes the primary key of a table in its select list, then every row of query results will be unique (because the primary key has a different value in each row). If the primary key is not included in the query results, duplicate rows can occur. For example, suppose you made this request:

*List the employee numbers of all sales office managers.*

SELECT MGR

FROM OFFICES

MGR

----

108

106

104

105

108

The query results have five rows (one for each office), but two of them are exact duplicates of one another. Why? Because Larry Fitch manages both the Los Angeles and Denver offices, and his employee number (108) appears in both rows of the OFFICES table. These query results are probably not exactly what you had in mind. If there are four different managers, you might have expected only four employee numbers in the query results.

You can eliminate duplicate rows of query results by inserting the keyword DISTINCT in the SELECT statement just before the select list. Here is a version of the previous query that produces the results you want:

*List the employee numbers of all sales office managers.*

SELECT DISTINCT MGR

FROM OFFICES

MGR

----

104

105

106

108

Conceptually, SQL carries out this query by first generating a full set of query results (five rows) and then eliminating rows that are exact duplicates of one another to form the final query results. The DISTINCT keyword can be specified regardless of the contents of the SELECT list (with certain restrictions for summary queries, as described in Chapter 8).

If the DISTINCT keyword is omitted, SQL does not eliminate duplicate rows. You can also specify the keyword ALL to explicitly indicate that duplicate rows are to be retained, but it is unnecessary since this is the default behavior.

**Row Selection (WHERE Clause)**

- 78 -

SQL queries that retrieve all rows of a table are useful for database browsing and reports, but for little else. Usually you'll want to select only some of the rows in a table and include only these rows in the query results. The WHERE clause is used to specify the rows you want to retrieve. Here are some examples of simple queries that use the WHERE clause:

Show me the offices where sales exceed target.

SELECT CITY, SALES, TARGET

FROM OFFICES

WHERE SALES > TARGET

CITY SALES TARGET

------------ ----------- -----------

New York $692,637.00 $575,000.00

Atlanta $367,911.00 $350,000.00

Los Angeles $835,915.00 $725,000.00

Show me the name, sales, and quota of employee number 105.

SELECT NAME, SALES, QUOTA

FROM SALESREPS

WHERE EMPL\_NUM = 105

NAME SALES QUOTA

----------- ----------- -----------

Bill Adams $367,911.00 $350,000.00

Show me the employees managed by Bob Smith (employee 104).

SELECT NAME, SALES

FROM SALESREPS

WHERE MANAGER = 104

NAME SALES

------------ -----------

Bill Adams $367,911.00

Dan Roberts $305,673.00

Paul Cruz $286,775.00

The WHERE clause consists of the keyword WHERE followed by a search condition that specifies the rows to be retrieved. In the previous query, for example, the search condition is MANAGER = 104. Figure 6-5 shows how the WHERE clause works. Conceptually, SQL goes through each row of the SALESREPS table, one-by-one, and applies the search condition to the row. When a column name appears in the search condition (such as the MANAGER column in this example), SQL uses the value of the column in the current row. For each row, the search condition can produce one of three results:

- 79 -



**Figure 6-5:** Row selection with the WHERE clause

• If the search condition is TRUE, the row is included in the query results. For example, the row for Bill Adams has the correct MANAGER value and is included.

• If the search condition is FALSE, the row is excluded from the query results. For example, the row for Sue Smith has the wrong MANAGER value and is excluded.

• If the search condition has a NULL (unknown) value, the row is excluded from the

query results. For example, the row for Sam Clark has a NULL value for the MANAGER column and is excluded.

Figure 6-6 shows another way to think about the role of the search condition in the WHERE clause. Basically, the search condition acts as a filter for rows of the table. Rows that satisfy the search condition pass through the filter and become part of the query results. Rows that do not satisfy the search condition are trapped by the filter and excluded from the query results.



**Figure 6-6:** The WHERE clause as a filter

**Search Conditions**

SQL offers a rich set of search conditions that allow you to specify many different kinds of queries efficiently and naturally. Five basic search conditions (called *predicates* in the ANSI/ISO standard) are summarized here and are described in the sections that follow:

• *Comparison test*. Compares the value of one expression to the value of another

expression. Use this test to select offices in the Eastern region, or salespeople whose sales are above their quotas.

• *Range test*. Tests whether the value of an expression falls within a specified range of

values. Use this test to find salespeople whose sales are between $100,000 and $500,000.

• *Set membership test.* Checks whether the value of an expression matches one of a

set of values. Use this test to select offices located in New York, Chicago, or Los Angeles.

- 80 -