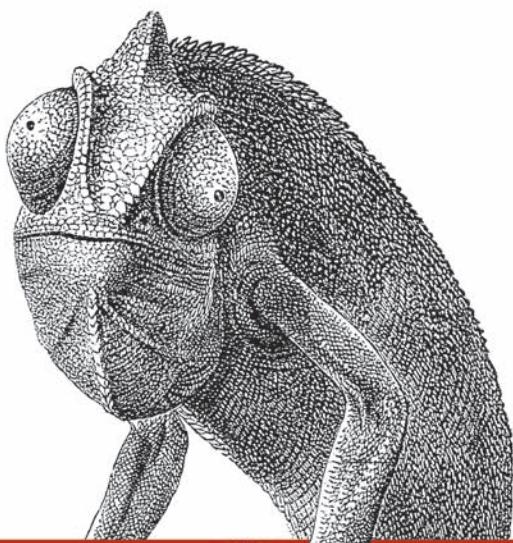


3rd Edition

Covers MySQL, Oracle,
PostgreSQL, and SQL Server



SQL

IN A NUTSHELL

A Desktop Quick Reference

O'REILLY®

Kevin E. Kline
with Daniel Kline & Brand Hunt

SQL IN A NUTSHELL



SQL in a Nutshell is the essential reference for the SQL language used in today's most popular database products. This new edition clearly documents every SQL command according to the latest ANSI standard, and details how those commands are implemented in Microsoft SQL Server 2008, Oracle 11g, MySQL 5.1, and PostgreSQL 8.3. You'll also get an overview of the Relational Database Management System (RDBMS) model and a clear-cut explanation of foundational RDBMS concepts. This comprehensive book provides:

- Background on the Relational Database Model, including current and previous SQL standards
- The fundamental concepts necessary for understanding relational databases and SQL commands
- An alphabetical command reference to SQL statements, according to the SQL2003 ANSI standard
- The implementation of each command by MySQL, Oracle, PostgreSQL, and SQL Server
- An alphabetical reference of the ANSI SQL2003 functions, as well as the vendor implementations
- Platform-specific functions unique to each implementation

Beginning where vendor documentation ends, *SQL in a Nutshell*, Third Edition, distills the experiences of professional database administrators and developers who have used SQL variants to support complex enterprise applications. Whether SQL is new to you or you've been using it since its earliest days, you'll get lots of tips and techniques in this concise and easy-to-use book.

Kevin E. Kline is the director of SQL Server Solutions at Quest Software, a provider of tools for database management and application monitoring.

Daniel Kline is an associate professor and chair of the graduate program in English at the University of Alaska, Anchorage.

Brand Hunt is a director in architecture and engineering at Merrill Lynch.

O'REILLY®

www.oreilly.com

US \$44.99

CAN \$44.99

ISBN: 978-0-596-51884-4



9 780596 518844

Safari
Books Online

Free online edition
for 45 days with
purchase of this book.
Details on last page.

SQL

IN A NUTSHELL

Third Edition

Kevin E. Kline

with Daniel Kline and Brand Hunt

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

SQL in a Nutshell, Third Edition

by Kevin E. Kline with Daniel Kline and Brand Hunt

Copyright © 2009 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editors: Julie Steele and Mary Treseler

Production Editor: Rachel Monaghan

Copyeditor: Rachel Head

Indexer: Angela Howard

Production Services: Octal Publishing, Inc.

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Jessamyn Read

Printing History:

January 2001: First Edition.

September 2004: Second Edition.

November 2008: Third Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *In a Nutshell* series designations, *SQL in a Nutshell*, the image of a chameleon, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-51884-4

[M]

Table of Contents

Preface	v
1. SQL History and Implementations	1
The Relational Model and ANSI SQL	2
History of the SQL Standard	10
SQL Dialects	14
2. Foundational Concepts	16
Database Platforms Described in This Book	16
Categories of Syntax	17
SQL2003 and Platform-Specific Datatypes	29
Constraints	50
3. SQL Statement Command Reference	59
How to Use This Chapter	59
SQL Platform Support	60
SQL Command Reference	63
4. SQL Functions	437
Types of Functions	437
ANSI SQL Aggregate Functions	438
ANSI SQL Window Functions	455
ANSI SQL Scalar Functions	463
Platform-Specific Extensions	483

Appendix: Shared and Platform-Specific Keywords	547
Index	555



Preface

Since its first incarnation in the 1970s, the Structured Query Language (SQL) has been developed hand in hand with the information boom, and as a result, it is the most widely used database manipulation language in business and industry. A number of different software companies and program developers, including those in the open source movement (<http://www.opensource.org>), have concurrently developed their own SQL dialects in response to specific professional needs. All the while, standards bodies have developed a growing list of common features.

SQL in a Nutshell, Third Edition, describes the latest ANSI standard, SQL2003 (SQL3) version of each SQL command, and then documents each platform's implementation of that command. In this book, you will find a concise explanation of the relational database management system (RDBMS) model, a clear-cut explanation of foundational RDBMS concepts, and thorough coverage of SQL syntax and commands.

Most importantly, at least if you're a programmer or developer, *SQL in a Nutshell*, Third Edition, provides a concise guide both to the most popular commercial database packages on the market (Microsoft SQL Server and Oracle). It is also the guide for two of the best-known open source database products (MySQL and PostgreSQL). The attention this book pays to open source SQL platforms recognizes the growing importance of the open source movement within the computing community.

The SQL syntax covered in this book includes:

- ANSI SQL2003 (also known as SQL3) standard syntax
- MySQL version 5.1
- Oracle Database 11g
- PostgreSQL version 8.2.1
- Microsoft SQL Server 2008

Why This Book?

The primary source of information for relational databases is the documentation and help files provided by the vendors themselves. While each vendor's documentation is an indispensable resource that most database programmers and database administrators turn to first, this documentation has a number of limitations:

- It describes the vendor's implementation of SQL without giving you any context as to how well that implementation meets the ANSI standard for SQL.
- It covers only a single, specific vendor's product. There is no coverage of translation, migration, or integration issues.
- It typically describes programming methods in a multitude of small, disconnected documents or help files.
- It covers individual commands, often in confusing detail, obscuring the simple and direct uses of commands that programmers and administrators use every day.

In other words, the documentation included with a vendor's database provides an exhaustive explanation of every aspect of that particular vendor's platform. This is only natural; after all, help texts are geared toward delivering the main facts about a product. They'll tell you a command's specific syntax (and all its obscure variants) and, in general terms, how to implement it. However, if you move between RDBMSs and you need to be productive very quickly, you will rarely use those obscure command variations; instead, you'll utilize the capabilities most common in real-life situations.

This book begins where the vendor documentation ends by distilling the experiences of professional database administrators and developers who have used these SQL variants day in and day out to support complex enterprise applications. It offers you the benefit of their experience in a compact and easily usable format. Whether SQL is new to you or you have been using SQL since its earliest days, there are always new tips and techniques to learn. And when you're moving between different implementations, it's always important to find out about the issues that can bite you if you're not careful and informed.

Who Should Read This Book?

SQL in a Nutshell, Third Edition, benefits several groups of users. It will be useful for programmers who require a concise and handy SQL reference tool; for developers who need to migrate from one SQL dialect to another; and for database administrators (DBAs) who need to both execute a myriad of SQL statements to keep their enterprise databases up and running, and create and manage objects such as tables, indexes, and views.

This book is a reference work, not a tutorial. The writing is not expository. For example, we won't explain the concept of an elementary loop. Experienced developers already know such things—you want the meat. So we will explain, for example, the detailed workings of an ANSI standard cursor, how it works on each of the database platforms we cover, the special capabilities of cursors on each database platform, and the various pitfalls of cursors and how to get around them.

While we don't intend for *SQL in a Nutshell*, Third Edition, to be a tutorial on SQL or a handbook for database design, we do provide some brief coverage of introductory topics, and we hope you'll find that helpful. Chapter 1 and Chapter 2 provide a concise introduction to SQL, covering the general origins, essential structure, and basic operation of the language. If you're new to SQL, these chapters will help you get started.

How This Book Is Organized

SQL in a Nutshell, Third Edition, is divided into four chapters and one appendix:

Chapter 1, *SQL History and Implementations*

Discusses the relational database model, describes the current and previous SQL standards, and introduces the SQL implementations covered in this book.

Chapter 2, *Foundational Concepts*

Describes the fundamental concepts necessary for understanding relational databases and SQL commands.

Chapter 3, *SQL Statement Command Reference*

Provides an alphabetical command reference to SQL statements. This chapter details the latest ANSI standard (SQL3) for each command, as well as the implementation of each command by MySQL, Oracle, PostgreSQL, and SQL Server.

Chapter 4, *SQL Functions*

Provides an alphabetical reference of the ANSI SQL3 functions, describing vendor implementations of all SQL3 functions. In addition, this chapter includes coverage of all platform-specific functions that are unique to each implementation.

Appendix, *Shared and Platform-Specific Keywords*

Provides a table of keywords declared in SQL3 and by the different database platforms. You can use this table to look for words that you should not use for object or variable names.

How to Use This Book

SQL in a Nutshell, Third Edition, is primarily a command reference. As a consequence, you'll probably use it to look up a variety of SQL commands and functions. However, with documentation for the ANSI standard itself plus four database platforms, the description for each command has the potential to get very large.

In order to reduce the verbiage describing each command, we compare each platform's implementation to the SQL3 standard. If the platform supports a clause described in the SQL3 discussion, we won't repeat that clause again.

Generic and transportable examples are provided within the body of each SQL3 command description. Since the SQL3 standard is ahead of most database platforms, examples aren't provided for elements of the SQL3 commands that are not

supported by any platform discussed in this book. In addition, more examples are provided for each database platform that highlight unique extensions and enhancements, of which there are many.

We recognize that our approach may necessitate jumping from a description of a platform's implementation of a command back to the corresponding SQL3 command description. However, we felt that this was better than packing the book with hundreds of pages of redundant content.

Resources

The following websites provide additional information about the various platforms covered in this book:

MySQL

The corporate resource for MySQL is <http://www.mysql.com>, and another good site is <http://dev.mysql.com/doc/refman/5.1/en/>. A great developer resource with lots of useful tips is Devshed.com. See <http://www.devshed.com/c/b/MySQL/> for MySQL-specific information.

PostgreSQL

The home for this open source database is located at <http://www.postgresql.org>. In addition to making a great deal of useful information available for download, this site also maintains mailing lists for PostgreSQL users. Another PostgreSQL site worth investigating is <http://www.pgsql.com>, which offers support for commercial customers.

Oracle

Oracle's cyberspace home is <http://www.oracle.com>. A great resource for hardcore Oracle users is <http://www.oracle.com/technology/>. You can also find all Oracle documentation at <http://www.oracle.com/technology/documentation/index.html>. For useful independent information about Oracle, be sure to check out the Independent Oracle User Group at <http://www.ioug.org>.

SQL Server

The official Microsoft SQL Server website is <http://www.microsoft.com/sql/>. Another good resource is found at the home of the Professional Association for SQL Server (PASS) at <http://www.sqlpass.org>.

Changes in the Third Edition

One of the biggest reasons to release a new edition of a technology book is because the technology has progressed. Since the second edition of this book was published, the ANSI standard has released one new version and all of the database platforms it covers have delivered at least one major release. Consequently, our readers want fresh content on the latest versions of SQL in the marketplace today.

Here are more details about changes in this third edition:

Reduced footprint

The readership of *SQL in a Nutshell*, Second Edition, loved its expansive coverage of all major database platforms. However, maintaining such a huge amount of content proved to be too difficult for the return on the investment. Therefore, based upon the results of a large readership survey, we decided to remove the two least popular database platforms from this edition: Sybase Adaptive Server and IBM's DB2 UDB.

Improved organization

The second edition did a great job of presenting everything readers could want to know about all of the commands and functions available in SQL and the major database platforms, but that didn't mean the content was always easy to find or navigate. We've added better indexes, tables of content, and page headers and footers so you can navigate much more quickly and effectively.

More examples

It's impossible to have too many examples. We've added to our already large set of basic examples, including more sample code that highlights the unique and powerful capabilities of the SQL standard and the extensions offered by each database platform.

Conventions Used in This Book

This book uses the following typographical conventions:

Constant width

Used to indicate programming syntax, code fragments, and examples.

Constant width italic

Used to indicate variables in code that should be replaced with user-supplied values.

Constant width bold

Used in code sections to highlight portions of the code.

Italic

Used to introduce new terms, for emphasis, to indicate commands or user-specified file and directory names, and to indicate variables within text.

Bold

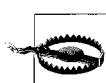
Used to display the names of database objects, such as tables, columns, and stored procedures.

UPPERCASE ITALIC

Used to indicate SQL keywords when they appear in the text.



Indicates a tip, suggestion, or general note.



Indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from this book *does* require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*SQL in a Nutshell*, Third Edition, by Kevin E. Kline with Daniel Kline and Brand Hunt. Copyright 2009 O'Reilly Media, Inc., 978-0-596-51884-4.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

How to Contact Us

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). We want to hear from you, especially with information that will make this book better. Please let us know about any errors you find, as well as your suggestions for future editions, by writing to:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the U.S. or Canada)
707-829-0515 (international/local)
707-829-0104 (fax)

We have a website for this book, where we'll list any examples, errata, or plans for future editions. You can access this page at:

<http://www.oreilly.com/catalog/9780596518844/>

Please help us out by pointing out any typos or syntactical errors that you encounter. (You can imagine how hard it is to proofread a book covering the ANSI standard and four separate products.) You may also ask technical questions or comment on the book by sending an email to:

bookquestions@oreilly.com

For more information about our books, conferences, software, Resource Centers, and the O'Reilly Network, see the O'Reilly website:

<http://www.oreilly.com>

Safari® Books Online



When you see a Safari® Books Online icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at <http://safari.oreilly.com>.

Acknowledgments

We'd like to take a moment to thank a few special individuals at O'Reilly Media. First, we owe a huge debt of gratitude to Julie Steele, the editor of this third edition. Julie helped keep our noses to the grindstone and ensured that we finished what we started. With her helpful and relaxed work style, Julie was always a pleasure to work with. Thank you for all you've done for us!

We also owe a debt to our fine technical reviewers. To Peter Gulutzan (SQL Standard, from the second edition), Thomas Lockhart (PostgreSQL), Ronald Bradford (Oracle/MySQL), and Richard Sonnen (Oracle): we owe you a hearty thanks! Your contributions have greatly improved the accuracy, readability, and value of this book. Without you, our sections on each of the language extensions would have been on shaky ground. In addition, we'd like to tip our hat to Peter Gulutzan and Trudy Pelzer for their book *SQL-99 Complete, Really!* (R&D), which helped us understand the ANSI SQL3 standards.

Kevin E. Kline's Acknowledgments

Many people helped deliver the big, thick book you hold in your hands. This note expresses our appreciation to those who helped make this book a reality.

First of all, a big thanks to my two awesome coauthors, Dan and Brand. You guys are amazing and a pleasure to work with. Next, Julie Steele, our editor at O'Reilly Media, gets a big hug for all of her help. You helped keep us on task and on track. Thank you!

To all of my colleagues at Quest Software go very big thanks for your support and encouragement. Christian Hasker, Andy Grant, Heather Eichman, David Gugick, Billy Bosworth, Douglas Chrystall, David Swanson, Jason Hall, Ariel Weil, and my many other friends at Quest Software: thank you for making these last six years with Quest Software such a blast.

Here's a dedication to my loved ones, Dylan, Emily, Anna, and Katie. You were my hope and breath and light when it seemed that no hope or breath or light remained anywhere in the world. Thank you for loving me so completely and so selflessly. And finally, to Rachel, more precious than jewels and more valuable than rubies, your love has restored my heart and my faith.

Daniel Kline's Acknowledgments

I'd like to thank my brother, Kevin, for his continued willingness to work with me; my colleagues at the University of Alaska Anchorage for their suggestions; and the users of the first two editions of *SQL in a Nutshell* for their honest feedback and useful critiques. We've also received some terrific feedback from the different translators of the first two editions, and I'd like to thank them for their help as well.

Brand Hunt's Acknowledgments

To my wife, Michelle: without your continued support and love, I wouldn't be a part of this project. I'm appreciative of every moment we've shared and of your forgiveness for my keeping you awake at night with the "tappity-tap-taps" emanating from the computer.

Thanks also to my parents, Rex and Jackie, the two biggest influences in everything I've ever done correctly—especially those things that frequently take multiple attempts (like writing!).

A huge thanks to my teammates, Kevin, Daniel, and Jonathan for letting me participate in this project and exercising so much patience in tutoring a first-time O'Reilly author. Your professionalism, work ethic, and ability to make the most tedious tasks fun is so admirable I plan to steal it and adopt it as my own!

Mad props to my friends and colleagues at Rogue Wave Software, ProWorks, NewCode Technology, and Systems Research and Development, for providing the ultimate sandbox for refining skills in SQL, databases, business, software development, writing, and friendship: Gus Waters, Greg Koerper, Marc Manley, Wendi Minne, Erin Foley, Elaine Cull, Randall Robinson, Dave Ritter, Edin Zulic, David Noor, Jim Shur, Chris Mosbrucker, Dan Robin, Mike Faux, Jason Prothero, Tim Romanowski, Andy Mosbrucker, Jeff Jonas, Jeff Butcher, Charlie Barbour, Steve Dunham, Brian Macy, and Ze'ev Mehler.



1

SQL History and Implementations

In the early 1970s, the seminal work of IBM research fellow Dr. E. F. Codd led to the development of a relational data model product called SEQUEL, or *Structured English Query Language*. SEQUEL ultimately became SQL, or *Structured Query Language*.

IBM, along with other relational database vendors, wanted a standardized method for accessing and manipulating data in a relational database. Although IBM was the first to develop relational database theory, Oracle was first to market the technology. Over time, SQL proved popular enough in the marketplace to attract the attention of the American National Standards Institute (ANSI), which released standards for SQL in 1986, 1989, 1992, 1999, 2003, and 2006. This text covers the ANSI 2003 standard because the 2006 standard deals with elements of SQL outside the scope of the commands described in this book. (In essence, the SQL2006 standard describes how XML would be used in SQL.)

Since 1986, various competing languages have allowed programmers and developers to access and manipulate relational data. However, few were as easy to learn or as universally accepted as SQL. Programmers and administrators now have the benefit of being able to learn a single language that, with minor adjustments, is applicable to a wide variety of database platforms, applications, and products.

SQL in a Nutshell, Third Edition, provides the syntax for five common implementations of SQL2003 (SQL3):

- The ANSI SQL standard
- MySQL version 5.1
- Oracle Database 11g
- PostgreSQL version 8.3
- Microsoft's SQL Server 2008

The Relational Model and ANSI SQL

Relational database management systems (RDBMSs) such as those covered in this book are the primary engines of information systems worldwide, and particularly of web applications and distributed client/server computing systems. They enable a multitude of users to quickly and simultaneously access, create, edit, and manipulate data without impacting other users. They also allow developers to write useful applications to access their resources and provide administrators with the capabilities they need to maintain, secure, and optimize organizational data resources.

An RDBMS is defined as a system whose users view data as a collection of tables related to each other through common data values. Data is stored in *tables*, which are composed of *rows* and *columns*. Tables of independent data can be linked (or *related*) to one another if they each have unique, identifying columns of data (called *keys*) that represent data values held in common. E. F. Codd first described relational database theory in his landmark paper “A Relational Model of Data for Large Shared Data Banks,” published in the *Communications of the ACM* (Association for Computing Machinery) in June, 1970. Under Codd’s new relational data model, data was *structured* (into tables of rows and columns); *manageable* using operations such as selections, projections, and joins; and *consistent* as the result of integrity rules such as keys and referential integrity. Codd also articulated rules that governed how a relational database should be designed. The process for applying these rules is now known as *normalization*.

Codd’s Rules for Relational Database Systems

Codd applied rigorous mathematical theories (primarily set theory) to the management of data, and he compiled a list of criteria a database must meet to be considered relational. At its core, the relational database concept centers around storing data in tables. This concept is now so common as to seem trivial; however, not long ago the goal of designing a system capable of sustaining the relational model was considered a long shot with limited usefulness.

Following are Codd’s *Twelve Principles of Relational Databases*:

1. Information is represented logically in tables.
2. Data must be logically accessible by table, primary key, and column.
3. Null values must be uniformly treated as “missing information,” not as empty strings, blanks, or zeros.
4. Metadata (data about the database) must be stored in the database just as regular data is.
5. A single language must be able to define data, views, integrity constraints, authorization, transactions, and data manipulation.
6. Views must show the updates of their base tables and vice versa.
7. A single operation must be available to do each of the following operations: retrieve data, insert data, update data, or delete data.
8. Batch and end-user operations are logically separate from physical storage and access methods.

9. Batch and end-user operations can change the database schema without having to recreate it or the applications built upon it.
10. Integrity constraints must be available and stored in the metadata, not in an application program.
11. The data manipulation language of the relational system should not care where or how the physical data is distributed and should not require alteration if the physical data is centralized or distributed.
12. Any row processing done in the system must obey the same integrity rules and constraints that set-processing operations do.

These principles continue to be the litmus test used to validate the “relational” characteristics of a database platform; a database that does not meet all of these rules is not fully relational. While these rules do not apply to applications development, they do determine whether the database engine itself can be considered truly “relational.” Currently, most commercial RDBMS products pass Codd’s test. Among the platforms discussed in *SQL in a Nutshell*, Third Edition, only MySQL failed to support all of these requirements, and only then in releases prior to the one covered in this book.

Understanding Codd’s principles assists programmers and developers in the proper development and design of relational databases (RDBs). The following sections detail how some of these requirements are met within SQL using RDBs.

Data structures (rules 1, 2, and 8)

Codd’s rules 1 and 2 state that “information is represented logically in tables” and that “data must be logically accessible by table, primary key, and column.” So, the process of defining a table for a relational database does not require that programs instruct the database how to interact with the underlying physical data structures. Furthermore, SQL logically isolates the processes of accessing data and physically maintaining that data, as required by rule 8: “batch and end-user operations are logically separate from physical storage and access methods.”

In the relational model, data is shown logically as a two-dimensional *table* that describes a single entity (for example, business expenses). Academics refer to tables as *entities* and to columns as *attributes*. Tables are composed of *rows*, or *records* (academics call them *tuples*), and *columns* (called *attributes*, since each column of a table describes a specific attribute of the entity). The intersection of a record and a column provides a single *value*. The column or columns whose values uniquely identify each record can act as a *primary key*. These days this representation seems elementary, but it was actually quite innovative when it was first proposed.

SQL3 defines a whole data structure hierarchy beyond simple tables, though tables are the core data structure. Relational design handles data on a table-by-table basis, not on a record-by-record basis. This table-centric orientation is the heart of set programming. Consequently, almost all SQL commands operate much more efficiently against sets of data within or across tables than against individual records. Said another way, effective SQL programming requires that you think in terms of sets of data, rather than of individual rows.

Figure 1-1 is a description of the SQL3 terminology used to describe the hierarchical data structures used by a relational database: *clusters* contain sets of *catalogs*; *catalogs* contain sets of *schemas*; *schemas* contain sets of *objects*, such as *tables* and *views*; and *tables* are composed of sets of *columns* and *records*.

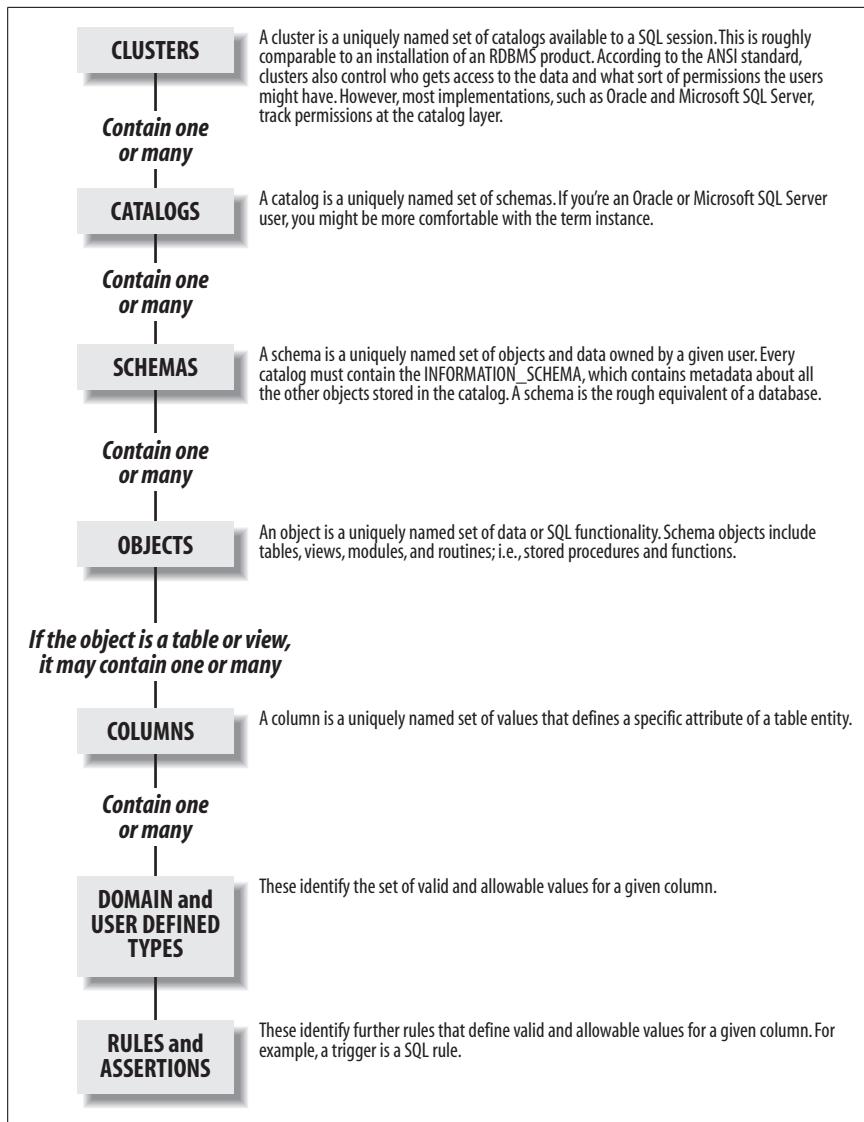


Figure 1-1. SQL3 dataset hierarchy

For example, in a **Business_Expense** table, a column called **Expense_Date** might show when an expense was incurred. Each record in the table describes a specific entity; in this case, everything that makes up a business expense (when it happened, how much it cost, who incurred the expense, what it was for, and so on).

Each attribute of an expense—in other words, each column—is supposed to be *atomic*; that is, each column is supposed to contain one, and only one, value. If a table is constructed in which the intersection of a row and column can contain more than one distinct value, one of SQL’s primary design guidelines has been violated. (Some of the database platforms discussed in this book do allow you to place more than one value into a column, via the `VARRAY` or `TABLE` datatypes.)

Rules of behavior are specified for column values. Foremost is that column values must share a common *domain*, better known as a *datatype*. For example, if the `Expense_Date` field is defined as having a `DATE` datatype, the value `ELMER` should not be placed into that field because it is a string, not a date, and the `Expense_Date` field can contain only dates. In addition, SQL3 allows further control of column values through the application of *constraints* (discussed in detail in Chapter 2) and *assertions*. A SQL constraint might, for instance, limit `Expense_Date` to expenses less than a year old. Additionally, data access for all individuals and computer processes is controlled at the schema level by an *AuthorizationID* or *user*. Permissions to access or modify specific sets of data may be granted or restricted on a per-user basis.

SQL databases also employ *character sets* and *collations*. Character sets are the “symbols” or “alphabets” used by the “language” of the data. For example, the American English character set does not contain the special character for *ñ* in the Spanish character set. Collations are sets of sorting rules that operate on a character set. A collation defines how a given data manipulation operation sorts data. For example, an American English character set might be sorted either by *character-order, case-insensitive*, or by *character-order, case-sensitive*.



The ANSI standard does not say how sorts should be done, only that platforms must provide common collations found in a particular language.

It is important to know what collation you are using when writing SQL code against a database platform, as it can have a direct impact on how queries behave, and particularly on the behavior of the `WHERE` and `ORDER BY` clauses of `SELECT` statements. For example, a query that sorts data using a binary collation will return data in a very different order than one that sorts data using, say, an American English collation.

NULLs (rule 3)

Most databases allow any of their supported datatypes to store `NULL` values. Inexperienced SQL programmers and developers tend to think of `NULL` as zero or blank. In fact, `NULL` is neither of these. In SQL3, `NULL` literally means that the value is unknown or indeterminate. (This question alone—whether `NULL` should be considered unknown or indeterminate—is the subject of much academic debate.) This differentiation enables a database designer to distinguish between those entries that represent a deliberately placed zero, for example, and those where either the data is not recorded in the system or a `NULL` has been explicitly entered. As an illustration of this semantic difference, consider a system that tracks payments. If a product has a `NULL` price, that does not mean the product

is free; instead, a NULL price indicates that the amount is not known or perhaps has not yet been determined.



There is a good deal of differentiation between the database platforms in terms of how they handle NULL values. This leads to some major porting issues between those platforms relating to NULLs. For example, an empty string (i.e., a NULL string) is inserted as a NULL value on Oracle. All the other databases covered in this book permit the insertion of an empty string into VARCHAR and CHAR columns.

One side effect of the indeterminate nature of a NULL value is that it cannot be used in a calculation or a comparison. Here are a few brief but very important rules, from the ANSI standard, to remember about the behavior of NULL values when dealing with NULLs in SQL statements:

- A NULL value cannot be inserted into a column defined as NOT NULL.
- NULL values are not equal to each other. It is a frequent mistake to compare two columns that contain NULL and expect the NULL values to match. (The proper way to identify a NULL value in a WHERE clause or in a Boolean expression is to use phrases such as “value IS NULL” and “value IS NOT NULL”.)
- A column containing a NULL value is ignored in the calculation of aggregate values such as AVG, SUM, or MAX COUNT.
- When columns that contain NULL values are listed in the GROUP BY clause of a query, the query output contains a single row for NULL values. In essence, the ANSI standard considers all NULLs found to be in a single group.
- DISTINCT and ORDER BY clauses, like GROUP BY, also see NULL values as indistinguishable from each other. With the ORDER BY clause, the vendor is free to choose whether NULL values sort high (first in the result set) or sort low (last in the result set) by default.

Metadata (rules 4 and 10)

Codd's fourth rule for relational databases states that data about the database must be stored in standard tables, just as all other data is. Data that describes the database itself is called *metadata*. For example, every time you create a new table or view in a database, records are created and stored that describe the new table. Additional records are needed to store any columns, keys, or constraints on the table. This technique is implemented in most commercial and open source SQL database products. For example, SQL Server uses what it calls “system tables” to track all the information about the databases, tables, and database objects in any given database. It also has “system databases” that keep track of information about the server on which the database is installed and configured.

The language (rules 5 and 11)

Codd's rules do *not* require SQL to be used with a relational database. His rules, particularly rules 5 and 11, only specify how the language should behave when coupled with a relational database. At one time SQL competed with other languages (such as Digital's RDO and Fox/PRO) that might have fit the relational bill, but SQL won out, for three reasons. First, SQL is a relatively simple, intuitive, English-like language that handles most aspects of data manipulation. Second, SQL is satisfactorily high-level. A programmer or database administrator (DBA) does not have to spend time ensuring that data is stored in the proper memory registers or that data is cached to disk; the database management system (DBMS) handles that task automatically. Finally, because no single vendor owns SQL, it was adopted across a number of platforms.

Views (rule 6)

A *view* is a virtual table that does not exist as a physical repository of data, but is instead constructed on the fly from a *SELECT* statement whenever that view is queried. Views enable you to construct different representations of the same source data for a variety of audiences without having to alter the way in which the data is stored.



Some vendors support database objects called *materialized views*. Don't let the similarity of terms confuse you; materialized views are not governed by the same rules as ANSI standard views.

Set operations (rules 7 and 12)

Other database manipulation languages, such as the venerable Xbase, perform their data operations quite differently from SQL. These languages require you to tell the program exactly how to treat the data, one record at a time. Since the program cycles down through a list of records, performing its logic on one record after another, this style of programming is frequently called *row processing* or *procedural programming*.

In contrast, SQL programs operate on logical sets of data. Set theory is applied in almost all SQL statements, including *SELECT*, *INSERT*, *UPDATE*, and *DELETE* statements. In effect, data is selected from a set called a "table." Unlike the row-processing style, *set processing* allows a programmer to tell the database simply *what* is required, not *how* each individual piece of data should be handled. Sometimes set processing is referred to as *declarative processing*, since a programmer declares only what data is wanted (as in, "Give me all employees in the southern region who earn more than \$70,000 per year") rather than describing the exact procedure to be used to retrieve or manipulate the data.



Set theory was the brainchild of mathematician Georg Cantor, who developed it at the end of the nineteenth century. At the time, set theory (and Cantor's theory of the infinite) was quite controversial. Today, set theory is such a common part of life that it is learned in elementary school. Things like card catalogs, the Dewey Decimal System, and alphabetized phone books are all simple and common examples of applied set theory.

Examples of set theory in conjunction with relational databases are detailed in the following section.

Codd's Rules in Action: Simple SELECT Examples

Up to this point, this chapter has focused on the individual aspects of a relational database platform as defined by Codd and implemented under ANSI SQL. This section presents a high-level overview of the most important SQL statement, *SELECT*, and some of its most salient points—namely, the relational operations known as *projections*, *selections*, and *joins*:

Projection

Retrieves specific columns of data

Selection

Retrieves specific rows of data

Join

Returns columns and rows from two or more tables in a single result set

Although at first glance it might appear as though the *SELECT* statement deals only with the relational selection operation, in actuality, *SELECT* deals with all three operations.

The following statement embodies the projection operation by selecting the first and last names of an author, plus his home state, from the **authors** table:

```
SELECT au_fname, au_lname, state
  FROM authors
```

The results from any such *SELECT* statement are presented as another table of data:

au_fname	au_lname	state
Johnson	White	CA
Marjorie	Green	CA
Cheryl	Carson	CA
Michael	O'Leary	CA
Meander	Smith	KS
Morningstar	Greene	TN
Reginald	Blotchet-Halls	OR
Innes	del Castillo	MI

The resulting data is sometimes called a *result set*, *work table*, or *derived table*, differentiating it from the *base table* in the database that is the target of the *SELECT* statement.

It is important to note that the relational operation of projection, not selection, is specified using the *SELECT* clause (that is, the keyword *SELECT* followed by a list of expressions to be retrieved) of a *SELECT* statement. Selection—the operation of retrieving specific rows of data—is specified using the *WHERE* clause in a *SELECT* statement. *WHERE* filters out unwanted rows of data and retrieves only the requested rows. Continuing with the previous example, the following statement selects authors from states other than California:

```
SELECT au_fname, au_lname, state
  FROM authors
 WHERE state <> 'CA'
```

Whereas the first query retrieved all authors, the result of this second query is a much smaller subset of records:

au_fname	au_lname	state
Meander	Smith	KS
Morningstar	Greene	TN
Reginald	Blotchet-Halls	OR
Innes	del Castillo	MI

By combining the capabilities of projection and selection in a single query, you can use SQL to retrieve only the columns and records that you need at any given time.

Joins are the next, and last, relational operation we're going to talk about in this section. A join relates one table to another in order to return a result set consisting of related data from both tables.



Different vendors allow you to join varying numbers of tables in a single join operation. For example, Oracle places no limit on the number of tables in a join, while Microsoft SQL Server allows up to 256 tables in a join operation.

The ANSI standard method of performing joins is to use the *JOIN* clause in a *SELECT* statement. An older method, known as a *theta join*, performs the join analysis in the *WHERE* clause. The following example shows both approaches. Each statement retrieves employee information from the **employee** base table as well as job descriptions from the **jobs** base table. The first *SELECT* uses the newer, ANSI *JOIN* clause, while the second *SELECT* uses a theta join:

```
-- ANSI style
SELECT a.au_fname, a.au_lname, t.title_id
  FROM authors AS a
  JOIN titleauthor AS t ON a.au_id = t.au_id
 WHERE a.state <> 'CA'

-- Theta style
SELECT a.au_fname, a.au_lname, t.title_id
  FROM authors AS a,
       titleauthor AS t
 WHERE a.au_id = t.au_id
   AND a.state <> 'CA'
```

For more information about joins, see the “JOIN Subclause” section in Chapter 3.

History of the SQL Standard

In response to the proliferation of SQL dialects, ANSI published its first SQL standard in 1986 to bring about greater conformity among vendors. This was followed by a second, widely adopted standard in 1989. The International Standards Organization (ISO) also approved the SQL standard. ANSI released one update in 1992, known as SQL92 or SQL2, and another in 1999, termed SQL99 or SQL3. The next update, made in 2003, is also referred to as SQL3 (or SQL2003). When we use that term in this book, we are referring to the 2003 revision of the standard.

Each time it revises the SQL standard, ANSI adds new features and incorporates new commands and capabilities into the language. For example, the SQL99 standard added a group of capabilities that handled object-oriented datatype extensions.

What's New in SQL2006

The ANSI standards body that regulates SQL issued a new standard in 2006, in which the important major improvements of SQL3 were retained and augmented. The ANSI SQL2006 release was evolutionary over the SQL3 release, but it did not include any significant changes to the SQL3 commands and functions that were described in the second edition of this book. Instead, SQL2006 described an entirely new functional area of behavior for the SQL standard. Briefly, SQL2006 describes how SQL and XML (the eXtensible Markup Language) interact. For example, the SQL2006 standard describes how to import and store XML data in a SQL database, manipulate that data, and then publish the data both in native XML form and as conventional SQL data wrapped in XML form. The SQL2006 standard provides a means of integrating SQL application code with XQuery, the XML Query Language standardized by the World Wide Web Consortium (W3C). Because XML and XQuery are disciplines in their own right, they are considered beyond the scope of this book and are not covered here.

What's New in SQL2003 (SQL3)

SQL99 had two main parts, *Foundation:1999* and *Bindings:1999*. The SQL3 Foundation section includes all of the Foundation and Bindings standards from SQL99, as well as a new section called *Schemata*.

The Core requirements of SQL3 did not change from Core SQL99, so the database platforms that conformed to Core SQL99 automatically conform to SQL3. Although the Core of SQL3 has no additions (except for a few new reserved words), a number of individual statements and behaviors have been updated or modified. Because these updates are reflected in the individual syntax descriptions of each statement in Chapter 3, we won't spend time on them here.

A few elements of the Core in SQL99 were deleted in SQL3, including:

- The *BIT* and *BIT VARYING* datatypes
- The *UNION JOIN* clause
- The *UPDATE...SET ROW* statement

A number of other features, most of which were or are rather obscure, have also been added, deleted, or renamed. Many of the new features of the SQL3 standard are currently interesting mostly from an academic standpoint, because none of the database platforms support them yet. However, a few new features hold more than passing interest:

Elementary OLAP functions

SQL3 adds an Online Analytical Processing (OLAP) amendment, including a number of windowing functions to support widely used calculations such as moving averages and cumulative sums. Windowing functions are aggregates computed over a window of data: *ROW_NUMBER*, *RANK*, *DENSE_RANK*, *PERCENT_RANK*, and *CUME_DIST*. OLAP functions are fully described in T611 of the standard. Some database platforms are starting to support the OLAP functions. Refer to Chapter 4 for details.

Sampling

SQL3 adds the *TABLESAMPLE* clause to the *FROM* clause. This is useful for statistical queries on large databases, such as a data warehouse.

Enhanced numeric functions

SQL3 adds a large number of numeric functions. In this case, the standard was mostly catching up with the trend in the industry, since one or more database platforms already supported the new functions. Refer to Chapter 4 for details.

Levels of Conformance

SQL99 is built upon SQL92's *levels of conformance*. SQL92 first introduced levels of conformance by defining three categories: *Entry*, *Intermediate*, and *Full*. Vendors had to achieve at least Entry-level conformance to claim ANSI SQL compliance. The U.S. National Institute of Standards and Technology (NIST) later added the *Transitional* level between the Entry and Intermediate levels, so NIST's levels of conformance were Entry, Transitional, Intermediate, and Full, while ANSI's were only Entry, Intermediate, and Full. Each higher level of the standard was a superset of the subordinate level, meaning that each higher level included all the features of the lower levels of conformance.

Later, SQL99 altered the base levels of conformance, doing away with the Entry, Intermediate, and Full levels. With SQL99, vendors must implement all the features of the lowest level of conformance, Core SQL99, in order to claim (and publish) that they are SQL99 ready. Core SQL99 includes the old Entry SQL92 feature set, features from other SQL92 levels, and some brand new features. A vendor may also choose to implement additional feature packages described in the SQL99 standard.

Supplemental Features Packages in the SQL3 Standard

The SQL3 standard represents the ideal, but very few vendors currently meet or exceed the Core SQL3 requirements. The Core standard is like the interstate speed limit: some drivers go above it and others go below it, but few go exactly the speed limit. Similarly, vendor implementations can vary greatly.

Two committees—one within ANSI, the other within ISO, and both composed of representatives from virtually every RDBMS vendor—drafted the supplemental feature definitions described in this section. In this collaborative and somewhat political environment, vendors compromised on exactly which proposed features and implementations would be incorporated into the new standard.

New features in the ANSI standard often are derived from an existing product or are the outgrowth of new research and development in the academic community. Consequently, vendor adoption of specific ANSI standards can be spotty. A relatively new addition to the SQL3 standard is SQL/XML (greatly expanded in SQL2006.) The other parts of the SQL99 standard remain in SQL3, though their names may have changed and they may have been slightly rearranged.

The nine supplemental features packages, representing different subsets of commands, are platform-optional. Some features might show up in multiple packages, while others do not appear in any of the packages. These packages and their features are described in the following list:

Part 1, SQL/Framework

Includes common definitions and concepts used throughout the standard. Defines the way the standard is structured and how the various parts relate to one another, and describes the conformance requirements set out by the standards committee.

Part 2, SQL/Foundation

Includes the Core, an augmentation of the SQL99 Core. This is the largest and most important part of the standard.

Part 3, SQL/CLI (Call-Level Interface)

Defines the call-level interface for dynamically invoking SQL statements from external application programs. Also includes over 60 routine specifications to facilitate the development of truly portable shrink-wrapped software.

Part 4, SQL/PSM (Persistent Stored Modules)

Standardizes procedural language constructs similar to those found in database platform-specific SQL dialects such as PL/SQL and Transact-SQL.

Part 9, SQL/MED (Management of External Data)

Defines the management of data located outside of the database platform using datalinks and a wrapper interface.

Part 10, SQL/OBJ (Object Language Binding)

Describes how to embed SQL statements in Java programs. It is closely related to JDBC, but offers a few advantages. It is also very different from the traditional host language binding possible in early versions of the standard.

Part 11, SQL/Schemata

Defines over 85 views (three more than in SQL99) used to describe the metadata of each database and stored in a special schema called *INFORMATION_SCHEMA*. Updates a number of views that existed in SQL99.

Part 12, SQL/JRT (Java Routines and Types)

Defines a number of SQL routines and types using the Java programming language. Several features of Java, such as Java static methods and classes, are now supported.

Part 14, SQL/XML

Adds a new type called XML, four new operators (`XMLPARSE`, `XMLSERIALIZE`, `XMLROOT`, and `XMLCONCAT`), several new functions (described in Chapter 4), and the new `IS DOCUMENT` predicate. Also includes rules for mapping SQL-related elements (like **identifiers**, **schemas**, and **objects**) to XML-related elements.

Note that parts 5, 6, 7, and 8 do not exist by design.

Be aware that an RDBMS platform may claim SQL3 compliance by meeting Core SQL99 standards, so read the vendor's fine print for a full description of its ANSI conformity features. By understanding what features comprise the nine packages, users can gain a clear idea both of the capabilities of a particular RDBMS and of how the various features behave when SQL code is transported to other database products.

The ANSI standards—which cover retrieval, manipulation, and management of data in commands such as `SELECT`, `JOIN`, `ALTER TABLE`, and `DROP`—formalize many SQL behaviors and syntax structures across a variety of platforms. These standards have become even more important as open source database products, such as MySQL and PostgreSQL, have grown in popularity and begun being developed by virtual teams rather than large corporations.

SQL in a Nutshell, Third Edition explains the SQL implementation of four popular RDBMSs. These vendors do not meet all the SQL3 standards; in fact, all RDBMS platforms play a constant game of tag with the standards bodies. Often, as soon as vendors close in on the standard, the standards bodies update, refine, or otherwise change the benchmark. Conversely, the vendors often implement new features that are not yet a part of the standard but that boost the effectiveness of their users.

SQL3 Statement Classes

Comparing statement classes further delineates SQL3 from SQL92. However, the older terms are still used, so readers need to know them. SQL92 grouped statements into three broad categories:

Data Manipulation Language (DML)

Provides specific data-manipulation commands such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`

Data Definition Language (DDL)

Contains commands that handle the accessibility and manipulation of database objects, including `CREATE` and `DROP`

Data Control Language (DCL)

Contains the permission-related commands `GRANT` and `REVOKE`

In contrast, SQL3 supplies seven core categories, now called *classes*, that provide a general framework for the types of commands available in SQL. These statement “classes” are slightly different from the SQL92 statement categories, because they attempt to identify the statements within each class more accurately and logically and they provide for the development of new features and statement classes. Additionally, the new statement classes now allow some “orphaned” statements that did not fit well into any of the old categories to be properly classified.

Table 1-1 identifies the SQL3 statement classes and lists some of the commands in each class, each of which is fully discussed later. At this point, the key is to remember the statement class titles.

Table 1-1. SQL3 statement classes

Class	Description	Example commands
SQL connection statements	Start and end a client connection	<i>CONNECT, DISCONNECT</i>
SQL control statements	Control the execution of a set of SQL statements	<i>CALL, RETURN</i>
SQL data statements	May have a persistent and enduring effect upon data	<i>SELECT, INSERT, UPDATE, DELETE</i>
SQL diagnostic statements	Provide diagnostic information and raise exceptions and errors	<i>GET DIAGNOSTICS</i>
SQL schema statements	May have a persistent and enduring effect on a database schema and objects within that schema	<i>ALTER, CREATE, DROP</i>
SQL session statements	Control default behavior and other parameters for a session	<i>SET</i> statements like <i>SET CONSTRAINT</i>
SQL transaction statements	Set the starting and ending point of a transaction	<i>COMMIT, ROLLBACK</i>

Those who work with SQL regularly should become familiar with both the old (SQL92) and the new (SQL3) statement classes, since both nomenclatures are still used to refer to SQL features and statements.

SQL Dialects

The constantly evolving nature of the SQL standard has given rise to a number of SQL *dialects* among the various vendors and platforms. These dialects commonly evolve because a given database vendor’s user community requires capabilities in the database before the ANSI committee creates an applicable standard. Occasionally, though, the academic or research communities introduce a new feature in response to pressures from competing technologies. For example, many database vendors are augmenting their current programmatic offerings with either Java (as is the case with DB2, Oracle, and Sybase) or VBScript (as is the case with Microsoft). In the future, programmers and developers will use these programming languages in concert with SQL to build SQL programs.

Many of these dialects include conditional processing capabilities (such as that controlled through *IF...THEN* statements), control-of-flow functions (such as *WHILE* loops), variables, and error-handling capabilities. Because ANSI had not yet developed a standard for these important features at the time users began to demand them, RDBMS developers and vendors created their own commands and syntax. In fact, some of the earliest vendors from the 1980s have variances in the most elementary commands, such as *SELECT*, because their implementations predate the standards. ANSI is now refining standards that address these inconsistencies.

Some of these dialects introduced procedural commands to support the functionality of a more complete programming language. For example, these procedural implementations contain error-handling commands, control-of-flow language, conditional commands, variable-handling commands, support for arrays, and many other extensions. Although these are technically divergent procedural implementations, they are called dialects here. The SQL/PSM (Persistent Stored Module) package provides many features associated with programming stored procedures and incorporates many of the extensions offered by these dialects.

Some popular dialects of SQL include:

PL/SQL

Found in Oracle. PL/SQL stands for Procedural Language/SQL and contains many similarities to the language Ada.

Transact-SQL

Used by both Microsoft SQL Server and Sybase Adaptive Server. As Microsoft and Sybase have moved away from the common platform they shared early in the 1990s, their implementations of Transact-SQL have also diverged.

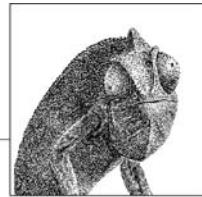
PL/pgSQL

SQL dialect and extensions implemented in PostgreSQL. The acronym stands for Procedural Language/PostgreSQL.

Users who plan to work extensively with a single database system should learn the intricacies of their preferred SQL dialect or platform.

2

Foundational Concepts



SQL provides an easy, intuitive way to interact with a database. While the SQL2003 (SQL3) standard does not define the concept of a “database,” it does define all the functions and concepts needed for a user to create, retrieve, update, and delete data. It is important to know the types of syntax in the SQL2003 standard and the particular platform-specific syntax guidelines. This chapter will provide you with a grounding in those areas.

Database Platforms Described in This Book

SQL in a Nutshell, Third Edition describes the SQL standard and the platform-specific implementations of several leading RDBMSs:

MySQL

MySQL is a popular open source DBMS that is known for its blistering performance. It runs on numerous operating systems, including most Linux variants. To improve performance, it has a slimmer feature set than many other DBMSs. This book covers MySQL 5.1.

Oracle

Oracle is a leading RDBMS in the commercial sector. It runs on a multitude of operating systems and hardware platforms. Its scalable, reliable architecture has made it the platform of choice for many users. In this edition, we cover Oracle Database 11g.

PostgreSQL

PostgreSQL is the most feature-rich open source database platform available. Whereas MySQL is best known for its high-speed performance, PostgreSQL is best known for its excellent support for ANSI standards and robust transaction processing capabilities, as well as its rich datatype and database object support. In addition to its full set of features, PostgreSQL runs on a wide variety of operating systems and hardware platforms. This book covers PostgreSQL 8.2.1.

SQL Server

Microsoft SQL Server is a popular RDBMS that runs only on the Windows platform. Its features include ease of use, an all-inclusive feature set, low cost, and high performance. This book covers Microsoft SQL Server 2008.

Categories of Syntax

To begin to use SQL, readers should understand how statements are written. SQL syntax falls into four main categories. Each category is introduced in the following list and then explained in further detail in the sections that follow:

Identifiers

Describe a user- or system-supplied name for a database object, such as a database, a table, a constraint on a table, a column in a table, a view, etc.

Literals

Describe a user- or system-supplied string or value that is not otherwise an identifier or a keyword. Literals may be strings like “hello”, numbers like 1234, dates like “Jan 01, 2002”, or Boolean values like TRUE.

Operators

Are symbols specifying an action to be performed on one or more expressions, most often in *DELETE*, *INSERT*, *SELECT*, or *UPDATE* statements. Operators are also used frequently in the creation of database objects.

Reserved words and keywords

Have special meaning to the database SQL parser. *Keywords* such as *SELECT*, *GRANT*, *DELETE*, or *CREATE* are words that cannot be used as identifiers within the database platform. These are usually commands or SQL statements. *Reserved words* are words that may become reserved some time in the future. Elsewhere in the book, we use the term *keyword* to describe both concepts. You can circumvent the restriction on using reserved words and keywords as identifiers by using *quoted identifiers*, which will be described in a moment. However, this is not recommended!

Identifiers

Keep in mind that RDBMSs are built upon set theory. In ANSI terms, *clusters* contain sets of catalogs, *catalogs* contain sets of schemas, *schemas* contain sets of objects, and so on. Most database platforms use corollary terms: *instances* contain one or more databases; *databases* contain one or more schemas; and *schemas* contain one or more tables, views, or stored procedures, and the privileges associated with each object. At each level of this structure, items require unique names (that is, identifiers) so that they can be referenced by programs and system processes. This means that each *object* (whether a database, table, view, column, index, key, trigger, stored procedure, or constraint) in an RDBMS must be identified. When issuing the command that creates a database object, you must specify an identifier (i.e., a name) for that new object.

There are two important categories of rules that experienced programmers keep in mind when choosing an identifier for a given item:

Naming conventions

Logical rules of thumb that govern how database designers name objects. Consistently following these rules ultimately creates better database structures and enables improved data tracking. These are not so much SQL requirements as the distilled experience of practiced programmers.

Identifier rules

Naming rules set by the SQL standard and implemented by the platforms. These rules govern characteristics such as how long a name may be. These identifier conventions are covered for each vendor later in this chapter.

Naming conventions

Naming conventions establish a standard baseline for choosing object identifiers. In this section, we present a list of naming conventions (rules for picking your identifiers) that are based on long years of experience. The SQL standard has no comment on naming conventions outside of the uniqueness of an identifier, its length, and the characters that are valid within the identifier. However, here are some conventions that you should follow:

Select a name that is meaningful, relevant, and descriptive

Do not name a table **XP03**; instead, name it **Expenses_2005**, showing that it stores expenses for the year 2005. Remember that other people will likely be using the table or database too, perhaps long after you have gone, and the names you use should make sense at a glance. Each database vendor has limits on object name size, but names generally can be long enough to make sense to anyone reading them.

Choose and apply the same case throughout

Use either all uppercase or all lowercase for all objects throughout the database. Some database servers are case-sensitive, so using mixed-case identifiers might cause problems later.

Use abbreviations consistently

Once you've chosen an abbreviation, use it consistently throughout the database. For example, if you use *EMP* as an abbreviation for *EMPLOYEE*, you should use *EMP* throughout the database; do not use *EMP* in some places and *EMPLOYEE* in others.

Use complete, descriptive, meaningful names with underscores for reading clarity

A column name like *UPPERCASEWITHUNDERSCORES* is not as easy to read as *UPPERCASE_WITH_UNDERSCORES*.

Do not put company or product names in database object names

Companies get acquired, and products change names. These elements are too transitory to be included in database object names.

Do not use overly obvious prefixes or suffixes

For example, don't use *DB_* as a prefix for a database, and don't prefix every view with *V_*. Simple queries to the system table of the database can tell the DBA or database programmer what type of object an identifier represents.

Do not fill up all available space for the object name

If the database platform allows a 32-character table name, try to leave at least a few free characters at the end. Some database platforms append prefixes or suffixes to table names when manipulating temporary copies of the tables.

Do not use quoted identifiers

Quoted identifiers are object names stored within double quotation marks. (The ANSI standard calls these *delimited identifiers*.) Quoted identifiers are also case-sensitive. Encapsulating an identifier within double quotes allows creation of names that may be difficult to use and may cause problems later. For example, users could embed spaces, special characters, mixed-case characters, or even escape sequences within a quoted identifier, but some third-party tools (and even vendor-supplied tools) cannot handle special characters in names. Therefore, quoted identifiers should not be used.



Some platforms allow delimiting symbols other than double quotes. For example, SQL Server uses brackets ([]) to designate quoted identifiers.

There are several benefits to following these naming conventions. First, your SQL code becomes, in a sense, self-documenting, because the chosen names are meaningful and understandable to other users. Second, your SQL code and database objects are easier to maintain—especially for other users who come later—because the objects are consistently named. Finally, maintaining consistency increases database functionality. If the database ever has to be transferred or migrated to another RDBMS, consistent and descriptive naming saves both time and energy. Giving a few minutes of thought to naming SQL objects in the beginning can prevent problems later.

Identifier rules

Identifier rules are rules for identifying objects within the database that are rigidly enforced by the database platforms. These rules apply to normal identifiers, not quoted identifiers. Rules specified by the SQL2003 standard generally differ somewhat from those of specific database vendors. Table 2-1 contrasts the SQL2003 rules with those of the RDBMS platforms covered in this book.

Table 2-1. Platform-specific rules for regular object identifiers (excludes quoted identifiers)

Characteristic	Platform	Specification
Identifier size		
	SQL3	128 characters.
	MySQL	64 characters; aliases may be 255 characters.
	Oracle	30 bytes (number of characters depends on the character set); database names are limited to 8 bytes; database links are limited to 128 bytes.
	PostgreSQL	63 characters (<i>NAMEDATALEN</i> property minus 1).
	SQL Server	128 characters; temp tables are limited to 116 characters.

Table 2-1. Platform-specific rules for regular object identifiers (excludes quoted identifiers) (continued)

Characteristic	Platform	Specification
Identifier may contain		
Identifier must begin with	SQL3	Any number or character, and the underscore (_) symbol.
	MySQL	Any number, character, or symbol. Cannot be composed entirely of numbers.
	Oracle	Any number or character, and the underscore (_), pound sign (#), and dollar sign (\$) symbols (though the last two are discouraged). Database links may also contain a period (.).
	PostgreSQL	Any number or character, and the underscore (_) symbol.
	SQL Server	Any number or character, and the underscore (_), at sign (@), pound sign (#), and dollar sign (\$) symbols.
Identifier cannot contain	SQL3	A letter.
	MySQL	A letter or number. Cannot be composed entirely of numbers.
	Oracle	A letter.
	PostgreSQL	A letter or underscore (_).
	SQL Server	A letter, underscore (_), at sign (@), or pound sign (#).
Allows quoted identifiers	SQL3	Spaces or special characters.
	MySQL	Period (.), slash (/), or any ASCII(0) or ASCII(255) character. Single quotes (') and double quotes (") are allowed only in quoted identifiers. Identifiers should not end with space characters.
	Oracle	Spaces, double quotes (""), or special characters.
	PostgreSQL	Double quotes (").
	SQL Server	Spaces or special characters.
Quoted identifier symbol	SQL3	Yes.
	MySQL	Yes.
	Oracle	Yes.
	PostgreSQL	Yes.
	SQL Server	Yes.
Identifier may be reserved	SQL3	Double quotes (").
	MySQL	Single quotes (') or double quotes (") in ANSI compatibility mode.
	Oracle	Double-quotes (").
	PostgreSQL	Double-quotes (").
	SQL Server	Double quotes (") or brackets ([/]); brackets are preferred.
	SQL3	No, unless as a quoted identifier.
	MySQL	No, unless as a quoted identifier.
	Oracle	No, unless as a quoted identifier.
	PostgreSQL	No, unless as a quoted identifier.
	SQL Server	No, unless as a quoted identifier.

Table 2-1. Platform-specific rules for regular object identifiers (excludes quoted identifiers) (continued)

Characteristic	Platform	Specification
Schema addressing		
	SQL3	<i>Catalog.schema.object</i>
	MySQL	<i>Database.object</i> .
	Oracle	<i>Schema.object</i> .
	PostgreSQL	<i>Database.schema.object</i> .
	SQL Server	<i>Server.database.schema.object</i> .
Identifier must be unique		
	SQL3	Yes.
	MySQL	Yes.
	Oracle	Yes.
	PostgreSQL	Yes.
	SQL Server	Yes.
Case Sensitivity		
	SQL3	No.
	MySQL	Only if underlying filesystem is case sensitive (e.g., Mac OS or Unix). Triggers, logfile groups, and tablespaces are always case sensitive.
	Oracle	No by default, but can be changed.
	PostgreSQL	No.
	SQL Server	No by default, but can be changed.
Other rules		
	SQL3	None.
	MySQL	May not contain numbers only.
	Oracle	Database links are limited to 128 bytes and may not be quoted identifiers.
	PostgreSQL	None.
	SQL Server	Microsoft commonly uses brackets rather than double quotes for quoted identifiers.

Identifiers must be unique within their scope. Thus, given our earlier discussion of the hierarchy of database objects, database names must be unique on a particular instance of a database server, while the names of tables, views, functions, triggers, and stored procedures must be unique within a particular schema. On the other hand, a table and a stored procedure *can* have the same name, since they are different types of object. The names of columns, keys, and indexes must be unique on a single table or view, and so forth. Check your database platform's documentation for more information—some platforms require unique identifiers where others may not. For example, Oracle requires that all index identifiers be unique throughout the database, while others (such as SQL Server) require that the index identifier be unique only for the table on which it depends.

Remember, quoted identifiers (object names encapsulated within a special delimiter, usually double quotes) may be used to break some of the identifier rules

specified earlier. One example is that quoted identifiers are case sensitive—that is, “*foo*” does not equal “*FOO*”. Furthermore, quoted identifiers may be used to bestow a reserved word as a name, or to allow normally unusable characters and symbols within a name. For instance, you normally can’t use the percent sign (%) in a table name. However, you can, if you must, use that symbol in a table name so long as you always enclose that table name within double quotes. That is, to name a table **expense%ratios**, you would specify the name in quotes: “*expense%ratios*”. Again, remember that in SQL3 such names are sometimes known as “delimited identifiers.”



Once you have created an object name as a quoted identifier, we recommend that users always reference it using its special delimiter.

Literals

SQL defines a literal value as any explicit numeric value, character string, temporal value (e.g., date or time), or Boolean value that is not an identifier or a keyword. SQL databases allow a variety of literal values in a SQL program. Literal values are allowed for most of the numeric, character, Boolean, and date datatypes. For example, SQL Server numeric datatypes include (among others) *INTEGER*, *REAL*, and *MONEY*. Thus, *numeric literals* can look like:

```
30
-117
+883.3338
-6.66
$70000
2E5
7E-3
```

As these examples illustrate, SQL Server allows signed and unsigned numerals, in scientific or normal notation. And since SQL Server has a money datatype, even a dollar sign can be included. SQL Server does *not* allow other symbols in numeric literals (besides 0 1 2 3 4 5 6 7 8 9 + - \$. E e), however, so do not include commas (or periods, in European countries where a comma is used in place of a period in decimal or monetary values). Most databases interpret a comma in a numeric literal as a *list item separator*. Thus, the literal value 3,000 would likely be interpreted as two values: 3 and, separately, 000.

Boolean, character string, and date literals look like:

```
TRUE
'Hello world!'
'0CT-28-1966 22:14:30:00'
```

Character string literals should always be enclosed in single quotation marks (''). This is the standard delimiter for all character string literals. Character string literals are not restricted just to the letters of the alphabet. In fact, any character in the character set can be represented as a string literal. All of the following are string literals:

```
'1998'  
'70,000 + 14000'  
'There once was a man from Nantucket,'  
'Oct 28, 1966'
```

and are compatible with the *CHARACTER* datatype. Remember not to confuse the string literal '1998' with the numeric literal 1998. Once string literals are associated with a character datatype, it is poor practice to use them in arithmetic operations without explicitly converting them to a numeric datatype. Some database products will perform automatic conversion of string literals containing numbers when comparing them against any *DATE* or *NUMBER* datatype values, but not all.

By doubling the delimiter, you can effectively represent a single quotation mark in a literal string, if necessary. That is, you can use two quotation marks each time a single quotation mark is part of the value. This example, taken from SQL Server, illustrates the idea:

```
SELECT 'So he said ''Who''s Le Petomaine?'''
```

This statement gives the following result:

```
-----  
So he said 'Who's Le Petomaine?'
```

Operators

An *operator* is a symbol specifying an action to be performed on one or more expressions. Operators are used most often in *DELETE*, *INSERT*, *SELECT*, and *UPDATE* statements, but they are also used frequently in the creation of database objects such as stored procedures, functions, triggers, and views.

Operators typically fall into these categories:

Arithmetic operators

Supported by all databases

Assignment operators

Supported by all databases

Bitwise operators

Supported by MySQL and SQL Server

Comparison operators

Supported by all databases

Logical operators

Supported by all databases

Unary operators

Supported by MySQL, Oracle, and SQL Server

Arithmetic operators

Arithmetic operators perform mathematical operations on two expressions of any datatype in the numeric datatype category. See Table 2-2 for a listing of the arithmetic operators.

Table 2-2. Arithmetic operators

Arithmetic operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modula (SQL Server only); returns the remainder of a division operation as an integer value



In MySQL, Oracle, and SQL Server, the + and - operators can be used to perform arithmetic operations on date values.

The various platforms also offer their own unique methods for performing arithmetic operations on date values.

Assignment operators

Except in Oracle, which uses :=, the *assignment operator* (=) assigns a value to a variable or the alias of a column heading. In all of the database platforms covered in this text, the keyword AS may serve as an operator for assigning table- or column-heading aliases.

Bitwise operators

Both Microsoft SQL Server and MySQL provide *bitwise operators* (see Table 2-3) as a shortcut to perform bit manipulations between two-integer expressions. Valid datatypes that are accessible to bitwise operators include *BINARY*, *BIT*, *INT*, *SMALLINT*, *TINYINT*, and *VARBINARY*. PostgreSQL supports the *BIT* and *BIT VARYING* datatypes; it also supports the bitwise operators *AND*, *OR*, *XOR*, concatenation, *NOT*, and shifts left and right.

Table 2-3. Bitwise operators

Bitwise operator	Meaning
&	Bitwise <i>AND</i> (two operands)
	Bitwise <i>OR</i> (two operands)
^	Bitwise exclusive <i>OR</i> (two operands)

Comparison operators

Comparison operators test whether two expressions are equal or unequal. The result of a comparison operation is a Boolean value: *TRUE*, *FALSE*, or *UNKNOWN*.

Also, note that the ANSI standard behavior for a comparison operation where one or more of the expressions is `NULL` is to return `NULL`. For example, the expression `23 + NULL` returns `NULL`, as does the expression `Feb 23, 2002 + NULL`. See Table 2-4 for a list of the comparison operators.

Table 2-4. Comparison operators

Comparison operator	Meaning
<code>=</code>	Equal to
<code>></code>	Greater than
<code><</code>	Less than
<code>>=</code>	Greater than or equal to
<code><=</code>	Less than or equal to
<code><></code>	Not equal to
<code>!=</code>	Not equal to (not ANSI standard)
<code>!<</code>	Not less than (not ANSI standard)
<code>!></code>	Not greater than (not ANSI standard)

Boolean comparison operators are used most frequently in a `WHERE` clause to filter the rows that qualify for the search conditions. The following example uses the greater than or equal to comparison operation:

```
SELECT *
  FROM Products
 WHERE ProductID >= 347
```

Logical operators

Logical operators are commonly used in a `WHERE` clause to test for the truth of some condition. They return a Boolean value of either `TRUE` or `FALSE`. Table 2-5 shows a list of logical operators. Note that not all database systems support all operators.

Table 2-5. Logical operators

Logical operator	Meaning
<code>ALL</code>	<code>TRUE</code> if all of a set of comparisons are <code>TRUE</code>
<code>AND</code>	<code>TRUE</code> if both Boolean expressions are <code>TRUE</code>
<code>ANY</code>	<code>TRUE</code> if any one of a set of comparisons is <code>TRUE</code>
<code>BETWEEN</code>	<code>TRUE</code> if the operand is within a range
<code>EXISTS</code>	<code>TRUE</code> if a subquery contains any rows
<code>IN</code>	<code>TRUE</code> if the operand is equal to one of a list of expressions or one or more rows returned by a subquery
<code>LIKE</code>	<code>TRUE</code> if the operand matches a pattern
<code>NOT</code>	Reverses the value of any other Boolean operator
<code>OR</code>	<code>TRUE</code> if either Boolean expression is <code>TRUE</code>
<code>SOME</code>	<code>TRUE</code> if some of a set of comparisons are <code>TRUE</code>

Unary operators

Unary operators perform an operation on only one expression of any of the datatypes in the numeric datatype category. Unary operators may be used on any numeric datatype, though the bitwise operator (~) may be used only on integer datatypes (see Table 2-6).

Table 2-6. Unary operators

Unary operator	Meaning
+	Numeric value is positive
-	Numeric value is negative
~	A bitwise <i>NOT</i> ; returns the complement of the number (not in Oracle)

Operator precedence

Sometimes operator expressions become rather complex. When an expression has multiple operators, *operator precedence* determines the sequence in which the operations are performed. The order of execution can significantly affect the resulting value.

Operators have different precedence levels. An operator on a higher level is evaluated before an operator on a lower level. The following listing shows the operators' precedence levels, from highest to lowest:

- () (parenthetical expressions)
- +, -, ~ (unary operators)
- *, /, % (mathematical operators)
- +, - (arithmetic operators)
- =, >, <, >=, <=, <>, !=, !>, !< (comparison operators)
- ^ (bitwise exclusive OR), & (bitwise AND), | (bitwise OR)
- NOT
- AND
- ALL, ANY, BETWEEN, IN, LIKE, OR, SOME
- = (variable assignment)

Operators are evaluated from left to right when they are of equal precedence. However, parentheses are used to override the default precedence of the operators in an expression. Expressions within a set of parentheses are evaluated first, while operations outside the parentheses are evaluated next.

For example, the following expressions in an Oracle query return very different results:

```
SELECT 2 * 4 + 5 FROM dual
-- Evaluates to 8 + 5, which yields an expression result of 13.
```

```
SELECT 2 * (4 + 5) FROM dual
-- Evaluates to 2 * 9, which yields an expression result of 18.
```

In expressions with nested parentheses, the most deeply nested expression is evaluated first.

This next example contains nested parentheses, with the expression $5 - 3$ in the most deeply nested set of parentheses. This expression yields a value of 2. Then the addition operator (+) adds this result to 4, which yields a value of 6. Finally, the 6 is multiplied by 2 to yield an expression result of 12:

```
SELECT 2 * (4 + (5 - 3)) FROM dual
-- Evaluates to 2 * (4 + 2), which further evaluates to 2 * 6,
-- and yields an expression result of 12.
RETURN
```



We recommend using parentheses to clarify precedence in all complex queries.

System delimiters and operators

String delimiters mark the boundaries of a string of alphanumeric characters. *System delimiters* are those symbols within the character set that have special significance to your database server. *Delimiters* are symbols that are used to judge the order or hierarchy of processes and list items. *Operators* are those delimiters used to judge values in comparison operations, including symbols commonly used for arithmetic or mathematical operations. Table 2-7 lists the system delimiters and operators allowed by SQL.

Table 2-7. SQL delimiters and operators

Symbol	Usage	Example
+	Addition operator; in SQL Server, also serves as a concatenation operator	On all database platforms: SELECT MAX(emp_id) + 1 FROM employee
-	Subtraction operator; also serves as a range indicator in <i>CHECK</i> constraints	As a subtraction operator: SELECT MIN(emp_id) - 1 FROM employee As a range operator, in a <i>CHECK</i> constraint: ALTER TABLE authors ADD CONSTRAINT authors_zip_num CHECK (zip LIKE '%[0-9]%')
*	Multiplication operator	SELECT salary * 0.05 AS 'bonus' FROM employee;
/	Division operator	SELECT salary / 12 AS 'monthly' FROM employee;
=	Equality operator	SELECT * FROM employee WHERE lname = 'Fudd'
<, >	Inequality operators (\neq is a nonstandard equivalent on several platforms)	On all platforms: SELECT * FROM employee WHERE lname <> 'Fudd'

Table 2-7. SQL delimiters and operators (continued)

Symbol	Usage	Example
<	Less than operator	<pre>SELECT lname, emp_id, (salary * 0.05) AS bonus FROM employee WHERE (salary * 0.05) <= 10000 AND exempt_status < 3</pre>
<=	Less than or equal to operator	
>	Greater than operator	<pre>SELECT lname, emp_id, (salary * 0.025) AS bonus FROM employee WHERE (salary * 0.025) > 10000 AND exempt_status >= 4</pre>
>=	Greater than or equal to operator	
()	Used in expressions and function calls, to specify order of operations, and as a subquery delimiter	<p>Expression:</p> <pre>SELECT (salary / 12) AS monthly FROM employee WHERE exempt_status >= 4</pre> <p>Function call:</p> <pre>SELECT SUM(travel_expenses) FROM "expense%ratios"</pre> <p>Order of operations:</p> <pre>SELECT (salary / 12) AS monthly, ((salary / 12) / 2) AS biweekly FROM employee WHERE exempt_status >= 4</pre> <p>Subquery:</p> <pre>SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > '01-JAN-2004')</pre>
%	Wildcard attribute indicator	<pre>SELECT * FROM employee WHERE lname LIKE 'Fud%'</pre>
,	List item separator	<pre>SELECT lname, fname, ssn, hire_date FROM employee WHERE lname = 'Fudd'</pre>
.	Identifier qualifier separator	<pre>SELECT * FROM scott.employee WHERE lname LIKE 'Fud%'</pre>
'	Character string indicators	<pre>SELECT * FROM employee WHERE lname LIKE 'FUD%' OR fname = 'ELMER'</pre>
"	Quoted identifier indicators	<pre>SELECT expense_date, SUM(travel_expense) FROM "expense%ratios" WHERE expense_date BETWEEN '01-JAN-2004' AND '01-APR-2004'</pre>

Table 2-7. SQL delimiters and operators (continued)

Symbol	Usage	Example
--	Single-line comment delimiter (two dashes followed by a space)	-- Finds all employees like Fudd, Fudge, and Fudston SELECT * FROM employee WHERE lname LIKE 'Fud%'
/*	Beginning multiline comment delimiter	/* Finds all employees like Fudd, Fudge, and Fudston */
*/	Ending multiline comment indicator	SELECT * FROM employee WHERE lname LIKE 'Fud%'

Keywords and Reserved Words

Just as certain symbols have special meaning and functionality within SQL, certain words and phrases have special significance. SQL *keywords* are words whose meanings are so closely tied to the operation of the RDBMS that they should not be used for any other purpose; generally, they are words used in SQL statements. *Reserved words*, on the other hand, do not have special significance now, but they probably will in a future release. Note that these words *can* be used as identifiers on most platforms, but they shouldn't be. For example, the word “SELECT” is a keyword and should not be used as a table name. To emphasize the fact that keywords should not be used as identifiers but nevertheless could be, the SQL standard calls them “nonreserved keywords.”



It is generally a good idea to avoid naming columns or tables after a keyword that occurs in *any* major platform, because database applications are frequently converted from one platform to another.

Reserved words and keywords are not always words used in SQL statements; they may also be words commonly associated with database technology. For example, *CASCADE* is used to describe data manipulations that allow their actions, such as a delete or update operation, to “flow down,” or cascade, to any subordinate tables. Reserved words and keywords are widely published so that programmers will not use them as identifiers that will, either now or at some later revision, cause a problem.

SQL3 specifies its own list of reserved words and keywords, as do the database platforms, because they each have their own extensions to the SQL command set. The SQL standard keywords, as well as the keywords in the different vendor implementations, are listed in the Appendix.

SQL2003 and Platform-Specific Datatypes

A table can contain one or many columns. Each column must be defined with a datatype that provides a general classification of the data that the column will store. In real-world applications, datatypes improve efficiency and provide some control over how tables are defined and how the data is stored within a table.

Using specific datatypes enables better, more understandable queries and helps control the integrity of the data.

The tricky thing about SQL2003 datatypes is that they do not always map directly to identical implementations in different platforms. Although the various platforms specify “datatypes” that correspond to the SQL2003 datatypes, these are not always true SQL2003 datatypes: for example, MySQL’s implementation of a *BIT* datatype is actually identical to a *CHAR(1)* datatype value. Nonetheless, each of the platform-specific datatypes is close enough to the standard to be both easily understandable and job-ready.

The official SQL2003 datatypes (as opposed to platform-specific datatypes) fall into the general categories described in Table 2-8. Note that the SQL2003 standard contains a few rarely used datatypes (*ARRAY*, *MULTISET*, *REF*, and *ROW*) that are shown only in Table 2-8 and not discussed elsewhere in the book.

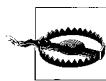
Table 2-8. SQL2003 categories and datatypes

Category	Example datatypes and abbreviations	Description
<i>BINARY</i>	<i>BINARY</i> <i>LARGE OBJECT (BLOB)</i>	This datatype stores binary string values in hexadecimal format. Binary string values are stored without reference to any character set and without any length limit.
<i>BOOLEAN</i>	<i>BOOLEAN</i>	This datatype stores truth values (either <i>TRUE</i> or <i>FALSE</i>).
<i>CHARACTER</i> string types	<i>CHAR</i> <i>CHARACTER VARYING (VARCHAR)</i>	These datatypes can store any combination of characters from the applicable character set. The varying datatypes allow variable lengths, while the other datatypes allow only fixed lengths. Also, the variable-length datatypes automatically trim trailing spaces, while the other datatypes pad all open space.
	<i>NATIONAL CHARACTER (NCHAR)</i> <i>NATIONAL CHARACTER VARYING (NCHAR VARYING)</i>	The national character datatypes are designed to support a particular implementation-defined character set.
	<i>CHARACTER LARGE OBJECT (CLOB)</i>	<i>CHARACTER LARGE OBJECT</i> and <i>BINARY LARGE OBJECT</i> are collectively referred to as <i>large object string types</i> .
	<i>NATIONAL CHARACTER LARGE OBJECT (NCLOB)</i>	Same as <i>CHARACTER LARGE OBJECT</i> , but supports a particular implementation-defined character set.
<i>DATALINK</i>	<i>DATALINK</i>	Defines a reference to a file or other external data source that is not part of the SQL environment.
<i>INTERVAL</i>	<i>INTERVAL</i>	Specifies a set of time values or span of time.
<i>COLLECTION</i>	<i>ARRAY</i> <i>MULTISET</i>	<i>ARRAY</i> was offered in SQL99, and <i>MULTISET</i> was added in SQL2003. Whereas an <i>ARRAY</i> is a set-length, ordered collection of elements, <i>MULTISET</i> is a variable-length, unordered collection of elements. The elements in an <i>ARRAY</i> and a <i>MULTISET</i> must be of a predefined datatype.

Table 2-8. SQL2003 categories and datatypes (continued)

Category	Example datatypes and abbreviations	Description
NUMERIC	<i>INTEGER (INT)</i> <i>SMALLINT</i> <i>BIGINT</i> <i>NUMERIC(p,s)</i> <i>DECIMAL(p,s)</i> <i>FLOAT(p,s)</i> <i>REAL</i> <i>DOUBLE PRECISION</i>	These datatypes store exact numeric values (integers or decimals) or approximate (floating-point) values. <i>INT</i> , <i>BIGINT</i> , and <i>SMALLINT</i> store exact numeric values with a predefined precision and a scale of zero. <i>NUMERIC</i> and <i>DEC</i> store exact numeric values with a definable precision and a definable scale. <i>FLOAT</i> stores approximate numeric values with a definable precision, while <i>REAL</i> and <i>DOUBLE PRECISION</i> have predefined precisions. You may define a precision (<i>p</i>) and scale (<i>s</i>) for a <i>DECIMAL</i> , <i>FLOAT</i> , or <i>NUMERIC</i> datatype to indicate the total number of allowed digits and the number of decimal places, respectively. <i>INT</i> , <i>SMALLINT</i> , and <i>DEC</i> are sometimes referred to as <i>exact numeric types</i> , while <i>FLOAT</i> , <i>REAL</i> , and <i>DOUBLE PRECISION</i> are sometimes called <i>approximate numeric types</i> .
TEMPORAL	<i>DATE</i> <i>TIME</i> <i>TIME WITH TIME ZONE</i> <i>TIMESTAMP</i> <i>TIMESTAMP WITH TIME ZONE</i>	These datatypes handle values related to time. <i>DATE</i> and <i>TIME</i> are self-explanatory. Datatypes with the <i>WITH TIME ZONE</i> suffix also include a time zone offset. The <i>TIMESTAMP</i> datatypes are used to store a value that represents a precise moment in time. Temporal types are also known as <i>datetime types</i> .
XML	<i>XML</i>	Stores XML data and can be used wherever a SQL datatype is allowed (e.g., for a column of a table, a field in a row, etc.). Operations on the values of an XML type assume a tree-based internal data structure. The internal data structure is based on the XML Information Set Recommendation (InfoSet), using a new document information item called the <i>XML root information item</i> .

Not every database platform supports every ANSI SQL datatype. Table 2-9 compares datatypes across the five platforms. The table is organized by datatype name. Be careful to look for footnotes when reading this table, because some platforms support a datatype of a given name but implement it in a different way than the ANSI standard and/or other vendors.



While the different platforms may support similarly named datatypes, the details of their implementations may vary. The sections that follow this table list the specific requirements of each platform's datatypes.

Table 2-9. Comparison of platform-specific datatypes

Vendor datatype	MySQL	Oracle	PostgreSQL	SQL Server	SQL2003 datatype
<i>BFILE</i>		Y			None
<i>BIGINT</i>	Y		Y	Y	<i>BIGINT</i>
<i>BINARY</i>	Y			Y	<i>BLOB</i>
<i>BINARY_FLOAT</i>		Y			<i>FLOAT</i>
<i>BINARY_DOUBLE</i>		Y			<i>DOUBLE PRECISION</i>
<i>BIT</i>	Y		Y	Y	None
<i>BIT VARYING, VARBIT</i>			Y		None
<i>BLOB</i>	Y	Y			<i>BLOB</i>
<i>BOOL, BOOLEAN</i>	Y		Y		<i>BOOLEAN</i>
<i>BOX</i>			Y		None
<i>BYTEA</i>			Y		<i>BLOB</i>
<i>CHAR, CHARACTER</i>	Y	Y	Y	Y	<i>CHARACTER</i>
<i>CHAR FOR BIT DATA</i>					None
<i>CIDR</i>			Y		None
<i>CIRCLE</i>			Y		None
<i>CLOB</i>		Y			<i>CLOB</i>
<i>CURSOR</i>				Y	None
<i>DATALINK</i>					<i>DATALINK</i>
<i>DATE</i>	Y	Y	Y	Y	<i>DATE</i>
<i>DATETIME</i>	Y			Y	<i>TIMESTAMP</i>
<i>DATETIMEOFFSET</i>				Y	<i>TIMESTAMP</i>
<i>DATETIME2</i>				Y	<i>TIMESTAMP WITH TIME ZONE</i>
<i>DBCLOB</i>					<i>NCLOB</i>
<i>DEC, DECIMAL</i>	Y	Y	Y	Y	<i>DECIMAL</i>
<i>DOUBLE, DOUBLE PRECISION</i>	Y	Y	Y ^a	Y	<i>FLOAT</i>
<i>ENUM</i>	Y		Y		None
<i>FLOAT</i>	Y	Y	Y	Y	<i>DOUBLE PRECISION</i>
<i>FLOAT4</i>			Y ^b		<i>FLOAT(p)</i>
<i>FLOAT8</i>			Y ^c		<i>FLOAT(p)</i>
<i>GRAPHIC</i>					<i>BLOB</i>
<i>GEOGRAPHY</i>				Y	None
<i>GEOMETRY</i>				Y	None
<i>HIERARCHYID</i>				Y	None
<i>IMAGE</i>				Y	None
<i>INET</i>			Y		None
<i>INT, INTEGER</i>	Y	Y	Y	Y	<i>INTEGER</i>
<i>INT2</i>			Y		<i>SMALLINT</i>
<i>INT4</i>			Y		<i>INT, INTEGER</i>
<i>INTERVAL</i>			Y		<i>INTERVAL</i>

Table 2-9. Comparison of platform-specific datatypes (continued)

Vendor datatype	MySQL	Oracle	PostgreSQL	SQL Server	SQL2003 datatype
<i>INTERVAL DAY TO SECOND</i>		Y	Y		<i>INTERVAL DAY TO SECOND</i>
<i>INTERVAL YEAR TO MONTH</i>		Y	Y		<i>INTERVAL YEAR TO MONTH</i>
<i>LINE</i>				Y	None
<i>LONG</i>		Y			None
<i>LONG VARCHAR</i>					None
<i>LONGBLOB</i>	Y				<i>BLOB</i>
<i>LONG RAW</i>		Y			<i>BLOB</i>
<i>LONG VARGRAPHIC</i>					None
<i>LONGTEXT</i>	Y				None
<i>LSEG</i>				Y	None
<i>MACADDR</i>				Y	None
<i>MEDIUMBLOB</i>	Y				None
<i>MEDIUMINT</i>	Y				<i>INT</i>
<i>MEDIUMTEXT</i>	Y				None
<i>MONEY</i>				Y ^d	None
<i>NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, NCHAR VARYING, NVARCHAR</i>	Y	Y	Y	Y	<i>NATIONAL CHARACTER VARYING</i>
<i>NCHAR, NATIONAL CHAR, NATIONAL CHARACTER</i>	Y	Y	Y	Y	<i>NATIONAL CHARACTER</i>
<i>NCLOB</i>		Y			<i>NCLOB</i>
<i>NTEXT, NATIONAL TEXT</i>				Y	<i>NCLOB</i>
<i>NVARCHAR2(n)</i>		Y			None
<i>NUMBER</i>	Y	Y	Y ^e	Y	None
<i>NUMERIC</i>				Y	<i>NUMERIC</i>
<i>OID</i>				Y	None
<i>PATH</i>				Y	None
<i>POINT</i>				Y	None
<i>POLYGON</i>				Y	None
<i>RAW</i>		Y			None
<i>REAL</i>	Y	Y	Y	Y	<i>REAL</i>
<i>ROWID</i>		Y			None
<i>ROWVERSION</i>				Y	None
<i>SERIAL, SERIAL4</i>	Y ^f		Y		None
<i>SERIAL8, BIGSERIAL</i>			Y		None
<i>SET</i>	Y				None
<i>SMALLDATETIME</i>				Y	None

Table 2-9. Comparison of platform-specific datatypes (continued)

Vendor datatype	MySQL	Oracle	PostgreSQL	SQL Server	SQL2003 datatype
<i>SMALLINT</i>	Y	Y	Y	Y	<i>SMALLINT</i>
<i>SMALLMONEY</i>				Y	<i>None</i>
<i>SQL_VARIANT</i>				Y	<i>None</i>
<i>TABLE</i>				Y	<i>None</i>
<i>TEXT</i>	Y		Y	Y	<i>None</i>
<i>TIME</i>	Y		Y	Y	<i>TIME</i>
<i>TIMESPAN</i>					<i>INTERVAL</i>
<i>TIMESTAMP</i>	Y	Y	Y	Y ^g	<i>TIMESTAMP</i>
<i>TIMESTAMP WITH TIMEZONE, TIMESTAMPTZ</i>			Y		<i>TIMESTAMP WITH TIMEZONE</i>
<i>TIMETZ</i>			Y		<i>TIME WITH TIMEZONE</i>
<i>TINYBLOB</i>	Y			Y	<i>None</i>
<i>TINYINT</i>	Y			Y	<i>None</i>
<i>TINYTEXT</i>	Y				<i>None</i>
<i>UNIQUEIDENTIFIER</i>				Y	<i>None</i>
<i>UROWID</i>		Y			<i>None</i>
<i>VARBINARY</i>	Y			Y	<i>BLOB</i>
<i>VARCHAR, CHAR VARYING, CHARACTER VARYING</i>	Y	Y ^h	Y	Y	<i>CHARACTER VARYING(n)</i>
<i>VARCHAR2</i>		Y			<i>CHARACTER VARYING</i>
<i>VARCHAR FOR BIT DATA</i>					<i>BIT VARYING</i>
<i>VARGRAPHIC</i>					<i>NCHAR VARYING</i>
<i>YEAR</i>	Y				<i>TINYINT</i>
<i>XML</i>			Y	Y	<i>XML</i>
<i>XMLTYPE</i>		Y			<i>XML</i>

^a Synonym for *FLOAT*.

^b Synonym for *REAL*.

^c Synonym for *DOUBLE PRECISION*.

^d Synonym for *DECIMAL(9,2)*.

^e Synonym for *DECIMAL*.

^f Synonym for *BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE*.

^g Implemented as a non-date datatype.

^h Oracle vastly prefers *VARCHAR2*.

The following sections list platform-specific datatypes, their SQL2003 datatype categories (if any), and pertinent details. Descriptions are provided for non-SQL2003 datatypes.

MySQL Datatypes

MySQL version 5.1 has support for spatial data, but not as datatypes. Spatial data is handled in a variety of classes provided in the OpenGIS Geometry Model, which is supported by the MyISAM, InnoDB, NDB, and ARCHIVE database engines. Only MyISAM supports both spatial and non-spatial indexes; the other database engines only support non-spatial indexes.

MySQL numeric datatypes support the following optional attributes:

UNSIGNED

The numeric value is assumed to be non-negative (positive or zero). For fixed-point datatypes such as *DECIMAL* and *NUMERIC*, the space normally used to show a positive or negative condition of the numeric value can be used as part of the value, providing a little extra numeric range in the column for these types. (There is no *SIGNED* optional attribute.)

ZEROFILL

Used for display formatting, this attribute tells MySQL that the numeric value is padded to its full size with zeros rather than spaces. *ZEROFILL* automatically forces the *UNSIGNED* attribute as well.

MySQL also enforces a maximum display size for columns of up to 255 characters. Columns longer than 255 characters are stored properly, but only 255 characters are displayed. Floating-point numeric datatypes may have a maximum of 30 digits after the decimal point.

The following list enumerates the datatypes MySQL supports. These include most of the SQL2003 datatypes, plus several additional datatypes used to contain lists of values, as well as datatypes used for binary large objects (*BLOBS*). Datatypes that extend the ANSI standard include *TEXT*, *ENUM*, *SET*, and *MEDIUMINT*. Special datatype attributes that go beyond the ANSI standard include *AUTO_INCREMENT*, *BINARY*, *NULL*, *UNSIGNED*, and *ZEROFILL*. The datatypes supported by MySQL are:

BIGINT[(n)] [UNSIGNED] [ZEROFILL] (SQL2003 datatype: BIGINT)

Stores signed or unsigned integers. The signed range is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The unsigned range is 0 to 18,446,744,073,709,551,615. *BIGINT* may perform imprecise calculations with very large numbers (63 bits), due to rounding issues.

BINARY[(n)] (SQL2003 datatype: BLOB)

Stores binary byte strings of optional length *n*. Otherwise, similar to the *CHAR* datatype.

BIT, BOOL (SQL2003 datatype: none)

Synonyms for *TINYINT*.

BLOB (SQL2003 datatype: BLOB)

Stores up to 65,535 characters of data. Support for indexing *BLOB* columns is found only in MySQL version 3.23.2 or greater (this feature is not found in any other platform covered in this book). In MySQL, *BLOBS* are functionally equivalent to the MySQL datatype *VARCHAR BINARY* (discussed later) with the default upper limit. *BLOBS* always require case-sensitive comparisons.

BLOB columns differ from MySQL *VARCHAR* *BINARY* columns by not allowing *DEFAULT* values. You cannot perform a *GROUP BY* or *ORDER BY* on *BLOB* columns. Depending on the storage engine being used, *BLOBs* also are sometimes stored separately from their tables, whereas all other datatypes in MySQL (with the exception of *TEXT*) are stored in the table file structure itself.

CHAR(n) [BINARY], CHARACTER(n) [BINARY] (SQL2003 datatype: CHARACTER(n))

Contains a fixed-length character string of 1 to 255 characters. *CHAR* pads with blank spaces when it stores values but trims spaces upon retrieval, just as ANSI SQL2003 *VARCHAR* does. The *BINARY* option allows binary searches rather than dictionary-order, case-insensitive searches.

DATE (SQL2003 datatype: DATE)

Stores a date within the range of 1000-01-01 to 9999-12-31 (delimited by quotes). MySQL displays these values by default in the format YYYY-MM-DD, though the user may specify some other display format.

DATETIME (SQL2003 datatype: TIMESTAMP)

Stores date and time values within the range of 1000-01-01 00:00:00 to 9999-12-31 23:59:59.

DECIMAL[(p,s)] [ZEROFILL] (SQL2003 datatype: DECIMAL(PRECISION, SCALE))

Stores exact numeric values as if they were strings, using a single character for each digit. Precision is 10 if omitted, and scale is 0 if omitted.

DOUBLE[(p,s)] [ZEROFILL], DOUBLE PRECISION[(p,s)] [ZEROFILL] (SQL2003 datatype: DOUBLE PRECISION)

Holds double-precision numeric values and is otherwise identical to the double-precision *FLOAT* datatype, except for the fact that its allowable range is $-1.7976931348623157\text{E}+308$ to $-2.2250738585072014\text{E}-308$, 0, and $2.2250738585072014\text{E}-308$ to $1.7976931348623157\text{E}+308$.

ENUM(“val1,” “val2,” …n) [CHARACTER SET cs_name] [COLLATE collation_name] (SQL2003 datatype: none)

Holds a list of allowable values (expressed as strings but stored as integers). Other possible values for the datatype are *NULL*, or an empty string (“”) as an error value. Up to 65,535 distinct values are allowed.

FLOAT[(p,s)] [ZEROFILL] (SQL2003 datatype: FLOAT(P))

Stores floating-point numbers in the range $-3.402823466\text{E}+38$ to $-1.175494351\text{E}-38$ and $1.175494351\text{E}-38$ to $3.402823466\text{E}+38$. *FLOAT* without a precision, or with a precision of ≤ 24 , is single-precision. Otherwise, *FLOAT* is double-precision. When specified alone, the precision can range from 0 to 53. When you specify both precision and scale, the precision may be as high as 255 and the scale may be as high as 253. All *FLOAT* calculations in MySQL are done with double precision and may, since *FLOAT* is an approximate datatype, encounter rounding errors.

INT[TEGER][(n)] [UNSIGNED] [ZEROFILL] [AUTO_INCREMENT] (SQL2003 datatype: INTEGER)

Stores signed or unsigned integers. For ISAM tables, the signed range is from -2,147,483,648 to 2,147,483,647 and the unsigned range is from 0 to 4,294,967,295. The range of values varies slightly for other types of tables. *AUTO_INCREMENT* is available to all of the *INT* variants; it creates a unique row identity for all new rows added to the table. (Refer to the section “CREATE/ALTER DATABASE Statement” in Chapter 3 for more information on *AUTO_INCREMENT*.)

LONGBLOB (SQL2003 datatype: BLOB)

Stores *BLOB* data up to 4,294,967,295 characters in length. Note that this might be too much information for some client/server protocols to support.

LONGTEXT [CHARACTER SET *cs_name*] [COLLATE *collation_name*] (SQL2003 datatype: CLOB)

Stores *TEXT* data up to 4,294,967,295 characters in length (less if the characters are multibyte). Note that this might be too much data for some client/server protocols to support.

MEDIUMBLOB (SQL2003 datatype: none)

Stores *BLOB* data up to 16,777,215 bytes in length. The first three bytes are consumed by a prefix indicating the total number of bytes in the value.

MEDIUMINT[(n)] [UNSIGNED] [ZEROFILL] (SQL2003 datatype: none)

Stores signed or unsigned integers. The signed range is from 8,388,608 to -8,388,608, and the unsigned range is 0 to 16,777,215.

MEDIUMTEXT [CHARACTER SET *cs_name*] [COLLATE *collation_name*] (SQL2003 datatype: none)

Stores *TEXT* data up to 16,777,215 characters in length (less if the characters are multibyte). The first three bytes are consumed by a prefix indicating the total number of bytes in the value.

NCHAR(*n*) [BINARY], [NATIONAL] CHAR(*n*) [BINARY] (SQL2003 datatype: NCHAR(*n*))

Synonyms for *CHAR*. The *NCHAR* datatypes provide UNICODE support beginning in MySQL v4.1.

NUMERIC(*p,s*) (SQL2003 datatype: DECIMAL(*p,s*))

Synonym for *DECIMAL*.

NVARCHAR(*n*) [BINARY], [NATIONAL] VARCHAR(*n*) [BINARY], NATIONAL CHARACTER VARYING(*n*) [BINARY] (SQL2003 datatype: NCHAR VARYING)

Synonyms for *VARYING* [BINARY]. Hold variable-length character strings up to 255 characters in length. Values are stored and compared in a case-insensitive fashion unless the *BINARY* keyword is used.

REAL(*p,s*) (SQL2003 datatype: REAL)

Synonym for *DOUBLE PRECISION*.

SERIAL

Synonym for `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE`.

`SET("val1," "val2," ...n)` [CHARACTER SET *cs_name*] [COLLATE *collation_name*] (SQL2003 datatype: *none*)

A `CHAR` datatype whose value must be equal to zero or more values specified in the list of values. Up to 64 items are allowed in the list of values.

`SMALLINT[(n)]` [UNSIGNED] [ZEROFILL] (SQL2003 datatype: `SMALLINT`)

Stores signed or unsigned integers. The signed range is from -32,768 to 32,767, and the unsigned range is from 0 to 65,535.

`TEXT` (SQL2003 datatype: *none*)

Stores up to 65,535 characters of data. `TEXT` datatypes are sometimes stored separately from their tables, depending on the storage engine used, whereas all other datatypes (with the exception of `BLOB`) are stored in their respective table file structures. `TEXT` is functionally equivalent to `VARCHAR` with no specific upper limit (besides the maximum size of the column), and it requires case-insensitive comparisons. `TEXT` differs from a standard `VARCHAR` column by not allowing `DEFAULT` values. `TEXT` columns cannot be used in `GROUP BY` or `ORDER BY` clauses. In addition, support for indexing `TEXT` columns is provided only in MySQL version 3.23.2 and greater.

`TIME` (SQL2003 datatype: *none*)

Stores time values in the range of '–838:59:59' to '838:59:59', in the format 'HH:MM:SS'. The values may be assigned as strings or numbers.

`TIMESTAMP` (SQL2003 datatype: `TIMESTAMP`)

Stores date values in the range of '1970-01-01 00:00:01' to partway through the year 2038. The values are expressed as the number of seconds since '1970-01-01 00:00:01'. Timestamp values are always displayed in the format 'YYYY-MM-DD HH:MM:SS'.

`TINYBLOB` (SQL2003 datatype: `BLOB`)

Stores `BLOB` values of up to 255 bytes, the first byte being consumed by a prefix indicating the total number of bytes in the value.

`TINYTEXT` (SQL2003 datatype: *none*)

Stores `TEXT` values of up to 255 characters (less if they are multibyte characters). The first byte is consumed by a prefix indicating the total number of bytes in the value.

`VARBINARY(n)` (SQL2003 datatype: `BLOB`)

Stores variable-length binary byte strings of length *n*. Otherwise, similar to the `VARCHAR` datatype.

`YEAR` (SQL2003 datatype: *none*)

Stores the year in a two- or four-digit (the default) format. Two-digit years allow values of '70' to '69', meaning 1970 to 2069, while four-digit years allow values of '1901' to '2155', plus '0000'. `YEAR` values are always displayed in 'YYYY' format but may be assigned as strings or numbers.

Oracle Datatypes

As you'll see in this section, Oracle supports a rich variety of datatypes, including most of the SQL3 datatypes and some special datatypes. The special datatypes, however, often require optional components to be installed. For example, Oracle supports spatial datatypes, but only if you have installed the Oracle Spatial add-on. The Oracle Spatial datatypes, including *SDO_GEOGRAPHY*, *SDO_TOPO_GEOGRAPHY*, and *SDO_GEORASTER*, are beyond the scope of this book. Refer to the Oracle Spatial documentation for further details on these types.

Oracle Multimedia datatypes use object types, similar to Java or C++ classes for multimedia data. Oracle Multimedia datatypes include *ORDAudio*, *ORDImage*, *ORDVideo*, *ORDDoc*, *ORDDicom*, *SI_Stillimage*, *SI_Color*, *SI_AverageColor*, *SI_ColorHistogram*, *SI_PositionalColor*, *SI_Texture*, *SI_FeatureList*, and *ORDImageSignature*.

Oracle also supports "Any Types" datatypes. These highly flexible datatypes are intended for use as procedure parameters and as table columns where the actual type is unknown. The Any Type datatypes are *ANYTYPE*, *ANYDATA*, and *ANYDATASET*.

A complete listing of the Oracle datatypes follows:

BFILE (SQL2003 datatype: DATALINK)

Holds a pointer to a *BLOB* stored outside the database, but present on the local server, of up to 4 GB in size. The database streams input (but not output) access to the external *BLOB*. If you delete a row containing a *BFILE* value, only the pointer value is deleted; the actual file structure is not deleted.

BINARY_DOUBLE (SQL2003 datatype: FLOAT)

Holds a 64-bit floating-point number.

BINARY_FLOAT (SQL2003 datatype: FLOAT)

Holds a 32-bit floating-point number.

BLOB (SQL2003 datatype: BLOB)

Holds a binary large object (*BLOB*) value of between 8 and 128 terabytes in size, depending on the database block size. In Oracle, large binary objects (*BLOBS*, *CLOBS*, and *NCLOBS*) have the following restrictions:

- They cannot be selected remotely.
- They cannot be stored in clusters.
- They cannot compose a varray.
- They cannot be a component of an *ORDER BY* or *GROUP BY* clause in a query.
- They cannot be used by an aggregate function in a query.
- They cannot be referenced in queries using *DISTINCT*, *UNIQUE*, or joins.
- They cannot be referenced in *ANALYZE...COMPUTE* or *ANALYZE...ESTIMATE* statements.
- They cannot be part of a primary key or index key.
- They cannot be used in the *UPDATE OF* clause in an *UPDATE* trigger.

CHAR(n) [BYTE | CHAR], CHARACTER(n) [BYTE | CHAR] (SQL2003 datatype: CHARACTER(n))

Holds fixed-length character data of up to 2,000 bytes in length. *BYTE* tells Oracle to use bytes for the size measurement. *CHAR* tells Oracle to use characters for the size measurement.

CLOB (SQL2003 datatype: CLOB)

Stores a character large object (*CLOB*) value of between 8 and 128 terabytes in size, depending on the database block size. See the description of the *BLOB* datatype for a list of restrictions on the use of the *CLOB* type.

DATE (SQL2003 datatype: DATE)

Stores a valid date and time within the range of 4712BC-01-01 00:00:00 to 9999AD-12-31 23:59:59.

DECIMAL(p,s) (SQL2003 datatype: DECIMAL(p,s))

A synonym for *NUMBER* that accepts precision and scale arguments.

DOUBLE PRECISION (SQL2003 datatype: DOUBLE PRECISION)

Stores floating-point values with double precision, the same as *FLOAT(126)*.

FLOAT(n) (SQL2003 datatype: FLOAT(n))

Stores floating-point numeric values with a binary precision of up to 126.

INTEGER(n) (SQL2003 datatype: INTEGER)

Stores signed and unsigned integer values with a precision of up to 38. *INTEGER* is treated as a synonym for *NUMBER*.

INTERVAL DAY(n) TO SECOND(x) (SQL2003 datatype: INTERVAL)

Stores a time span in days, hours, minutes, and seconds, where *n* is the number of digits in the day field (values from 0 to 9 are acceptable, and 2 is the default) and *x* is the number of digits used for fractional seconds in the seconds field (values from 0 to 9 are acceptable, and 6 is the default).

INTERVAL YEAR(n) TO MONTH (SQL2003 datatype: INTERVAL)

Stores a time span in years and months, where *n* is the number of digits in the year field. The value of *n* can range from 0 to 9, with a default of 2.

LONG (SQL2003 datatype: none)

Stores variable-length character data of up to 2 gigabytes in size. Note, however, that *LONG* is not scheduled for long-term support by Oracle. Use another datatype, such as *CLOB*, instead of *LONG* whenever possible.

LONG RAW (SQL2003 datatype: none)

Stores raw variable-length binary data of up to 2 gigabytes in size. *LONG RAW* and *RAW* are typically used to store graphics, sounds, documents, and other large data structures. *BLOB* is preferred over *LONG RAW* in Oracle, because there are fewer restrictions on its use. *LONG RAW* is deprecated.

NATIONAL CHARACTER VARYING(n), NATIONAL CHAR VARYING(n),

NCHAR VARYING(n) (SQL2003 datatype: NCHAR VARYING (n))

Synonyms for *NVARCHAR2*.

NCHAR(*n*), NATIONAL CHARACTER(*n*), NATIONAL CHAR(*n*) (SQL2003 datatype: NATIONAL CHARACTER)

Holds UNICODE character data of 1 to 2,000 bytes in length. Default size is 1 byte.

NCLOB (SQL2003 datatype: NCLOB)

Represents a CLOB that supports multibyte and UNICODE values of between 8 and 128 terabytes in size, depending on the database block size. See the description of the BLOB datatype for a list of restrictions on the use of the NCLOB type.

NUMBER(*p,s*), NUMERIC(*p,s*) (SQL2003 datatype: NUMERIC(*p,s*))

Stores a number with a precision of 1 to 38 and a scale of -84 to 127.

NVARCHAR2(*n*) (SQL2003 datatype: none)

Represents Oracle's preferred UNICODE variable-length character datatype. Can hold data of 1 to 4,000 bytes in size.

RAW(*n*) (SQL2003 datatype: none)

Stores raw, variable-length binary data of up to 2,000 bytes in size. The value *n* is the specified size of the datatype. RAW is also deprecated in Oracle 11g. (See LONG RAW.)

REAL (SQL2003 datatype: REAL)

Stores floating-point values as single-precision. Same as FLOAT(63).

ROWID (SQL2003 datatype: none)

Represents a unique, base-64 identifier for each row in a table, often used in conjunction with the ROWID pseudocolumn.

SMALLINT (SQL2003 datatype: SMALLINT)

Synonym for INTEGER.

TIMESTAMP(*n*) {[WITH TIME ZONE] | [WITH LOCAL TIME ZONE]}

(SQL2003 datatype: TIMESTAMP[WITH TIME ZONE])

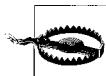
Stores a full date and time value, where *n* is the number of digits (values from 0 to 9 are acceptable, and 6 is the default) in the fractional part of the seconds field. WITH TIME ZONE stores whatever time zone you pass to it (the default is your session time zone) and returns a time value in that same time zone. WITH LOCAL TIME ZONE stores data in the time zone of the current session and returns data in the time zone of the user's session.

UROWID[*(n)*] (SQL2003 datatype: none)

Stores a base-64 value showing the logical address of the row in its table. Defaults to 4,000 bytes in size, but you may optionally specify a size of anywhere up to 4,000 bytes.

VARCHAR(*n*), CHARACTER VARYING(*n*), CHAR VARYING(*n*) (SQL2003 datatype: CHARACTER VARYING(*n*))

Holds variable-length character data of 1 to 4,000 bytes in size.



Oracle does not recommend using `VARCHAR` and has for many years instead encouraged the use of `VARCHAR2`.

`VARCHAR2(n [BYTE | CHAR]) (SQL2003 datatype: CHARACTER VARYING(n))`

Holds variable-length character data of up to 4,000 bytes in length, as defined by *n*. `BYTE` tells Oracle to use bytes for the size measurement. `CHAR` tells Oracle to use characters for the size measurement. If you use `CHAR`, Oracle internally must still transform that into some number of bytes, which is then subject to the 4,000-byte upper limit.

`XMLTYPE (SQL2003 datatype: XML)`

Stores XML data within the Oracle database. The XML data is accessed using XPath expressions as well as a number of built-in XPath functions, SQL functions, and PL/SQL packages. The `XMLTYPE` datatype is a system-defined type, so it is usable as an argument in functions, or as the datatype of a column in a table or view. When used in a table, the data can be stored in a `CLOB` column or object-relationally.

PostgreSQL Datatypes

The PostgreSQL database supports most SQL2003 datatypes, plus an extremely rich set of datatypes that store spatial and geometric data. PostgreSQL sports a rich set of operators and functions especially for the geometric datatypes, including capabilities such as rotation, finding intersections, and scaling. It also supports additional versions of existing datatypes that are smaller and take up less disk space than their corresponding primary datatypes. For example, PostgreSQL offers several variations on `INTEGER` to accommodate small or large numbers and thereby consume proportionally less or more space. Here's a list of the datatypes it supports:

`BIGINT, INT8 (SQL2003 datatype: none)`

Stores signed or unsigned 8-byte integers within the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

`BIGSERIAL`

See `SERIALS`.

`BIT (SQL2003 datatype: BIT)`

Stores a fixed-length bit string.

`BIT VARYING(n), VARBIT(n) (SQL2003 datatype: BIT VARYING)`

Stores a variable-length bit string whose length is denoted by *n*.

`BOOL, BOOLEAN (SQL2003 datatype: BOOLEAN)`

Stores a logical Boolean (true/false/unknown) value. The keywords `TRUE` and `FALSE` are preferred, but PostgreSQL supports the following valid literal values for the “true” state: `TRUE`, *t*, `true`, *y*, `yes`, and `1`. Valid “false” values are: `FALSE`, *f*, `false`, *n*, `no`, and `0`.

BOX((x1, y1), (x2, y2)) (SQL2003 datatype: none)

Stores the values of a rectangular box in a 2D plane. Values are stored in 32 bytes and are represented as ((x1, y1), (x2, y2)), signifying the opposite corners of the box (upper-right and lower-left corners, respectively). The outer parentheses are optional.

BYTEA (SQL2003 datatype: BINARY LARGE OBJECT)

Holds raw, binary data; typically used to store graphics, sounds, or documents. For storage, this datatype requires 4 bytes plus the actual size of the binary string.

CHAR(n), CHARACTER(n) (SQL2003 datatype: CHARACTER(n))

Contains a fixed-length character string padded with spaces up to a length of *n*. Attempting to insert a value longer than *n* results in an error (unless the extra length is composed of spaces, which are then truncated such that the result fits in *n* characters).

CIDR(x.x.x.x/y) (SQL2003 datatype: none)

Describes an IP version 4 (IPv4) network or host address in a 12-byte storage space. The range is any valid IPv4 network address. Data in *CIDR* datatypes is represented as *x.x.x.x/y*, where the *xs* are the IP address and *y* is the number of bits in the netmask. *CIDR* does not accept nonzero bits to the right of a zero bit in the netmask.

CIRCLE(x, y, r) (SQL2003 datatype: none)

Describes a circle in a 2D plane. Values are stored in 24 bytes of storage space and are represented as (x, y, r). The x, y value represents the coordinates of the center of the circle, while r represents the length of the radius. Parentheses or arrow brackets may optionally delimit the values for x, y, and r.

DATE (SQL2003 datatype: DATE)

Holds a calendar date (year, day, and month) without the time of day in a 4-byte storage space. Dates must be between 4713 BC and 32767 AD. *DATE*'s lowest resolution, naturally, is to the day.

DECIMAL[(p,s)], NUMERIC[(p,s)] (SQL2003 datatype: DECIMAL(p,s), NUMERIC(p,s))

Stores exact numeric values with a precision (p) in the range of 0 to 9 and a scale (s) of 0, with no upper limit.

FLOAT4, REAL (SQL2003 datatype: FLOAT(p))

Stores floating-point numbers with a precision of 0 to 8 and 6 decimal places.

FLOAT8, DOUBLE PRECISION (SQL2003 datatype: FLOAT(p), 7 <= p < 16)

Stores floating-point numbers with a precision of 0 to 16 and 15 decimal places.

INET(x.x.x.x/y) (SQL2003 datatype: none)

Stores an IP version 4 network or host address in a 12-byte storage space. The range is any valid IPv4 network address. The *xs* represent the IP address, and *y* is the number of bits in the netmask. The netmask defaults to 32. Unlike *CIDR*, *INET* accepts nonzero bits to the right of the netmask.

INTEGER, INT, INT4 (SQL2003 datatype: INTEGER)

Stores signed or unsigned 4-byte integers within the range of -2,147,483,648 to 2,147,483,647.

INTERVAL(p) (SQL2003 datatype: none)

Holds general-use time-span values within the range of -178,000,000 to 178,000,000 years in a 12-byte storage space. *INTERVAL*'s lowest resolution is to the microsecond. This is a different datatype than the ANSI standard, which requires an interval qualifier such as *INTERVAL YEAR TO MONTH*.

LINE((x1, y1), (x2, y2)) (SQL2003 datatype: none)

Holds line data, without endpoints, in 2D plane values. Values are stored in 32 bytes and are represented as ((x1, y1), (x2, y2)), indicating the start and end points of a line. The enclosing parentheses are optional for line syntax.

LSEG((x1, y1), (x2, y2)) (SQL2003 datatype: none)

Holds line segment (*LSEG*) data, with endpoints, in a 2D plane. Values are stored in 32 bytes and are represented as ((x1, y1), (x2, y2)). The outer parentheses are optional for *LSEG* syntax. For those who are interested, the “line segment” is what most people traditionally think of as a line. For example, the lines on a playing field are actually line segments.



In true geometric nomenclature, a *line* stretches to infinity, having no terminus at either end, while a *line segment* has end points. PostgreSQL has datatypes for both, but they are functionally equivalent.

MACADDR (SQL2003 datatype: none)

Holds a value for the MAC address of a computer's network interface card in a 6-byte storage space. *MACADDR* accepts a number of industry standard representations, such as:

08002B:010203
08002B-010203
0800.2B01.0203
08-00-2B-01-02-03
08:00:2B:01:02:03

MONEY, DECIMAL(9,2) (SQL2003 datatype: none)

Stores U.S.-style currency values in the range of -21,474,836.48 to 21,474,836.47.

NUMERIC[(p,s)], DECIMAL[(p,s)] (SQL2003 datatype: none)

Stores exact numeric values with a precision (p) and scale (s).

OID (SQL2003 datatype: none)

Stores unique object identifiers.

PATH($(x_1, y_1), \dots, n$) (SQL2003 datatype: none)

Describes an open and closed geometric path in a 2D plane. Values are represented as $[(x_1, y_1), \dots, n]$ and consume $4 + 32n$ bytes of storage space. Each (x, y) value represents a point on the path. Paths are either open, where the first and last points do not intersect, or closed, where the first and last points do intersect. Parentheses are used to encapsulate closed paths, while brackets encapsulate open paths.

POINT(x, y) (SQL2003 datatype: none)

Stores values for a geometric point in a 2D plane in a 16-byte storage space. Values are represented as (x, y) . The point is the basis for all other two-dimensional spatial datatypes supported in PostgreSQL. Parentheses are optional for point syntax.

POLYGON($(x_1, y_1), \dots, n$) (SQL2003 datatype: none)

Stores values for a closed geometric path in a 2D plane using $4 + 32n$ bytes of storage. Values are represented as $((x_1, y_1), \dots, n)$; the enclosing parentheses are optional. **POLYGON** is essentially a closed-path datatype.

SERIAL, SERIAL4 (SQL2003 datatype: none)

Stores an autoincrementing, unique integer ID for indexing and cross-referencing. Can contain up to 4 bytes of data (a range of numbers from 1 to 2,147,483,647). Tables defined with this datatype cannot be directly dropped: you must first issue the **DROP SEQUENCE** command, then follow up with the **DROP TABLE** command.

SERIAL8, BIGSERIAL (SQL2003 datatype: none)

Stores an autoincrementing, unique integer ID for indexing and cross-referencing. Can contain up to 8 bytes of data (a range of numbers from 1 to 9,223,372,036,854,775,807). Tables defined with this datatype cannot be directly dropped: you must first issue the **DROP SEQUENCE** command, then follow up with the **DROP TABLE** command.

SMALLINT (SQL2003 datatype: SMALLINT)

Stores signed or unsigned 2-byte integers within the range of -32,768 to 32,767. **INT2** is a synonym.

TEXT (SQL2003 datatype: CLOB)

Stores large, variable-length character-string data of up to 1 gigabyte. PostgreSQL automatically compresses **TEXT** strings, so the disk size may be less than the string size.

TIME[(p)] [WITHOUT TIME ZONE | WITH TIME ZONE] (SQL2003 datatype: TIME)

Holds the time of day and stores either no time zone (using 8 bytes of storage space) or the time zone of the database server (using 12 bytes of storage space). The allowable range is from 00:00:00.00 to 23:59:59.99. The lowest granularity is 1 microsecond. Note that time zone information on most Unix systems is available only for the years 1902 through 2038.

TIMESTAMP[*p*] [WITHOUT TIME ZONE | WITH TIME ZONE] (SQL2003)

datatype: TIMESTAMP [WITH TIME ZONE | WITHOUT TIME ZONE]

Holds the date and time and stores either no time zone or the time zone of the database server. The range of values is from 4713 BC to 1465001 AD. *TIMESTAMP* uses 8 bytes of storage space per value. The lowest granularity is 1 microsecond. Note that time zone information on most Unix systems is available only for the years 1902 through 2038.

TIMETZ (SQL2003 *datatype: TIME WITH TIME ZONE*)

Holds the time of day, including the time zone.

VARCHAR(*n*), CHARACTER VARYING(*n*) (SQL2003 *datatype: CHARACTER VARYING(*n*)*)

Stores variable-length character strings of up to a length of *n*. Trailing spaces are not stored.

SQL Server Datatypes

Microsoft SQL Server supports most SQL2003 datatypes, as well as some additional datatypes used to uniquely identify rows of data within a table and across multiple servers, such as *UNIQUEIDENTIFIER*. These datatypes are included in support of Microsoft's hardware philosophy of "scale-out" (that is, deploying on many Intel-based servers) rather than "scale-up" (deploying on a single huge, high-end Unix server or a Windows Data Center Server).



Here's an interesting side note about SQL Server dates: SQL Server supports dates starting at the year 1753, and you can't store dates prior to that year using any of SQL Server's date datatypes. Why not? The rationale is that the English-speaking world started using the Gregorian calendar in 1753 (the Julian calendar was used prior to September, 1753), and converting dates prior to Julian to the Gregorian calendar can be quite challenging.

The datatypes SQL Server supports are:

BIGINT (SQL2003 *datatype: BIGINT*)

Stores signed and unsigned integers in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, using 8 bytes of storage space. See *INT* for *IDENTITY* property rules that also apply to *BIGINT*.

BINARY[*n*] (SQL2003 *datatype: BLOB*)

Stores a fixed-length binary value of 1 to 8,000 bytes in size. *BINARY* datatypes consume *n* + 4 bytes of storage space.

BIT (SQL2003 *datatype: BOOLEAN*)

Stores a value of 1, 0, or NULL (to indicate "unknown"). Up to eight *BIT* columns on a single table will be stored in a single byte. An additional eight *BIT* columns consume one more byte of storage space. *BIT* columns cannot be indexed.

CHAR[(n)], CHARACTER[(n)] (SQL2003 datatype: CHARACTER(n))

Holds fixed-length character data of 1 to 8,000 characters in length. Any unused space is, by default, padded with spaces. (You can disable the automatic padding.) Storage size is n bytes.

CURSOR (SQL2003 datatype: none)

A special datatype used to describe a cursor as a variable or stored procedure *OUTPUT* parameter. It cannot be used in a *CREATE TABLE* statement. The CURSOR datatype is always nullable.

DATE (SQL2003 datatype: DATE)

Holds a date in the range of January 1, 0001 AD to December 31, 9999 AD.

DATETIME (SQL2003 datatype: TIMESTAMP)

Holds a date and time within the range of 1753-01-01 00:00:00 through 9999-12-31 23:59:59. Values are stored in an 8-byte storage space.

DATETIME2 (SQL2003 datatype: TIMESTAMP)

Holds a date and time within the range of January 1, 0001 AD to December 31, 9999 AD, to an accuracy of 100 nanoseconds.

DATETIMEOFFSET (SQL2003 datatype: TIMESTAMP)

Holds a date and time within the range of January 1, 0001 AD to December 31, 9999 AD, to an accuracy of 100 nanoseconds. Also includes time zone information. Values are stored in a 10-byte storage space.

DECIMAL(p,s), DEC(p,s), NUMERIC(p,s) (SQL2003 datatype: DECIMAL(p,s), NUMERIC(p,s))

Stores decimal values up to 38 digits long. The values p and s define the precision and scale, respectively. The default value for the scale is 0. The precision of the datatype determines how much storage space it will consume:

Precision 1–9 uses 5 bytes

Precision 10–19 uses 9 bytes

Precision 20–28 uses 13 bytes

Precision 29–39 uses 17 bytes

See *INT* for *IDENTITY* property rules that also apply to *DECIMAL*.

DOUBLE PRECISION (SQL2003 datatype: none)

Synonym for *FLOAT(53)*.

FLOAT[(n)] (SQL2003 datatype: FLOAT, FLOAT(n))

Holds floating-point numbers in the range of -1.79E+308 through 1.79E+308. The precision, represented by n , may be in the range of 1 to 53. The storage size is 4 bytes for 7 digits, where n is in the range of 1 to 24. Anything larger requires 8 bytes of storage.

HIERARCHYID (SQL2003 datatype: none)

Represents a hierarchy or tree structure within the relational data. Although it may consume more space, *HIERARCHYID* will usually consume 5 bytes or less. Refer to the vendor documentation for more information on this special datatype.

IMAGE (SQL2003 datatype: BLOB)

Stores a variable-length binary value of up to 2,147,483,647 bytes in length. This datatype is commonly used to store graphics, sounds, and files such as MS-Word documents and MS-Excel spreadsheets. *IMAGE* cannot be freely manipulated; both *IMAGE* and *TEXT* columns have a lot of constraints on how they can be used. See *TEXT* for a list of the commands and functions that work on an *IMAGE* datatype.

INT [IDENTITY [(seed, increment)]] (SQL2003 datatype: INTEGER)

Stores signed or unsigned integers within the range of -2,147,483,648 to 2,147,483,647 in 4 bytes of storage space. All integer datatypes, as well as the decimal type, support the *IDENTITY* property. An identity is an automatically incrementing row identifier. Refer to the section “CREATE/ALTER DATABASE Statement” in Chapter 3 for more information.

MONEY (SQL2003 datatype: none)

Stores monetary values within the range of -922,337,203,685,477.5808 to 922,337,203,685,477.5807, in an 8-byte storage space.

NCHAR(n), NATIONAL CHAR(n), NATIONAL CHARACTER(n) (SQL2003 datatype: NATIONAL CHARACTER(n))

Holds fixed-length UNICODE data of up to 4,000 characters in length. The storage space consumed is double the character length inserted into the field ($2 * n$).

NTEXT, NATIONAL TEXT (SQL2003 datatype: NCLOB)

Holds UNICODE text passages of up to 1,073,741,823 characters in length. See *TEXT* for rules about the commands and functions available for *NTEXT*.

NUMERIC(p,s) (SQL2003 datatype: DECIMAL(p,s))

Synonym for *DECIMAL*. See *INT* for rules about the *IDENTITY* property that also apply to this type.

NVARCHAR(n), NATIONAL CHAR VARYING(n), NATIONAL CHARACTER VARYING(n) (SQL2003 datatype: NATIONAL CHARACTER VARYING(n))

Holds variable-length UNICODE data of up to 4,000 characters in length. The storage space consumed is double the character length inserted into the field ($2 * n$). The system setting *SET ANSI_PADDING* is always enabled (ON) for *NCHAR* and *NVARCHAR* fields in SQL Server.

REAL, FLOAT(24) (SQL2003 datatype: REAL)

Holds floating-point numbers in the range of -3.40E+38 through 3.40E+38 in a 4-byte storage space. *REAL* is functionally equivalent to *FLOAT(24)*.

ROWVERSION (SQL2003 datatype: none)

Stores a number that is unique within the database whenever a row in the table is updated. Called *TIMESTAMP* in earlier versions.

SMALLDATETIME (SQL2003 datatype: none)

Holds a date and time within the range of ‘1900-01-01 00:00’ through ‘2079-06-06 23:59’, accurate to the nearest minute. (Minutes are rounded down when seconds are 29.998 or less; otherwise, they are rounded up.) Values are stored in 4 bytes.

SMALLINT (SQL2003 datatype: SMALLINT)

Stores signed or unsigned integers in the range of -32,768 and 32,767, in 2 bytes of storage space. See *INT* for rules about the *IDENTITY* property that also apply to this type.

SMALLMONEY (SQL2003 datatype: none)

Stores monetary values within the range of -214,748.3648 to 214,748.3647, in 4 bytes of storage space.

SQL_VARIANT (SQL2003 datatype: none)

Stores values of other SQL Server-supported datatypes, except *TEXT*, *NTEXT*, *ROWVERSION*, and other *SQL_VARIANT* commands. Can store up to 8,016 bytes of data and supports *NULL* and *DEFAULT* values. *SQL_VARIANT* is used in columns, parameters, variables, and return values of functions and stored procedures.

TABLE (SQL2003 datatype: none)

Special datatype that stores a result set for a later process. Used solely in procedural processing, and cannot be used in a *CREATE TABLE* statement. This datatype alleviates the need for temporary tables in many applications. It can reduce the need for stored procedure recompiles, thus speeding execution of stored procedures and user-defined functions.

TEXT (SQL2003 datatype: CLOB)

Stores very large passages of text (up to 2,147,483,647 characters in length). *TEXT* and *IMAGE* values are often more difficult to manipulate than, say, *VARCHAR* values. For example, you cannot place an index on a *TEXT* or *IMAGE* column. *TEXT* values can be manipulated using the functions *DATALENGTH*, *PATINDEX*, *SUBSTRING*, *TEXTPTR*, and *TEXTVALID* as well as the commands *READTEXT*, *SET TEXTSIZE*, *UPDATETEXT*, and *WRITETEXT*.

TIME (SQL2003 datatype: TIME)

Stores an automatically generated binary number that guarantees uniqueness in the current database and is therefore different from the ANSI *TIMESTAMP* datatype. *TIMEs* consume 8 bytes of storage space. *ROWVERSION* is now preferred over *TIME* to uniquely track each row.

TIMESTAMP (SQL2003 datatype: TIMESTAMP)

Stores the time of day based on a 24-hour clock without time zone awareness, to an accuracy of 100 nanoseconds, in a 5-byte storage space.

TINYINT (SQL2003 datatype: none)

Stores unsigned integers within the range 0 to 255 in 1 byte of storage space. See *INT* for rules about the *IDENTITY* property that also apply to this type.

UNIQUEIDENTIFIER (SQL2003 datatype: none)

Represents a value that is globally unique across all databases and all servers. Values are represented as *xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx*, where each *x* is a hexadecimal digit in the range 0 to 9 or a to f. The only operations allowed against *UNIQUEIDENTIFIERs* are comparisons and *NULL* checks. Column constraints and properties are allowed on *UNIQUEIDENTIFIER* columns, with the exception of the *IDENTITY* property.

VARBINARY[(n)] (SQL2003 datatype: BLOB)

Describes a variable-length binary value of up to 8,000 bytes in size. The storage space consumed is equivalent to the size of the data inserted, plus 4 bytes.

VARCHAR[(n)], CHAR VARYING[(n)], CHARACTER VARYING[(n)] (SQL2003 datatype: CHARACTER VARYING(n))

Holds fixed-length character data of 1 to 8,000 characters in length. The amount of storage space required is determined by the actual size of the value entered in bytes, not the value of *n*.

XML (SQL2003 datatype: XML)

Stores XML data in a column or a variable of variable size in storage space up to but not exceeding 2 gigabytes in size.

Constraints

Constraints allow you to automatically enforce the integrity of data and to filter the data that is placed in a database. In a sense, constraints are rules that define which data values are valid during *INSERT*, *UPDATE*, and *DELETE* operations. When a data-modification transaction breaks the rules of a constraint, the transaction is rejected.

In the ANSI standard, there are four constraint types: *CHECK*, *PRIMARY KEY*, *UNIQUE*, and *FOREIGN KEY*. (The RDBMS platforms may allow more; refer to Chapter 3 for details.)

Scope

Constraints may be applied at the column level or the table level:

Column-level constraints

Are declared as part of a column definition and apply only to that column.

Table-level constraints

Are declared independently from any column definitions (traditionally, at the end of a *CREATE TABLE* statement) and may apply to one or more columns in the table. A table constraint is required when you wish to define a constraint that applies to more than one column.

Syntax

Constraints are defined when you create or alter a table. The general syntax for constraints is shown here:

```
CONSTRAINT [constraint_name] constraint_type [(column[, ...])]  
[predicate] [constraint_defer] [deferment_timing]
```

The syntax elements are as follows:

`CONSTRAINT [constraint_name]`

Begins a constraint definition and, optionally, provides a name for the constraint. When you omit `constraint_name`, the system will create a name for you automatically. On some platforms, you may omit the `CONSTRAINT` keyword as well.



System-generated names are often incomprehensible. It is good practice to specify human-readable, sensible names for constraints.

`constraint_type`

Declares the constraint as one of the allowable types: `CHECK`, `PRIMARY KEY`, `UNIQUE`, or `FOREIGN KEY`. More information about each type of constraint appears later in this section.

`column[, ...]`

Associates one or more columns with the constraint. Specify the columns in a comma-delimited list, enclosed in parentheses. The column list should be omitted for column-level constraints. Columns are not used in every constraint. For example, `CHECK` constraints do not generally use column references.

`predicate`

Defines a predicate for `CHECK` constraints.

`constraint_deferment`

Declares a constraint as `DEFERRABLE` or `NOT DEFERRABLE`. When a constraint is deferrable, you can specify that it be checked for a rules violation at the end of a transaction. When a constraint is not deferrable, it is checked for a rules violation at the conclusion of every SQL statement.

`deferment_timing`

Declares a deferrable constraint as `INITIALLY DEFERRED` or `INITIALLY IMMEDIATE`. When set to `INITIALLY DEFERRED`, the constraint check time will be deferred until the end of a transaction, even if the transaction is composed of many SQL statements. In this case, the constraint must also be `DEFERRABLE`. When set to `INITIALLY IMMEDIATE`, the constraint is checked at the end of every SQL statement. In this case, the constraint may be either `DEFERRABLE` or `NOT DEFERRABLE`. The default is `INITIALLY IMMEDIATE`.

Note that this syntax may vary among the different vendor platforms. Check the individual platform sections in Chapter 3 for more details.

PRIMARY KEY Constraints

A `PRIMARY KEY` constraint declares one or more columns whose values uniquely identify each record in the table. It is considered a special case of the `UNIQUE` constraint. Here are some rules about primary keys:

- Only one primary key may exist on a table at a time.
- Columns in the primary key cannot have datatypes of *BLOB*, *CLOB*, *NCLOB*, or *ARRAY*.
- Primary keys may be defined at the column level for a single column key or at the table level if multiple columns make up the primary key.
- Values in the primary key column(s) must be unique and not *NULL*.
- In a multicolumn primary key, called a *concatenated key*, the combination of values in all of the key columns must be unique and not *NULL*.
- Foreign keys can be declared that reference the primary key of a table to establish direct relationships between tables (or possibly, though rarely, within a single table).

The following ANSI standard code includes the options for creating both a table- and column-level primary key constraint on a table called **distributors**. The first example shows a column-level primary-key constraint, while the second shows a table-level constraint:

```
-- Creating a column-level constraint
CREATE TABLE distributors
  (dist_id      CHAR(4)      NOT NULL PRIMARY KEY,
   dist_name    VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city         VARCHAR(20),
   state        CHAR(2)      ,
   zip          CHAR(5)      ,
   phone        CHAR(12)     ,
   sales_rep    INT         );
-- Creating a table-level constraint
CREATE TABLE distributors
  (dist_id      CHAR(4)      NOT NULL,
   dist_name    VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city         VARCHAR(20),
   state        CHAR(2)      ,
   zip          CHAR(5)      ,
   phone        CHAR(12)     ,
   sales_rep    INT         ,
   CONSTRAINT    pk_dist_id PRIMARY KEY (dist_id));
```

In the example showing a table-level primary key, we could easily have created a concatenated key by listing several columns separated by commas.

FOREIGN KEY Constraints

A *FOREIGN KEY* constraint defines one or more columns in a table as referencing columns in a unique or primary key in another table. (A foreign key can reference a unique or primary key in the same table as the foreign key itself, but

such foreign keys are rare.) Foreign keys can then prevent the entry of data into a table when there is no matching value in the related table. They are the primary means of identifying the relationships between tables in a relational database. Here are some rules about foreign keys:

- Many foreign keys may exist on a table at a time.
- A foreign key can be declared to reference either the primary key or a unique key of another table to establish a direct relationship between the two tables.

The full SQL2003 syntax for foreign keys is more elaborate than the general syntax for constraints shown earlier, and it's dependent on whether you are making a table-level or column-level declaration:

```
-- Table-level foreign key
[CONSTRAINT [constraint_name] ]
FOREIGN KEY (local_column[, ...] )
  REFERENCES referenced_table [ (referenced_column[, ...]) ]
  [MATCH {FULL | PARTIAL | SIMPLE} ]
  [ON UPDATE {NO ACTION | CASCADE | RESTRICT |
    SET NULL | SET DEFAULT} ]
  [ON DELETE {NO ACTION | CASCADE | RESTRICT |
    SET NULL | SET DEFAULT} ]
  [constraint_deferment] [deferment_timing]

-- Column-level foreign key
[CONSTRAINT [constraint_name] ]
  REFERENCES referenced_table [ (referenced_column[, ...]) ]
  [MATCH {FULL | PARTIAL | SIMPLE} ]
  [ON UPDATE {NO ACTION | CASCADE | RESTRICT |
    SET NULL | SET DEFAULT} ]
  [ON DELETE {NO ACTION | CASCADE | RESTRICT |
    SET NULL | SET DEFAULT} ]
  [constraint_deferment] [deferment_timing]
```

The keywords common to a standard constraint declaration were described earlier, in the “Syntax” section. Keywords specific to foreign keys are described in the following list:

FOREIGN KEY (local_column[, ...])

Declares one or more columns of the table being created or altered that are subject to the foreign key constraint. This syntax is used *only* in table-level declarations and is excluded from column-level declarations. We recommend that the ordinal positions and datatypes of the columns in the *local_column* list match the ordinal positions and datatypes of the columns in the *referenced_column* list.

REFERENCES referenced_table [(referenced_column[, ...])]

Names the table and, where appropriate, the column(s) that hold the valid list of values for the foreign key. A *referenced_column* must already be named in a *NOT DEFERRABLE PRIMARY KEY* or *NOT DEFERRABLE UNIQUE KEY* statement. The table types must also match; for example, if one is a local temporary table, both must be local temporary tables.

MATCH {FULL | PARTIAL | SIMPLE}

Defines the degree of matching required between the local and referenced columns in foreign-key constraints when NULLs are present:

FULL

Declares that a match is acceptable when: 1) none of the referencing columns are NULL and match all of the values of the referenced column, or 2) all of the referencing columns are NULL. In general, you should either use *MATCH FULL* or ensure that all columns involved have *NOT NULL* constraints.

PARTIAL

Declares that a match is acceptable when at least one of the referenced columns is NULL and the others match the corresponding referenced columns.

SIMPLE

Declares that a match is acceptable when any of the values of the referencing column is NULL or a match. This is the default.

ON UPDATE

Specifies that, when an *UPDATE* operation affects one or more referenced columns of the primary or unique key on the referenced table, a corresponding action should be taken to ensure that the foreign key does not lose data integrity. *ON UPDATE* may be declared independently of or together with the *ON DELETE* clause. When omitted, the default for the ANSI standard is *ON UPDATE NO ACTION*.

ON DELETE

Specifies that, when a *DELETE* operation affects one or more referenced columns of the primary or unique key on the referenced table, a corresponding action should be taken to ensure that the foreign key does not lose data integrity. *ON DELETE* may be declared independently of or together with the *ON UPDATE* clause. When omitted, the default for the ANSI standard is *ON DELETE NO ACTION*.

NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT

Defines the action the database takes to maintain the data integrity of the foreign key when a referenced primary or unique key constraint value is changed or deleted:

NO ACTION

Tells the database to do nothing when a primary key or unique key value referenced by a foreign key is changed or deleted.

CASCADE

Tells the database to perform the same action (i.e., *DELETE* or *UPDATE*) on the matching foreign key when a primary key or unique key value is changed or deleted.

RESTRICT

Tells the database to prevent changes to the primary key or unique key value referenced by the foreign key.

SET NULL

Tells the database to set the value in the foreign key to NULL when a primary key or unique key value is changed or deleted.

SET DEFAULT

Tells the database to set the value in the foreign key to the default (using default values you specify for each column) when a primary key or unique key value is changed or deleted.

As with the code example for primary keys, you can adapt this generic syntax to both column-level and table-level foreign key constraints. Note that column-level and table-level constraints perform their function in exactly the same way; they are merely defined at different levels of the *CREATE TABLE* command. In the following example, we create a single-column foreign key on the **salesrep** column referencing the **empid** column of the **employee** table. We create the foreign key two different ways, the first time at the column level and the second time at the table level:

```
-- Creating a column-level constraint
CREATE TABLE distributors
  (dist_id      CHAR(4)      PRIMARY KEY,
   dist_name    VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city         VARCHAR(20),
   state        CHAR(2)      ,
   zip          CHAR(5)      ,
   phone        CHAR(12)     ,
   sales_rep    INT         NOT
                           NULL REFERENCES employee(empid));

-- Creating a table-level constraint
CREATE TABLE distributors
  (dist_id      CHAR(4)      NOT NULL,
   dist_name    VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city         VARCHAR(20),
   state        CHAR(2)      ,
   zip          CHAR(5)      ,
   phone        CHAR(12)     ,
   sales_rep    INT         ,
   CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
   CONSTRAINT fk_empid
     FOREIGN KEY (sales_rep)
     REFERENCES employee(empid));
```

UNIQUE Constraints

A *UNIQUE* constraint, sometimes called a *candidate key*, declares that the values in one column, or the combination of values in more than one column, must be unique. Rules concerning unique constraints include:

- Columns in a unique key cannot have datatypes of *BLOB*, *CLOB*, *NCLOB*, or *ARRAY*.
- The column or columns in a unique key may not be identical to those in any other unique keys, or to any columns in the primary key of the table.
- A single *NULL* value, if the unique key allows *NULL* values, is allowed.
- SQL2003 allows you to substitute the column list shown in the general syntax diagram for constraints with the keyword *(VALUE)*. *UNIQUE (VALUE)* indicates that all columns in the table are part of the unique key. The *VALUE* keyword also disallows any other unique or primary keys on the table.

In the following example, we limit the number of distributors we do business with to only one distributor per zip code. We also allow one (and only one) “catch-all” distributor with a *NULL* zip code. This functionality can be implemented easily using a *UNIQUE* constraint, either at the column or the table level:

```
-- Creating a column-level constraint
CREATE TABLE distributors
  (dist_id      CHAR(4)      PRIMARY KEY,
   dist_name    VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city         VARCHAR(20),
   state        CHAR(2)      ,
   zip          CHAR(5)      UNIQUE,
   phone        CHAR(12)      ,
   sales_rep    INT          NOT NULL
   REFERENCES employee(empid));

-- Creating a table-level constraint
CREATE TABLE distributors
  (dist_id      CHAR(4)      NOT NULL,
   dist_name    VARCHAR(40),
   dist_address1 VARCHAR(40),
   dist_address2 VARCHAR(40),
   city         VARCHAR(20),
   state        CHAR(2)      ,
   zip          CHAR(5)      ,
   phone        CHAR(12)      ,
   sales_rep    INT          ,
   CONSTRAINT pk_dist_id PRIMARY KEY (dist_id),
   CONSTRAINT fk_emp_id FOREIGN KEY (sales_rep)
   REFERENCES employee(empid),
   CONSTRAINT unq_zip UNIQUE (zip) );
```

CHECK Constraints

CHECK constraints allow you to perform comparison operations to ensure that values match specific conditions that you set out. The syntax for a check constraint is very similar to the general syntax for constraints:

```
[CONSTRAINT] [constraint_name] CHECK (search_conditions)
[constraint_deferment] [deferment_timing]
```

Most of the elements of the check constraint were introduced earlier in this section. The following element is unique to this constraint:

search_conditions

Specifies one or more search conditions that constrain the values inserted into the column or table, using one or more expressions and a predicate. Multiple search conditions may be applied to a column in a single check constraint using the *AND* and *OR* operators (think of a *WHERE* clause).

A check constraint is considered matched when the search conditions evaluate to *TRUE* or *UNKNOWN*. Check constraints are limited to Boolean operations (e.g., *=*, *>=*, *<=*, or *<>*), though they may include any SQL2003 predicate, such as *IN* or *LIKE*. Check constraints may be appended to one another (when checking a single column) using the *AND* and *OR* operators. Here are some other rules about check constraints:

- A column or table may have one or more check constraints.
- A search condition cannot contain aggregate functions, except in a subquery.
- A search condition cannot use nondeterministic functions or subqueries.
- A check constraint can only reference like objects. That is, if a check constraint is declared on a global temporary table, it cannot then reference a permanent table.
- A search condition cannot reference these ANSI functions: *CURRENT_USER*, *SESSION_USER*, *SYSTEM_USER*, *USER*, *CURRENT_PATH*, *CURRENT_DATE*, *CURRENT_TIME*, *CURRENT_TIMESTAMP*, *LOCALTIME*, and *LOCALTIMESTAMP*.

The following example adds a check constraint to the *dist_id* and *zip* columns. (This example uses generic code run on SQL Server.) The *zip* code must fall into the normal ranges for postal zip codes, while the *dist_id* values are allowed to contain either four alphabetic characters or two alphabetic and two numeric characters:

```
-- Creating column-level CHECK constraints
CREATE TABLE distributors
  (dist_id      CHAR(4)
  CONSTRAINT pk_dist_id PRIMARY KEY
  CONSTRAINT ck_dist_id CHECK
    (dist_id LIKE '[A-Z][A-Z][A-Z][A-Z]' OR
     dist_id LIKE '[A-Z][A-Z][0-9][0-9]'),
```

```
dist_name      VARCHAR(40),
dist_address1 VARCHAR(40),
dist_address2 VARCHAR(40),
city          VARCHAR(20),
state          CHAR(2)
    CONSTRAINT def_st DEFAULT ("CA"),
zip           CHAR(5)
    CONSTRAINT unq_dist_zip UNIQUE
    CONSTRAINT ck_dist_zip CHECK
        (zip LIKE '[0-9][0-9][0-9][0-9][0-9]'),
phone         CHAR(12),
sales_rep     INT
    NOT NULL DEFAULT USER REFERENCES employee(emp_id))
```



3

SQL Statement Command Reference

This chapter is the heart of the book. It consists of an alphabetical listing of SQL commands, with detailed explanations and examples. Each command and function is identified in Table 3-1 as being “supported,” “supported with variations,” “supported with limitations,” or “not supported,” for each of the four SQL dialects covered in this book: MySQL 5.1, Oracle Database 11g, PostgreSQL 8.2.1, and Microsoft’s SQL Server 2008. After a brief description of the SQL2003 standard, each vendor application is discussed briefly but thoroughly, with supporting examples. If a specific platform does not support a particular command, that fact will be noted in the table that heads the command, and that command will not be listed for that platform. Similarly, although this book is not a comprehensive review of the SQL2003 standard, each command has been validated against that standard.

How to Use This Chapter

When researching a command in this chapter:

1. Read “SQL Platform Support.”
2. Check the platform support table.
3. Read the section on the SQL2003 syntax and description, even if you are looking for a specific platform implementation.
4. Finally, read the specific platform implementation information.

Any common features between the platform implementations of a command are discussed and compared against the SQL2003 section. Thus, the subsection on a platform’s implementation of a particular command may not describe every aspect of that command, since some of its details may be covered in the SQL2003 section. Please note that if there is a keyword that appears in a command’s syntax but not in its keyword description, this is because we chose not to repeat descriptions that appear under the ANSI entry.

SQL Platform Support

Table 3-1 provides a listing of the SQL statements, the platforms that support them, and the degree to which they support them. The following list offers useful tips for reading Table 3-1, as well as an explanation of what each abbreviation stands for:

1. The first column contains the SQL commands, in alphabetical order.
2. The SQL statement class for each command is indicated in the second column.
3. The subsequent columns list the level of support for each vendor:

Supported (S)

The platform supports the SQL2003 standard for the particular command.

Supported, with variations (SWV)

The platform supports the SQL2003 standard for the particular command, using vendor-specific code or syntax.

Supported, with limitations (SWL)

The platform supports some but not all of the functions specified by the SQL2003 standard for the particular command.

Not supported (NS)

The platform does not support the particular command according to the SQL2003 standard.

The sections that follow the table describe the commands in detail. Related *CREATE* and *ALTER* commands (e.g., *CREATE DATABASE* and *ALTER DATABASE*) are discussed together (e.g., in a section titled “*CREATE/ALTER DATABASE Statement*”).

Remember that even if a specific SQL2003 command is listed in the table as “Not supported,” the platform usually has alternative coding or syntax to enact the same command or function. Therefore, be sure to read the discussion and examples for each command later in this chapter. Likewise, a few of the commands in Table 3-1 are not found in the SQL2003 standard; these have been indicated with the term “Non-ANSI” under the heading “SQL2003 class” in the table.

There are also a few ANSI commands in the table (such as *CREATE DOMAIN* and *ALTER DOMAIN*) that are not currently supported by any of the database platforms covered in this text. Since this book focuses on the implementation of the SQL language, unsupported ANSI commands are shown in Table 3-1 but are not documented elsewhere in the book.

Table 3-1. Alphabetical quick SQL command reference

SQL2003 command	SQL2003 class	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
<i>ALL/ANY/SOME</i>	SQL-data	S	S	S	S
<i>BETWEEN</i>	SQL-data	S	S	S	S

Table 3-1. Alphabetical quick SQL command reference (continued)

SQL2003 command	SQL2003 class	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
ALTER DATABASE	SQL-schema	SWV	SWV	SWV	SWV
ALTER FUNCTION	SQL-schema	SWL	SWV	SWL	SWV
ALTER INDEX	Non-ANSI	SWV	SWV	SWV	SWV
ALTER METHOD	SQL-schema	NS	NS	NS	NS
ALTER PROCEDURE	SQL-schema	SWV	SWV	NS	SWV
ALTER ROLE	SQL-schema	NS	SWV	SWV	SWL
ALTER SCHEMA	SQL-schema	SWL	NS	SWL	SWL
ALTER TABLE	SQL-schema	SWV	SWV	SWV	SWV
ALTER TRIGGER	Non-ANSI	NS	SWV	SWV	SWV
ALTER TYPE	SQL-schema	NS	SWV	NS	NS
ALTER VIEW	Non-ANSI	SWV	SWV	SWV	SWV
CALL	SQL-control	S	SWV	NS	NS
CLOSE CURSOR	SQL-data	NS	S	S	SWV
COMMIT	SQL-transaction	SWV	SWV	SWV	SWV
CONNECT	SQL-connection	NS	S	NS	SWV
CREATE DATABASE	Non-ANSI	SWV	SWV	SWV	SWV
CREATE FUNCTION	SQL-schema	SWL	SWV	SWL	SWV
CREATE INDEX	Non-ANSI	SWV	SWV	SWV	SWV
CREATE METHOD	SQL-schema	NS	NS	NS	NS
CREATE PROCEDURE	SQL-schema	SWV	SWV	NS	SWV
CREATE ROLE	SQL-schema	NS	SWV	SWV	SWL
CREATE SCHEMA	SQL-schema	SWL	SWV	SWL	SWL
CREATE TABLE	SQL-schema	SWV	SWV	SWV	SWV
CREATE TRIGGER	SQL-schema	SWL	SWV	SWV	SWV
CREATE TYPE	SQL-schema	NS	SWL	SWV	SWV
CREATE VIEW	SQL-schema	SWV	SWV	SWV	SWV
DECLARE CURSOR	SQL-data	SWL	SWL	SWL	SWL
DELETE	SQL-data	SWV	SWV	SWV	SWV
DISCONNECT	SQL-connection	NS	SWL	NS	SWL
DROP DATABASE	Non-ANSI	SWV	S	SWV	SWV
DROP DOMAIN	SQL-schema	NS	NS	S	NS
DROP FUNCTION	SQL-schema	SWV	SWV	SWV	SWV

Table 3-1. Alphabetical quick SQL command reference (continued)

SQL2003 command	SQL2003 class	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
<i>DROP INDEX</i>	Non-ANSI	SWV	SWV	SWV	SWV
<i>DROP METHOD</i>	SQL-schema	NS	SWV	NS	NS
<i>DROP</i>	SQL-schema	SWV	S	NS	S
<i>PROCEDURE</i>					
<i>DROP ROLE</i>	SQL-schema	NS	SWV	SWV	SWV
<i>DROP SCHEMA</i>	SQL-schema	SWV	SWV	SWV	SWV
<i>DROP TABLE</i>	SQL-schema	SWV	SWV	SWV	SWV
<i>DROP TRIGGER</i>	SQL-schema	SWV	SWV	SWV	SWV
<i>DROP TYPE</i>	SQL-schema	NS	S	S	NS
<i>DROP VIEW</i>	SQL-schema	SWV	S	S	S
<i>EXCEPT</i>	SQL-data	NS	SWL	SWL	SWL
<i>EXISTS</i>	SQL-data	S	S	S	S
<i>FETCH</i>	SQL-data	SWL	SWL	SWV	SWV
<i>GRANT</i>	SQL-schema	SWV	SWV	SWV	SWV
<i>IN</i>	SQL-data	S	S	S	S
<i>INSERT</i>	SQL-data	SWV	SWV	SWV	SWV
<i>INTERSECT</i>	SQL-data	NS	SWL	SWL	SWL
<i>IS</i>	SQL-data	S	S	S	S
<i>JOIN</i>	SQL-data	SWV	S	SWV	SWL
<i>LIKE</i>	SQL-data	S	S	SWV	S
<i>MERGE</i>	SQL-data	NS	SWV	NS	SWV
<i>OPEN</i>	SQL-data	S	S	NS	S
<i>ORDER BY</i>	SQL-data	SWL	SWV	SWV	SWL
<i>RELEASE SAVEPOINT</i>	SQL-transaction	S	NS	S	NS
<i>RETURN</i>	SQL-control	S	S	S	S
<i>REVOKE</i>	SQL-schema	SWV	SWV	SWV	SWV
<i>ROLLBACK</i>	SQL-transaction	SWL	SWV	SWV	SWV
<i>SAVEPOINT</i>	SQL-transaction	S	S	S	SWL
<i>SELECT</i>	SQL-data	SWV	SWV	SWV	SWV
<i>SET</i>	SQL-session	S	NS	S	S
<i>SET CONNECTION</i>	SQL-connection	NS	NS	NS	SWL
<i>SET CONSTRAINT</i>	SQL-connection	NS	SWV	SWV	NS
<i>SET PATH</i>	SQL-session	NS	NS	NS	NS
<i>SET ROLE</i>	SQL-session	NS	SWV	NS	NS
<i>SET SCHEMA</i>	SQL-session	NS	NS	NS	NS
<i>SET SESSION AUTHORIZATION</i>	SQL-session	NS	NS	S	NS
<i>SET TIME ZONE</i>	SQL-session	NS	SWV	SWL	NS

Table 3-1. Alphabetical quick SQL command reference (continued)

SQL2003 command	SQL2003 class	MySQL v5.1	Oracle v11g	PostgreSQL v8.2.1	SQL Server 2008
SET TRANSACTION	SQL-session	SWV	SWL	S	SWV
START TRANSACTION	SQL-transaction	SWL	NS	NS	NS
SUBQUERY	SQL-data	SWL	S	S	S
TRUNCATE TABLE	In the SQL-data style (not an official SQL3 statement)	S	S	S	S
UNION	SQL-data	NS	SWL	SWL	SWL
UPDATE	SQL-data	SWV	SWV	SWV	SWV
WHERE	SQL-data	S	S	S	S

SQL Command Reference

ALL/ANY/SOME Operators

The *ALL* operator performs a Boolean test of a subquery for the existence of a value in all rows. The *ANY* operator and its synonym *SOME* perform a Boolean test of a subquery for the existence of a value in any of the rows tested.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
SELECT ...
WHERE expression comparison {ALL | ANY | SOME} ( subquery )
```

Keywords

WHERE expression

Tests a scalar expression (such as a column) against every value in the *subquery* for *ALL*, and against every value until a match is found for *ANY* and *SOME*. All rows must match the expression to return a Boolean *TRUE* value for the *ALL* operator, while one or more rows must match the expression to return a Boolean *TRUE* value for the *ANY* and *SOME* operators.

comparison

Compares the *expression* to the *subquery*. The *comparison* must be a standard comparison operator like *=*, *<>*, *!=*, *>*, *>=*, *<*, or *<=*.

Rules at a Glance

The *ALL* operator returns a Boolean *TRUE* value when one of two things happens: either the subquery returns an empty set (i.e., no records), or every record in the set meets the comparison. *ALL* returns *FALSE* when any record in the set does not match the value comparison. The *ANY* and *SOME* operators return a Boolean *TRUE* when at least one record in the subquery matches the comparison operation, and *FALSE* when no record matches the comparison operation (or when a subquery returns an empty result set). If even one return value of the subquery is *NULL*, the operation evaluates as *NULL*, not as *TRUE*.



Do not include special clauses like *ORDER BY*, *GROUP BY*, *CUBE*, *ROLLUP*, *WITH*, etc. in your subquery.

For example, this query returns authors who currently have no titles:

```
SELECT au_id
  FROM authors
 WHERE au_id <> ALL(SELECT au_id FROM titleauthor)
```

You can use *ANY* or *SOME* to perform filtering checks of different kinds. For example, the following query will retrieve from the *employees* table any records that exist in the *employees_archive* table where the employee has the same *job_lvl* as any employee working in Anchorage, Alaska:

```
SELECT *
  FROM employees
 WHERE job_lvl = ANY(SELECT job_lvl FROM employees_archive
                      WHERE city = 'Anchorage')
```

Programming Tips and Gotchas

The *ALL* and *ANY/SOME* operators are somewhat difficult to get used to. Most developers find it easier to use similar functions like *IN* and *EXISTS*.



EXISTS is semantically equivalent to the *ANY/SOME* operator.

MySQL

MySQL supports *ALL* and *ANY/SOME* in the manner described above, with the exception of versions of MySQL prior to 4.0. On pre-v4.0 MySQL platforms, consider using the *IN* operator instead of *EXISTS* for *ANY/SOME*.

Oracle

Oracle supports the ANSI standard versions of *ALL* and *ANY/SOME* with one minor variation, which is that you can supply a list of values instead of a subquery. For example, to find all employees who have a *job_lvl* value equal to 9 or 14:

```
SELECT * FROM employee
 WHERE job_lvl = ALL(9, 14);
```

PostgreSQL

PostgreSQL supports *ALL* and *ANY/SOME* in the manner described above.

SQL Server

SQL Server supports the ANSI standard versions of *ALL* and *ANY/SOME*. It also supports some additional comparison operators: not greater than (*!>*) and not less than (*!<*).

See Also

BETWEEN
EXISTS
SELECT
WHERE

BETWEEN Operator

The *BETWEEN* operator performs a Boolean test of a value against a range of values. It returns *TRUE* when the value is included in the range and *FALSE* when the value falls outside of the range. The results are *NULL* (unknown) if any of the range values are *NULL*.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
SELECT ...
WHERE expression [NOT] BETWEEN lower_range AND upper_range
```

Keywords

WHERE *expression*

Compares a scalar expression, such as a column, to the range of values bounded by *upper_range* and *lower_range*.

[NOT] BETWEEN *lower_range* AND *upper_range*

Compares the *expression* to the *lower_range* and *upper_range*. The comparison is inclusive, meaning that it is equivalent to saying “where *expression* is [not] greater than or equal to *lower_range* and less than or equal to *upper_range*.”

Rules at a Glance

The *BETWEEN* operator is used to test an expression against a range of values. The *BETWEEN* operator may be used with any datatype except *BLOB*, *CLOB*, *NCLOB*, *REF*, and *ARRAY*.

For example, this query returns *title_ids* that have year-to-date sales of between 10,000 and 20,000:

```
SELECT title_id
FROM titles
WHERE ytd_sales BETWEEN 10000 AND 20000
```

CALL Statement

BETWEEN is inclusive of the range of values listed, so it includes the values 10,000 and 20,000 in the search. If you want an exclusive search, you must use the greater than (*>*) and less than (*<*) symbols:

```
SELECT title_id
  FROM titles
 WHERE ytd_sales > 10000
   AND ytd_sales < 20000
```

The *NOT* operator allows you to search for values outside of the *BETWEEN* range. For example, you can find all the **title_ids** that were not published during 2003:

```
SELECT title_id
  FROM titles
 WHERE pub_date NOT BETWEEN '01-JAN-2003'
   AND '31-DEC-2003'
```

Programming Tips and Gotchas

Some coders are very particular about how the keyword *AND* is used in *WHERE* clauses. To prevent a casual reviewer from thinking that the *AND* used in a *BETWEEN* operation is a logical *AND* operator, you might want to use parentheses to encapsulate the entire *BETWEEN* clause:

```
SELECT title_id
  FROM titles
 WHERE (ytd_sales BETWEEN 10000 AND 20000)
   AND pubdate >= '1991-06-12 00:00:00.000'
```

Platform Differences

All of the platforms support *BETWEEN* in the manner described above.

See Also

ALL/ANY/SOME
EXISTS
SELECT
WHERE

CALL Statement

The *CALL* statement invokes a stored procedure.

Platform	Command
MySQL	Supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
CALL procedure_name([parameter[, ...]])
```

Keywords

CALL *procedure_name*

Specifies the stored procedure that you want to invoke. It must be a previously defined stored procedure that is available in the current user context (instance, database, schema, etc.).

(*parameter*[, ...])

Provides values for each input parameter required by the stored procedure. Each parameter listed is required by the stored procedure in the same ordinal position; thus, the fifth parameter listed will provide the value for the fifth argument required by the stored procedure. The parameters must be enclosed in parentheses and separated by commas. Note that the parentheses are required even if there are no parameter values (e.g., if there are no parameter values, you must still use *CALL()*). Strings should be enclosed in single quotes. If the stored procedure has only *OUT* parameters, place host variables or parameter markers within the parentheses.

Rules at a Glance

The *CALL* statement makes it easy to invoke a stored procedure. Simply provide the name of the stored procedure and include any parameters used by the stored procedure, enclosing them within parentheses.

This Oracle example creates a simple stored procedure, and then calls it:

```
CREATE PROCEDURE update_employee_salary
  (emp_id NUMBER, updated_salary NUMBER)
IS
BEGIN
  UPDATE employee SET salary = updated_salary
  WHERE employee_id = emp_id;
END;
CALL update_employee_salary(1517, 95000);
```

Programming Tips and Gotchas

The return status of a called stored procedure can be found, typically, by using *GET DIAGNOSTIC*. *GET DIAGNOSTIC* is not widely supported among the various database platforms, so check the platform documentation for more details.

Many platforms also support an alternative command called *EXECUTE* to perform the same functionality. In some cases you may prefer *EXECUTE* to *CALL*, since the former can be used to execute any kind of prepared SQL, including methods, functions, or batches of SQL code.

MySQL

MySQL supports the ANSI-standard form of the *CALL* statement.

Oracle

Oracle allows the *CALL* statement to invoke standalone stored procedures, functions, and methods, as well as stored procedures and functions contained within a type or package. Following is the Oracle syntax:

```
CALL [schema.][{type_name | package_name}].procedure_name@dblink
  [parameter[, ...])]
  [INTO :variable_name [[INDICATOR] :indicator_name]]
```

where:

CALL [schema.][{type_name | package_name}.]procedure_name@dblink

Calls the named object. You may fully enumerate the object name, including the schema, type, etc., or allow Oracle to assume the current schema and database instance. If the procedure or function resides in another database, simply specify the database via a database link name, shown as *dblink* in the syntax. *dblink* must refer to a previously created database link.

INTO :variable_name

Specifies the name of a previously declared variable that will store the value returned when you call a function. *INTO* is required, and only usable, when calling a function.

INDICATOR :indicator_name

Retains the condition of the host variable (for example, whether the return value is NULL) for functions precompiled in a Pro*C/C++ routine.

The parameters used in an Oracle *CALL* statement *may not* include pseudocolumns or the *VALUE* or *REF* functions. You must use a host variable for any parameter that corresponds to an *OUT* or *IN OUT* argument of the called stored procedure.

PostgreSQL

Not supported.

SQL Server

Not supported. Instead, use the non-ANSI *EXECUTE* statement.

See Also

CREATE/ALTER PROCEDURE

CLOSE CURSOR Statement

The *CLOSE CURSOR* statement closes a server-side cursor previously created with a *DECLARE CURSOR* statement.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported, with variations

SQL2003 Syntax

CLOSE cursor_name

Keywords

cursor_name

Names a cursor previously created with the *DECLARE CURSOR* statement.

Rules at a Glance

The *CLOSE* statement closes a cursor and destroys the cursor result set. All the database platforms release any locks that were held by the cursor, though this is not specified in the ANSI standard. For example:

```
CLOSE author_names_cursor;
```

Programming Tips and Gotchas

You can also close a cursor implicitly using a *COMMIT* statement or, for cursors defined with *WITH HOLD*, using a *ROLLBACK* statement.

MySQL

MySQL supports the ANSI-standard form of the statement.

Oracle

Oracle supports the ANSI-standard form of the statement.

PostgreSQL

PostgreSQL supports the ANSI-standard syntax. PostgreSQL issues an implicit *CLOSE* statement for every open cursor when a transaction is ended with a *COMMIT* or *ROLLBACK* statement.

SQL Server

Microsoft SQL Server supports the ANSI-standard syntax, and an additional *GLOBAL* keyword:

```
CLOSE [GLOBAL] cursor_name
```

where:

GLOBAL

Identifies the previously defined cursor as a global cursor.

In the ANSI-standard behavior, closing a cursor destroys the cursor result set. Locking is a physical feature of each database platform and thus not a part of the ANSI SQL definition. However, all the database platforms covered here drop the locks taken up by the cursor. Another physical implementation detail is that SQL Server does not automatically reallocate memory structures consumed by a cursor to the memory pool. To accomplish such reallocation, you must issue a *DEALLOCATE cursor_name* command.

This example from Microsoft SQL Server opens a cursor and fetches a result set of all employees who have a last name starting with “K”:

```
DECLARE employee_cursor CURSOR FOR
    SELECT lname, fname
    FROM pubs.dbo.employee
    WHERE lname LIKE 'K%'
OPEN employee_cursor
FETCH NEXT FROM employee_cursor
WHILE @@FETCH_STATUS = 0
BEGIN
    FETCH NEXT FROM employee_cursor
END
CLOSE employee_cursor
DEALLOCATE employee_cursor
GO
```

See Also

DECLARE CURSOR
FETCH
OPEN

COMMIT Statement

The *COMMIT* statement explicitly ends an open transaction and makes the changes permanent in the database. Transactions can be opened implicitly as part of an *INSERT*, *UPDATE*, or *DELETE* statement, or opened explicitly with a *START* statement. In either case, an explicitly issued *COMMIT* statement will end the open transaction.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

`COMMIT [WORK] [AND [NO] CHAIN]`

Keywords***COMMIT [WORK]***

Ends the current, open transaction and writes any data manipulated by the transaction to the database. The optional keyword *WORK* is noise and has no effect.

AND [NO] CHAIN

AND CHAIN tells the DBMS to treat the next transaction as if it were a part of the preceding transaction. In effect, the two transactions are separate units of work, but they share a common transaction environment (such as transaction isolation level). Including the optional *NO* keyword tells the DMBS to explicitly use the ANSI default behavior. The *COMMIT* keyword by itself is functionally equivalent to the statement *COMMIT WORK AND NO CHAIN*.

Rules at a Glance

For simple operations, you will execute transactions (that is, SQL code that manipulates or changes data and objects in a database) without explicitly declaring a transaction. However, all transactions are best managed by explicitly closing them with a *COMMIT* statement. Because records and even entire tables can be locked for the duration of a transaction, it is extremely important that transactions are completed as quickly as possible. Thus, manually issuing a *COMMIT* statement with a transaction can help control user concurrency issues and locking problems on the database.

Programming Tips and Gotchas

The most important gotcha to consider is that some database platforms perform automatic and *implicit* transactions, while others require *explicit* transactions. If you assume a platform uses one method of transactions over the other, you may get bitten.

Thus, when moving between database platforms you should follow a standard, preset way of addressing transactions. We recommend always using explicit transactions with *START TRAN*, on database platforms that support it, to begin a transaction, and *COMMIT* or *ROLLBACK* to end a transaction.

In addition to finalizing a single or group of data-manipulation operation(s), *COMMIT* has some interesting effects on other aspects of a transaction. First, it closes any associated open cursors. Second, any temporary table(s) specified with *ON COMMIT DELETE ROWS* (an optional clause of the *CREATE TABLE* statement) are cleared of data. Third, all deferred constraints are checked. If any deferred constraints are violated, the transaction is rolled back. Finally, all locks opened by the transaction are released. Please note that SQL2003 dictates that transactions are *implicitly opened* when one of the following statements is executed:

- *ALTER*
- *CLOSE*
- *COMMIT AND CHAIN*
- *CREATE*
- *DELETE*
- *DROP*
- *FETCH*
- *FREE LOCATOR*
- *GRANT*
- *HOLD LOCATOR*
- *INSERT*
- *OPEN*
- *RETURN*
- *REVOKE*
- *ROLLBACK AND CHAIN*
- *SELECT*
- *START TRANSACTION*
- *UPDATE*

If you did not explicitly open a transaction when you started one of the commands listed above, the standard dictates that the DBMS platform open a transaction for you.

MySQL

MySQL supports *COMMIT* and transactions two transaction-safe engines, InnoDB and NDB Cluster, using this syntax:

`COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]`

The *RELEASE* keyword tells MySQL to disconnect the current client connection once the current transaction is complete. The *NO* keyword, for either *CHAIN* or *RELEASE*, tells MySQL to override the default behavior or explicitly disallows chaining or auto-releasing transactions.

Oracle

Oracle supports the standard, but not the *AND [NO] CHAIN* clause. In addition, Oracle has a couple of extensions to the standard clause:

```
COMMIT [WORK] [ {COMMENT 'text' | FORCE 'text'[, int]} ];
```

where:

COMMENT 'text'

Associates a comment with the current transaction where 'text' is a literal string up to 255 characters long. The text string is stored in the Oracle data dictionary view **DBA_2PC_PENDING** with the transaction ID, in case the transaction rolls back.

FORCE 'text'[, int]

Allows an in-doubt distributed transaction to be manually committed, where 'text' is a literal string that identifies the local or global transaction ID. Transactions can be identified by querying the Oracle data dictionary view **DBA_2PC_PENDING**. The optional *int* parameter is an integer that explicitly assigns a system change number (SCN) to the transaction. Without *int*, the transaction commits using the current SCN.

Issuing a *COMMIT* statement with the *FORCE* clause will commit only the transaction explicitly identified in the *FORCE* clause. It will not affect the current transaction unless it is explicitly identified. The *FORCE* clause is not usable in PL/SQL statements.

The following example commits a transaction while associating a comment with the transaction:

```
COMMIT WORK COMMENT 'In-doubt transaction, Call (949) 555-1234';
```

PostgreSQL

PostgreSQL implements the following syntax:

```
COMMIT [WORK | TRANSACTION];
```

In PostgreSQL, both the *WORK* and *TRANSACTION* keywords are optional. When you issue a *COMMIT*, all open transactions are written to disk and the results of those transactions then become visible to other users. For example:

```
INSERT INTO sales VALUES('7896','JR3435','Oct 28 1997',25,  
'Net 60','BU7832');  
COMMIT WORK;
```

SQL Server

SQL Server does support the *AND [NO] CHAIN* clause. SQL Server supports the keyword *TRANSACTION* as an equivalent to *WORK*, as in the following syntax:

```
COMMIT { [TRANSACTION] [transaction_name] } | [WORK] }
```

Microsoft SQL Server allows the creation of a specific, named transaction using the *START TRAN* statement. The *COMMIT TRANSACTION* syntax allows you to specify an explicit, named transaction to close or to store a transaction name in a variable. Curiously, SQL Server still commits only the most recently opened transaction, despite the name of the transaction that is specified.

When you issue *COMMIT WORK*, SQL Server terminates all open transactions and writes their changes to the database. You *may not* specify a transaction name when using *COMMIT WORK*.

See Also

ROLLBACK
START TRANSACTION

CONNECT Statement

The *CONNECT* statement establishes a connection to the DBMS and to a specific database within the DBMS.

Platform	Command
MySQL	Not supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported, with variations

SQL2003 Syntax

```
CONNECT TO {DEFAULT | {[server_name]} [AS connection_name]
[USER user_name]}
```

Keywords**DEFAULT**

Initiates a default session with the server where the user authorization is the default and the current database is the default. The standard specifies that *CONNECT TO DEFAULT* is issued implicitly if you start a SQL session without first issuing a *CONNECT* statement.

server_name

Establishes a connection to a named server. The *server_name* may be a string literal enclosed in single quotes, or a host variable.

AS connection_name

Establishes a connection with the name *connection_name* shown in single quotes. The optional *connection_name* may be a string literal enclosed in single quotes or a host variable. It is optional only for the first connection attached to a server. All subsequent sessions must include the *AS* clause. This is useful for telling user connections apart when many users (or perhaps even a single user) have many open sessions on a given *server_name*.

USER user_name

Establishes a connection to the named server under the *user_name* specified.

Rules at a Glance

Issue the *CONNECT* statement to open an interactive SQL session with a DBMS. The period between issuing the *CONNECT* and *DISCONNECT* statements is commonly called a *session*. Typically, you will complete all work on a DBMS during an explicitly invoked session.

If you do not specify the server name, the connection name, or the username, the DBMS will provide default values for you. The defaults vary from platform to platform.

To connect to the *houston* server under a specific user ID, you might issue the command:

```
CONNECT TO houston USER pubs_admin
```

If the DBMS requires named connections, you might use the following, alternative syntax:

```
CONNECT TO houston USER pubs_admin
AS pubs_administrative_session;
```

Or, if you want just a simple, short-term connection, you might use the default:

```
CONNECT TO DEFAULT;
```

Programming Tips and Gotchas

If the *CONNECT* statement is invoked before the previous session has been explicitly disconnected, the old session becomes dormant and the new session becomes active. You can then switch between connections using the *SET CONNECTION* statement.



The Oracle tool SQL*Plus uses the *CONNECT* command somewhat differently: to connect a user to a specific schema.

MySQL

Not supported.

Oracle

The *CONNECT* clause allows a database connection under a specific username. Alternatively, a connection can be established for special privileges with *AS SYSOPER* or *AS SYSDBA*. The Oracle syntax is:

```
CONN[ECT] [[username/password] [AS {SYSOPER | SYSDBA}]]
```

where:

CONN[ECT] [username/password]

Establishes a connection to the database instance.

AS {SYSOPER | SYSDBA}

Establishes the connection as one of the two optional system roles.

If another connection is already open, *CONNECT* commits any open transactions, closes the current session, and opens the new one.

Oracle also allows the *CONNECT* statement in its SQL*Plus and *i*SQL*Plus tools.

PostgreSQL

PostgreSQL does not explicitly support the *CONNECT* statement. However, it does support the statement *SPI_CONNECT* under the Server Programming Interface (SPI) and *PG_CONNECT* under the PG/TCL programming package.

SQL Server

SQL Server supports the basic elements of the *CONNECT* statement within Embedded SQL (inside of C++ or Visual Basic programs), as in the following:

```
CONNECT TO [server_name.]database_name [AS connection_name]
USER {login_name[.password] | $integrated}
```

where:

CONNECT TO [server_name.]database_name

Specifies the server and database to which you want to connect. You may leave off the *server_name* if you wish to default to the local server.

AS connection_name

Names the connection with an alphanumeric string of up to 30 characters in length. Symbols are allowed, except for hyphens (-), but the first character must be a letter. *CURRENT* and *ALL* are reserved and may not be used for a *connection_name*. You need only define connection names when making more than one connection.

USER {login_name[,password] | \$integrated}

Connects the session under the *login_name* specified using either the *password* provided or Windows integrated security. The password is optional when supplying a login name.

For example, we can connect to the server named *new_york* as the Windows login *pubs_admin*:

```
CONNECT TO new_york.pubs USER pubs_admin
```

To issue the same command under SQL Server standard security:

```
EXEC SQL CONNECT TO new_york.pubs USER pubs_admin
```

To issue the same command under Windows integrated security:

```
EXEC SQL CONNECT TO new_york.pubs USER $integrated
```

To switch to a different connection, use the *SET CONNECTION* statement.

See Also

[SET CONNECTION](#)

CREATE/ALTER DATABASE Statement

The ANSI standard does not actually contain a *CREATE DATABASE* statement. However, since it is nearly impossible to operate a SQL database without this command, we've added *CREATE DATABASE* here. Almost all database platforms support some version of this command.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

Rules at a Glance

This command creates a new, empty database with a specific name. Most DBMS platforms require the user to possess administrator privileges in order to create a new database. Once the new database is created, you can populate it with database objects (such as tables, views, triggers, and so on) and populate the tables with data.

Depending on the platform, *CREATE DATABASE* may also create corresponding files on the filesystem that contain the data and metadata of the database.

Programming Tips and Gotchas

Since *CREATE DATABASE* is not an ANSI command, it is prone to rather extreme variation in syntax between platforms.

MySQL

In MySQL, *CREATE DATABASE* essentially creates a new directory that holds the database objects:

```
CREATE { DATABASE | SCHEMA } [IF NOT EXISTS] database_name
  [ [DEFAULT] CHARACTER SET character_set ]
  [ [DEFAULT] COLLATE collation_set ]
```

The following is the syntax for MySQL's implementation of the *ALTER DATABASE* statement:

```
ALTER { DATABASE | SCHEMA } database_name
  { [ [DEFAULT] CHARACTER SET character_set ]
    [ [DEFAULT] COLLATE collation_set ] | 
    UPGRADE DATA DIRECTORY NAME }
```

where:

*{CREATE | ALTER} {DATABASE | SCHEMA} *database_name**

Creates a database and directory of *database_name*. The database directory appears under the MySQL data directory. Tables in MySQL then appear as files in the database directory. The *SCHEMA* keyword is synonymous with *DATABASE*.

IF NOT EXISTS

Avoids an error if the database already exists.

*[DEFAULT] CHARACTER SET *character_set**

Optionally defines the default character set used by the database. Refer to the MySQL documentation for a full listing of the available character sets.

*[DEFAULT] COLLATE *collation_set**

Optionally defines the database collation used by the database. Refer to the MySQL documentation for a full listing of the available collations.

UPGRADE DATA DIRECTORY NAME

Updates the name of the directory associated with the database to use the encoding implemented in MySQL v5.1 and later for mapping database names to database directories.

For example, we could create a database called **sales_archive** on a MySQL server. If the database already exists, we want to abort the command without causing an error:

```
CREATE DATABASE IF NOT EXISTS sales_archive
```

Any tables that we create in the **sales_archive** database will now appear as files within the *sales_archive* directory.

Oracle

Oracle provides an extraordinary level of control over database file structures, far beyond merely naming the database and specifying a path for the database files. *CREATE* and *ALTER DATABASE* are very powerful commands in Oracle, and some of the more sophisticated clauses are best used only by experienced DBAs. These commands can be very large and complex—*ALTER DATABASE* alone consumes over 50 pages in the Oracle vendor documentation!

Novices should be aware that *CREATE DATABASE*, when run, erases all data that is already in existence in the specified datafiles. Similarly, all the data in an existing database will be lost.

Following is the syntax to create a new database in Oracle:

```

CREATE DATABASE [database_name]
  {[USER SYS IDENTIFIED BY password | USER SYSTEM IDENTIFIED BY password]}
  [CONTROLFILE REUSE]
  {[LOGFILE definition[, ...]} [MAXLOGFILES int] [[MAXLOGMEMBERS] int]
    [[MAXLOGHISTORY] int] [{ARCHIVELOG | NOARCHIVELOG}] [FORCE LOGGING]}
  [MAXDATAFILES int]
  [MAXINSTANCES int]
  [CHARACTER SET charset]
  [NATIONAL CHARACTER SET charset]
  [EXTENT MANAGEMENT {DICTIONARY | LOCAL
    [ {AUTOALLOCATE | UNIFORM [SIZE int [K | M]]} ]}]
  [DATAFILE definition[, ...]]
  [SYSAUX DATAFILE definition[, ...]]
  [DEFAULT TABLESPACE tablespace_name
    [DATAFILE file_definition]
    EXTENT MANAGEMENT {DICTIONARY |
      LOCAL {AUTOALLOCATE | UNIFORM [SIZE int [K | M]]}}]
  [ [{BIGFILE | SMALLFILE}] DEFAULT TEMPORARY TABLESPACE tablespace_name
    [TEMPFILE file_definition]
    EXTENT MANAGEMENT {DICTIONARY |
      LOCAL {AUTOALLOCATE | UNIFORM [SIZE int [K | M]]}}]
  [ [{BIGFILE | SMALLFILE}] UNDO TABLESPACE tablespace_name
    [DATAFILE temp_datafile_definition] ]
  [SET TIME_ZONE = '{ {+ | -} hh:mi | time_zone_region }' ]
  [SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE]

```

Following is the syntax to alter an existing database:

```

ALTER DATABASE [database_name]
  [{ARCHIVELOG | NOARCHIVELOG} | 
   {MOUNT [{STANDBY | CLONE} DATABASE] | OPEN {READ ONLY | READ WRITE}
     [RESETLOGS | NORESETLOGS] | [UPGRADE | DOWNGRADE]} | 
   {ACTIVATE {PHYSICAL | LOGICAL} STANDBY DATABASE [FINISH APPLY]
     [SKIP {STANDBY LOGFILE}]} | 
   SET STANDBY {DATABASE} TO MAXIMIZE {PROTECTION | AVAILABILITY | 
     PERFORMANCE} | 
   REGISTER {OR REPLACE} {PHYSICAL | LOGICAL} LOGFILE ['file']
     [FOR logminer_session_name] | 
   {COMMIT | PREPARE} TO SWITCHOVER TO
     {{ {PHYSICAL | LOGICAL} PRIMARY | STANDBY} [WITH{OUT}
       SESSION SHUTDOWN] [WAIT | NOWAIT]} | 
     CANCEL} | 
   START LOGICAL STANDBY APPLY [IMMEDIATE] [NODELAY]
     [{INITIAL int | NEW PRIMARY dblink_name | {FINISH | 
       SKIP FAILED TRANSACTION}}] | 
   {STOP | ABORT} LOGICAL STANDBY APPLY |
   [CONVERT TO {PHYSICAL | SNAPSHOT} STANDBY] |

```

```

{RENAME GLOBAL_NAME TO database[.domain[.domain ...]] |  

  CHARACTER SET character_set |  

  NATIONAL CHARACTER SET character_set |  

  DEFAULT TABLESPACE tablespace_name |  

  DEFAULT TEMPORARY TABLESPACE {GROUP int | tablespace_name} |  

  {DISABLE BLOCK CHANGE TRACKING | ENABLE BLOCK CHANGE TRACKING [USING  

    FILE 'file'] [REUSE]} |  

  FLASHBACK {ON | OFF} |  

  SET TIME_ZONE = '{ {+ | -} hh:mi | time_zone_region }' |  

  SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE} |  

{ENABLE | DISABLE} { [PUBLIC] THREAD int | INSTANCE 'instance_name' } |  

{GUARD {ALL | STANDBY | NONE}} |  

{CREATE DATAFILE 'file'[, ...] [AS {NEW | file_definition[, ...]}] |  

  {DATAFILE 'file' | TEMPFILE 'file'[, ...]  

    {ONLINE | OFFLINE [FOR DROP | RESIZE int [K | M]] |  

      END BACKUP | AUTOEXTEND {OFF | ON [NEXT int [K | M]]} [MAXSIZE  

        [UNLIMITED | int [K | M]]]} |  

    {TEMPFILE 'file' | TEMPFILE 'file'[, ...]  

      {ONLINE | OFFLINE | DROP [INCLUDING DATAFILES] |  

        RESIZE int [K | M] | AUTOEXTEND {OFF | ON [NEXT int [K | M]]}  

        [MAXSIZE [UNLIMITED | int [K | M]]]} |  

      RENAME FILE 'file'[, ...] TO 'new_file_name'[, ...]} |  

    {[{NO] FORCE LOGGING} | [{NO]ARCHIVELOG [MANUAL]} |  

      {ADD | DROP] SUPPLEMENTAL LOG DATA [(ALL | PRIMARY KEY | UNIQUE |  

        FOREIGN KEY | FOR PROCEDURAL REPLICATION)[, ...]] COLUMNS |  

      [ADD | DROP] [STANDBY] LOGFILE  

        {{[THREAD int | INSTANCE 'instance_name']} {[GROUP int |  

          logfile_name[, ...]} [SIZE int [K | M]] | [REUSE] |  

          [MEMBER] 'file' [REUSE][, ...] TO logfile_name[, ...]} |  

      ADD LOGFILE MEMBER 'file' [REUSE][, ...] TO {[GROUP int |  

        logfile_name[, ...]}]} |  

      DROP [STANDBY] LOGFILE {MEMBER 'file' | {[GROUP int | logfile_name  

        [, ...]} } |  

      CLEAR [UNARCHIVED] LOGFILE {[GROUP int | logfile_name[, ...]}[, ...]  

        [UNRECOVERABLE DATAFILE]} |  

{CREATE [LOGICAL | PHYSICAL] STANDBY CONTROLFILE AS 'file' [REUSE] |  

  BACKUP CONTROLFILE TO  

    {'file' [REUSE] | TRACE [AS 'file' [REUSE]]} [ {RESETLOGS |  

    NORESETLOGS} ]} |  

{RECOVER  

  {[AUTOMATIC [FROM 'location']]} |  

  {[STANDBY] DATABASE  

    {[UNTIL {CANCEL | TIME date | CHANGE int}]} |  

    USING BACKUP CONTROLFILE} |  

  {{[STANDBY] {TABLESPACE tablespace_name[, ...] | DATAFILE  

    'file'[, ...]} [UNTIL {CONSISTENT WITH] CONTROLFILE]} |  

    TABLESPACE tablespace_name[, ...] | DATAFILE 'file'[, ...]} |  

    LOGFILE filename[, ...]} [{TEST | ALLOW int CORRUPTION | [NO]PARALLEL  

    int}]} |  

  CONTINUE [DEFAULT] |  

  CANCEL}

```

```
{MANAGED STANDBY DATABASE
  {[USING CURRENT LOGFILE]
  [DISCONNECT [FROM SESSION]]
  [NODELAY]
  [UNTIL CHANGE int]
  [FINISH]
  [CANCEL]} |
  TO LOGICAL STANDBY {database_name | KEEP IDENTITY}}
  {{BEGIN | END} BACKUP}
```

The syntax elements in Oracle are as follows. First, for *CREATE DATABASE*:

{*CREATE* | *ALTER*} *DATABASE* [*database_name*]

Creates or alters a database with the name *database_name*. The database name can be up to 8 bytes in length and may not contain European or Asian characters. You can omit the database name and allow Oracle to create the name for you, but beware that the names Oracle creates can be counterintuitive.

USER SYS IDENTIFIED BY *password* | *USER SYSTEM IDENTIFIED BY* *password*

Specifies passwords for the *SYS* and *SYSTEM* users. You may specify neither or both of these clauses, but not just one of them.

CONTROLFILE REUSE

Causes existing control files to be reused, enabling you to specify existing files in the *CONTROL_FILES* parameter in *INIT.ORA*. Oracle will then overwrite any information those files may contain. This clause is normally used when recreating a database. Consequently, you probably don't want to use this clause in conjunction with *MAXLOGFILES*, *MAXLOGMEMBER*, *MAXLOGHISTORY*, *MAXDATAFILES*, or *MAXINSTANCES*.

LOGFILE definition

Specifies one or more logfiles for the database. You may define multiple files all with the same size and characteristics, using the *file* parameter, or you may define multiple files each with its own size and characteristics. The entire logfile definition syntax is rather ponderous, but it offers a great deal of control:

```
LOGFILE { ('file'[, ...]) [SIZE int [K | M]]
  [GROUP int] [REUSE] }[, ...]
```

LOGFILE ('*file*'[, ...])

Defines one or more files that will act as redo logfiles; *file* is both the filename and the path. Any files defined in the *CREATE DATABASE* statement are assigned to redo log thread number 1. When specifying multiple redo logfiles, each filename should be enclosed in single quotes and separated from the other names by commas. The entire list should be enclosed in parentheses.

SIZE *int* [K | M]

Specifies the size of the redo logfile in bytes as an integer value, *int*. Alternatively, you may define the redo logfile in larger units than bytes by appending a *K* (for kilobytes) or an *M* (for megabytes).

GROUP *int*

Defines the integer ID, *int*, of the redo logfile group. The value may be from 1 to the value of the *MAXLOGFILES* clause. An Oracle database must have at least two redo logfile groups. Oracle will create a redo logfile group for you, with a default size of 100 MB, if you omit this group ID.

REUSE

Reuses an existing redo logfile.

MAXLOGFILES *int*

Sets the maximum number of logfiles, *int*, available to the database being created. The minimum, maximum, and default values for this clause are OS-dependent.

MAXLOGMEMBERS *int*

Sets the maximum number of members (i.e., copies) for a redo logfile group. The minimum value is 1, while the maximum and default values for this clause are OS-dependent.

MAXLOGHISTORY *int*

Sets the maximum number of archived redo logfiles available to a Real Application Cluster (RAC). You can use the *MAXLOGHISTORY* clause only when Oracle is in ARCHIVELOG mode on a RAC. The minimum value is 0, while the maximum and default values for this clause are OS-dependent.

ARCHIVELOG | NOARCHIVELOG

Defines how redo logs operate. When used with *ALTER DATABASE*, specifying one of these allows the current setting to be changed. *ARCHIVELOG* saves data stored in the redo log(s) to an archiving file, providing for media recoverability. Conversely, *NOARCHIVELOG* allows a redo log to be reused without archiving the contents. Both options provide recoverability, although *NOARCHIVELOG* (the default) does *not* provide media recovery.

FORCE LOGGING

Places all instances of the database into *FORCE LOGGING* mode, in which all changes to the database are logged, except for changes to temporary tablespaces and segments. This setting takes precedence over any tablespace- or object-level settings.

MAXDATAFILES *int*

Sets the initial number of datafiles, *int*, available to the database being created. Note that the *INIT.ORA* setting, *DB_FILES*, also limits the number of datafiles accessible to the database instance.

MAXINSTANCES *int*

Sets the maximum number of instances, *int*, that may mount and open the database being created. The minimum value is 1, while the maximum and default values for this clause are OS-dependent.

CHARACTER SET *charset*

Controls the language character set in which the data is stored. The value for *charset* cannot be *AL16UTF16*. The default value is OS-dependent.

NATIONAL CHARACTER SET *charset*

Controls the national language character set for data stored in *NCHAR*, *NCLOB*, and *NVARCHAR2* columns. The value for *charset* must be either *AL16UTF16* (the default) or *UTF8*.

EXTENT MANAGEMENT {DICTIONARY | LOCAL}

Creates a locally managed *SYSTEM* tablespace (otherwise, the *SYSTEM* tablespace will be dictionary-managed). This clause requires a default temporary tablespace. If you omit the *DATAFILE* clause you can also omit the default temporary tablespace, because Oracle will create them both for you.

DATAFILE definition

Specifies one or more datafiles for the database. (All these datafiles become part of the *SYSTEM* tablespace.) You may repeat filenames to define multiple files with the same size and characteristics. Alternately, you may repeat the entire *DATAFILE* clause, with each occurrence defining one or more files with the same size and characteristics. The entire datafile definition syntax is rather large, but it offers a great deal of control:

```
DATAFILE { ('file1'[, ...]) [GROUP int] [SIZE int [K | M]] [REUSE]
  [AUTOEXTEND {OFF | ON [NEXT int [K | M]]}]
  [MAXSIZE [UNLIMITED | int [K | M]]] } [,...]
```

DATAFILE ('file1'[, ...])

Defines one or more files that will act as the datafile(s), where *file1* is both the filename and the path. For multiple files, each filename should be enclosed in single quotes and separated from the others by a comma. The entire list should be enclosed in parentheses.

GROUP int

Defines the integer ID, *int*, of the datafile group. The value may be from 1 to the value of the *MAXLOGFILES* clause. An Oracle database must have at least two datafile groups. Oracle will create them for you, at 100 MB each, if you omit this clause.

SIZE int [K | M]

Specifies the size of the datafile in bytes as an integer value, *int*. Alternately, you may define the datafile in large units by appending a *K* (for kilobytes) or an *M* (for megabytes).

REUSE

Reuses an existing datafile.

AUTOEXTEND {OFF | ON [NEXT int [K | M]]}

Enables (ON) the automatic extension of new or existing datafiles or tempfiles (but does not redo logfiles). *NEXT* specifies the next increment of space allocated to the file in bytes, kilobytes (*K*), or megabytes (*M*) when more space is needed.

MAXSIZE [UNLIMITED | int [K | M]]

Specifies the maximum disk space allowed for automatic extension of the file. *UNLIMITED* allows the file to grow without an upper limit (except, of course, the total capacity of the drive). Otherwise, you may define the maximum size limit as an integer, *int*, in bytes (the default), kilobytes with the keyword *K*, or megabytes with the keyword *M*.

SYSAUX DATAFILE definition

Specifies one or more datafiles for the *SYSAUX* tablespace. By default, Oracle creates and manages the *SYSTEM* and *SYSAUX* tablespaces automatically. You must use this clause if you have specified a datafile for the *SYSTEM* tablespace. If you omit the *SYSAUX* clause when using Oracle-managed files, Oracle will create the *SYSAUX* tablespace as an online, permanent, locally managed tablespace with a single datafile of 100 MB, using automatic segment-space management and logging.

BIGFILE | SMALLFILE

Specifies the default file type of a subsequently created tablespace. *BIGFILE* indicates that the tablespace will contain a single datafile or tempfile of up to 8 exabytes (8 million terabytes) in size, while *SMALLFILE* indicates the tablespace is a traditional Oracle tablespace. The default, when omitted, is *SMALLFILE*.

DEFAULT TABLESPACE tablespace_name

Specifies a default permanent tablespace for the database for all non-SYSTEM users. When this clause is omitted, the *SYSTEM* tablespace is the default permanent tablespace for non-SYSTEM users.

DEFAULT TEMPORARY TABLESPACE tablespace_name [TEMPFILE file_definition]

Defines the name and location of the default temporary tablespace for the database. Users who are not explicitly assigned to a temporary tablespace will operate in this one. If you don't create a default temporary tablespace, Oracle uses the *SYSTEM* tablespace. Under *ALTER DATABASE*, this clause allows you to change the default temporary tablespace.

TEMPFILE file_definition

The tempfile definition is optional when the *DB_CREATE_FILE_DEST* *INIT.ORA* parameter is set. Otherwise, you'll have to define the tempfile yourself. The *TEMPFILE* definition syntax is identical to the *DATAFILE* definition syntax described earlier in this section.

EXTENT MANAGEMENT {DICTIONARY | LOCAL {AUTOALLOCATE | UNIFORM [SIZE int [K | M]}}}

Defines the way in which the *SYSTEM* tablespace is managed. When this clause is omitted, the *SYSTEM* tablespace is dictionary-managed. Once created as a locally managed tablespace, it cannot be converted back to a dictionary-managed tablespace, nor can any new dictionary-managed tablespaces be created in the database.

DICTIONARY

Specifies that the Oracle data dictionary manages the tablespace. This is the default. The *AUTOALLOCATE* and *UNIFORM* subclauses are not used with this clause.

LOCAL

Declares that the tablespace is locally managed. This clause is optional, since all temporary tablespaces have locally managed extents by default. Use of this clause requires a default temporary tablespace. If you do not manually create one, Oracle will automatically create one called *TEMP*, of 10 MB in size, with *AUTOEXTEND* disabled.

AUTOALLOCATE

Specifies that new extents will be allocated as needed by the locally managed tablespace.

UNIFORM [SIZE int [K | M]]

Specifies that all extents of the tablespace are the same size (*UNIFORM*), in bytes, as *int*. The *SIZE* clause allows you to configure the size of the extents to your liking in bytes (the default), in kilobytes using the keyword *K*, or in megabytes using the keyword *M*. The default is 1M.

UNDO TABLESPACE *tablespace_name* [DATAFILE *temp_datafile_definition*]

Defines the name and location for undo data, creating a tablespace named *tablespace_name*, but only if you have set the *UNDO_MANAGEMENT INIT.ORA* parameter to *AUTO*. If you don't use this clause, Oracle manages undo space via rollback segments. (You may also set the *INIT.ORA* parameter to *UNDO_TABLESPACE*. If you do so, the value of the parameter and the *tablespace_name* used here must be identical.)

DATAFILE *temp_datafile_definition*

Creates and assigns the datafile, as you have defined it, to the undo tablespace. Refer to the earlier description of *DATAFILE* for the full syntax of this clause. This clause is required if you have not specified a value for the *INIT.ORA* parameter *DB_CREATE_FILE_DEST*.

SET TIME_ZONE = '{ {+ | -} hh:mi | *time_zone_region* }'

Sets the time zone for the database, either by specifying a delta from Greenwich Mean Time (now called Coordinated Universal Time) or by specifying a time zone region. (For a list of time zone regions, query the *tzname* column of the *v\$timezone_names* view.) If you do not use this clause, Oracle defaults to the operating-system time zone.

SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE

Sets all tablespaces created by the current *CREATE DATABASE* or *ALTER DATABASE* statement as either *BIGFILE* or *SMALLFILE*. When creating databases, this clause also applies to the *SYSTEM* and *SYSAUX* tablespaces.

And for *ALTER DATABASE*:

MOUNT [{STANDBY | CLONE}] DATABASE

Mounts a database for users to access. The *STANDBY* keyword mounts a physical standby database, enabling it to receive archived redo logs from the primary instance. The *CLONE* keyword mounts a clone database. This clause cannot be used with *OPEN*.

OPEN {READ WRITE | READ ONLY} [{RESETLOGS | NORESETLOGS} [{UPGRADE | DOWNGRADE}]

Opens the database separately from the mounting process. (Mount the database first.) *READ WRITE* opens the database in read/write mode, allowing users to generate redo logs. *READ ONLY* allows reads of but disallows changes to redo logs. *RESETLOGS* discards all redo information not applied during recovery and sets the log sequence number to 1. *NORESETLOGS* retains the logs in their present condition. The optional *UPGRADE* and *DOWNGRADE* clauses tell Oracle to dynamically modify the system parameters as required for database upgrade or downgrade, respectively. The default is *OPEN READWRITE NORESETLOGS*.

ACTIVATE {PHYSICAL | LOGICAL} STANDBY DATABASE [{FINISH APPLY} | {SKIP | STANDBY LOGFILE}]

Promotes a standby database to the primary database. You can optionally specify a *PHYSICAL* standby, the default, or a *LOGICAL* standby. *FINISH APPLY* initiates the application of the remaining redo log, bringing the logical standby database to the same state as the primary database. When it's finished, the database completes the switchover from the logical standby to the primary database. Use the *SKIP* clause to immediately promote a physical standby and discard any data still unapplied by the *RECOVER MANAGED STANDBY DATABASE FINISH* statement. The clause *STANDBY LOGFILE* is noise.

SET STANDBY [DATABASE] TO MAXIMIZE {PROTECTION | AVAILABILITY | PERFORMANCE}

Sets the level of protection for data in the primary database. The old terms *PROTECTED* and *UNPROTECTED* equate to *MAXIMIZE PROTECTION* and *MAXIMIZE PERFORMANCE*, respectively.

PROTECTION

Provides the highest level of data protection, but has the greater overhead and negatively impacts availability. This setting commits transactions only after all data necessary for recovery has been physically written in at least one physical standby database that uses the *SYNC* log transport mode.

AVAILABILITY

Provides the second highest level of data protection, but the highest level of availability. This setting commits transactions only after all data necessary for recovery has been physically written in at least one physical or logical standby database that uses the *SYNC* log transport mode.

PERFORMANCE

Provides the highest level of performance, but compromises data protection and availability. This setting commits transactions before all data necessary for recovery has been physically written to a standby database.

REGISTER [OR REPLACE] [PHYSICAL | LOGICAL] LOGFILE ['file']

Manually registers redo logfiles from a failed primary server when issued from a standby server. The logfile may optionally be declared as *PHYSICAL* or *LOGICAL*. The *OR REPLACE* clause allows updates to details of an existing archivelog entry.

FOR logminer_session_name

Registers the logfile with a single, specific LogMiner session in an Oracle Streams environment.

{COMMIT | PREPARE} TO SWITCHOVER TO {[PHYSICAL | LOGICAL] PRIMARY | STANDBY}

Performs a graceful switchover, moving the current primary database to standby status and promoting a standby database to primary. (In a RAC environment, all instances other than the current instance have to be shut down.) To gracefully switch over, you should issue the command twice (the first time to prepare the primary and standby databases to begin exchanging logfiles in advance of the switchover) using *PREPARE TO SWITCHOVER*. To demote the primary database and switch over to the standby, use *COMMIT TO SWITCHOVER*. The *PHYSICAL* clause puts the primary database into physical standby mode. The *LOGICAL* clause puts the primary database into logical standby mode. However, you must then issue an *ALTER DATABASE START LOGICAL STANDBY APPLY* statement.

[WITH[OUT] SESSION SHUTDOWN] [WAIT | NOWAIT]

WITH SESSION SHUTDOWN closes any open application sessions and rolls back any uncommitted transactions during a switchover of physical databases (but not logical ones). *WITHOUT SESSION SHUTDOWN*, the default, causes the *COMMIT TO SWITCHOVER* statement to fail if it encounters any open application sessions. *WAIT* returns control to the console after completion of the *SWITCHOVER* command, while *NOWAIT* returns control before the command completes.

START LOGICAL STANDBY APPLY [IMMEDIATE] [NODELAY] [{INITIAL *int* | NEW PRIMARY *dblink_name*} | {FINISH | SKIP FAILED TRANSACTION}]

Starts to apply redo logs to the logical standby database. *IMMEDIATE* tells the Oracle LogMiner to read the redo data in the standby redo logfiles. *NODELAY* tells Oracle to ignore a delay for the apply, such as when the primary database is unavailable or disabled. *INITIAL* is used the first time you apply logs to the standby database. *NEW PRIMARY* is required after a switchover has completed or after a standby database has processed all redo logs and another standby is promoted to primary. Use *SKIP FAILED TRANSACTION* to skip the last transaction and to restart the apply. Use *FINISH* to apply the data in the redo logs if the primary database is disabled.

[STOP | ABORT] LOGICAL STANDBY APPLY

Stops the application of redo logs to a logical standby server. *STOP* performs an orderly stop, while *ABORT* performs an immediate stop.

CONVERT TO {PHYSICAL | SNAPSHOT} STANDBY

Converts a primary database or snapshot standby database into a physical standby database (for *PHYSICAL*), or converts a physical standby database into a snapshot standby database (for *SNAPSHOT*).

RENAME GLOBAL_NAME TO *database*[.domain[.domain ...]]

Changes the global name of the database, where *database* is the new name of up to 8 bytes in length. The optional *domain* specifications identify the database's location in the network. This does not propagate database name changes to any dependent objects like synonyms, stored procedures, etc.

{DISABLE | ENABLE} BLOCK CHANGE TRACKING [USING FILE '*file*' | REUSE]

Tells Oracle to stop or start tracking the physical locations of all database updates, respectively, and maintain the information in a special file called the *block change tracking file*. Oracle will automatically create the file as defined by the *DB_CREATE_FILE_DEST* parameter, unless you add the *USING FILE '*file*'* clause, where '*file*' is the path and name of the file. *REUSE* tells Oracle to overwrite an existing block change tracking file of the same name as '*file*'. The *USING* and *REUSE* subclauses are allowed only with the *ENABLE BLOCK* clause.

FLASHBACK {ON | OFF}

Places the database into or out of *FLASHBACK* mode, respectively. When in flashback mode, an Oracle database automatically creates and maintains flashback database logs in the flash recovery area. When *OFF*, the flashback database logs are deleted and unavailable.

SET TIME ZONE

Specifies the time zone for the server. Refer to the description of the *SET TIME ZONE* statement in the discussion of the *CREATE TABLE* syntax for more information.

{ENABLE | DISABLE} { [PUBLIC] THREAD *int* | INSTANCE '*instance_name*'}

In RAC environments, you can *ENABLE* or *DISABLE* a redo log thread by number (*int*). You may optionally specify an *instance_name* to enable or disable a thread mapped to a specific database instance of an Oracle RAC environment. The *instance_name* may be up to 80 characters long. The *PUBLIC* keyword makes the thread available to any instance. When omitted, the thread is available only when explicitly requested. To enable a thread, the thread must have at least two redo logfile groups. To disable a thread, the database must be open but not mounted by an instance using the thread.

GUARD {ALL | STANDBY | NONE}

Protects the data in a database from changes. *ALL* prevents users other than *SYS* from making any changes. *STANDBY* prevents all users other than *SYS* from making changes in a logical standby. *NONE* provides normal security for the database.

CREATE DATAFILE 'file'[, ...] [AS {NEW | file_definition}]

Creates a new, empty datafile, replacing an existing one. The value '*file*' identifies a file (either by filename or file number) that was lost or damaged without a backup. *AS NEW* creates a new file in the default filesystem using an Oracle-supplied name. *AS file_definition* allows you to specify a filename and sizing details, as defined under “*TEMPFILE file_definition*” section in the preceding list.

DATAFILE 'file' | TEMPFILE 'file'[, ...] {ONLINE | OFFLINE [FOR DROP] | RESIZE int [K | M] | END BACKUP | AUTOEXTEND {OFF | ON [NEXT int [K | M]]} [MAXSIZE [UNLIMITED | int [K | M]]]}

Changes the attributes, such as the size, of one or more existing datafiles or tempfiles. You may alter one or more files in a comma-delimited list, identified in the value '*file*' by filename or file number. Do not mix datafile and tempfile declarations; only one or the other should appear in this clause at a time.

ONLINE

Sets the file online.

OFFLINE [FOR DROP]

Sets the file offline, allowing media recovery. *FOR DROP* is required to take a file offline in *NOARCHIVELOG* mode, but it does not actually destroy the file. It is ignored in *ARCHIVELOG* mode.

RESIZE int [K | M]

Sets a new size for an existing datafile or tempfile.

END BACKUP

Described later in the main list, under *END BACKUP*. Used only with the *DATAFILE* clause.

AUTOEXTEND {OFF | ON [NEXT int [K | M]]} [MAXSIZE [UNLIMITED | int [K | M]]]

Described in the preceding list, under the *DATAFILE* definition.

DROP [INCLUDING DATAFILES]

Drops not only the tempfile, but all datafiles on the filesystem associated with the tempfile. Oracle also adds an entry to the alert log for each file that is erased. Used only with the *TEMPFILE* clause.

RENAME FILE 'file'[, ...] TO 'new_file_name'[, ...]

Renames a datafile, tempfile, or redo logfile member from the old name, *file*, to the *new_file_name*. You can rename multiple files at once by specifying multiple old and new filenames, separated by commas. This command does not rename files at the operating-system level. Rather, it specifies new names that Oracle will use to open the files. You need to rename at the operating-system level yourself.

[NO] FORCE LOGGING

Puts the database into force logging mode (*FORCE LOGGING*) or takes it out of force logging mode (*NO FORCE LOGGING*). In the former, Oracle logs all changes to the database except in temporary tablespaces or segments. This database-level *FORCE LOGGING* setting supersedes all tablespace-level declarations regarding force logging mode.

[NO]ARCHIVELOG [MANUAL]

Tells Oracle to create redo logfiles, but that the user will handle the archiving of the redo logfiles explicitly. This is used only with the *ALTER DATABASE* statement and *only* for backward compatibility for users with older tape backup systems. When this clause is omitted, Oracle defaults the redo logfile destination to the *LOG_ARCHIVE_DEST_n* initialization parameter.

[ADD | DROP] SUPPLEMENTAL LOG DATA [(ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY | FOR PROCEDURAL REPLICATION), ...] COLUMNS

ADD places additional column data into the log stream whenever an update is executed. It also enables *minimal supplemental logging*, which ensures that LogMiner can support chained rows and special storage arrangements such as clustered tables. Supplemental logging is disabled by default. You can add the clauses *PRIMARY KEY COLUMNS*, *UNIQUE KEY COLUMNS*, *FOREIGN KEY COLUMNS*, or *ALL* (to get all three options) if you need to enable full referential integrity via foreign keys in another database, such as a logical standby, or *FOR PROCEDURAL REPLICATION* for logging PL/SQL calls. In either case, Oracle places either the primary key columns, the unique key columns (or, if none exist, a combination of columns that uniquely identify each row), the foreign key columns, or all three into the log. *DROP* tells Oracle to suspend supplemental logging.

[ADD | DROP] [STANDBY] LOGFILE {[THREAD int | INSTANCE 'instance_name']} {[GROUP int | logfile_name[, ...]]} [SIZE int [K | M]] | [REUSE] | [MEMBER] 'file' [REUSE][, ...]

ADD includes one or more primary or standby redo logfile groups to the specified instance. *THREAD* assigns the added files to a specific thread number (*int*) on a RAC. When omitted, the default is the thread assigned to the current instance. *GROUP* assigns the redo logfile groups to a specific group within the thread. *MEMBER* adds the specified '*file*' (or files in a comma-delimited list) to an existing redo logfile group. *REUSE* is needed if the file already exists. *DROP LOGFILE MEMBER* drops one or more redo logfile members, after issuing an *ALTER SYSTEM SWITCH LOGFILE* statement.

CLEAR [UNARCHIVED] LOGFILE {[GROUP int | logfile_name[, ...]]}[, ...] [UNRECOVERABLE DATAFILE]

Reinitializes one or more (in a comma-delimited list) specified online redo logs. *UNRECOVERABLE DATAFILE* is required when any datafile is offline and the database is in *ARCHIVELOG* mode.

CREATE {LOGICAL | PHYSICAL} STANDBY CONTROLFILE AS 'file' [REUSE]

Creates a control file that maintains a logical or physical standby database. *REUSE* is needed if the file already exists.

BACKUP CONTROLFILE TO {'file' [REUSE] | TRACE [AS 'file' [REUSE]]} [{RESETLOGS | NORESETLOGS}]

Backs up the current control file of an open or mounted database. *TO 'file'* identifies a full path and filename for the control file. *TO TRACE* writes SQL statements to recreate the control file to a trace file. *TO TRACE AS 'file'* writes all the SQL statements to a standard file rather than a trace file. *REUSE* is needed if the file already exists. *RESETLOGS* initializes the trace file with the statement *ALTER DATABASE OPEN RESETLOGS* and is valid only when online logs are unavailable. *NORESETLOGS* initializes the trace file with the statement *ALTER DATABASE OPEN NORESETLOGS* and is valid only when online logs are available.

RECOVER

Controls media recovery for the database, standby database, tablespace, or file. In Oracle, the *ALTER TABLE* command is one of the primary means of recovering a damaged or disabled database, file, or tablespace. Use *RECOVER* when the database is mounted (in exclusive mode), the files and tablespaces involved are not in use (offline), and the database is in either an open or closed state. The entire database can be recovered only when it is closed, but specific files or tablespaces can be recovered in a database that is open.

AUTOMATIC [FROM 'location']

Tells Oracle to automatically generate the name of the next archived redo logfile necessary for continued operation during recovery. Oracle will prompt you if it cannot find the next file. The *FROM 'location'* clause tells Oracle where to find the archived redo logfile group. The following subclauses may be applied to an automatically recovered database.

STANDBY

Specifies that the database or tablespace to recover is a standby type.

DATABASE {[UNTIL {CANCEL | TIME date | CHANGE int}] | USING BACKUP CONTROLFILE}

Tells Oracle to recover the entire database. The *UNTIL* keyword tells Oracle to continue recovery until *ALTER DATABASE...RECOVER CANCEL (CANCEL)* is issued, until a specified time in the format YYYY-MM-DD:HH24:MI:SS is reached (*TIME*), or until a specific system change number is reached (*CHANGE int*, where *int* is the number). The clause *USING BACKUP CONTROLFILE* enables use of the backup, rather than current, control file.

[STANDBY] [TABLESPACE *tablespace_name*[, ...] | DATAFILE '*file*'[, ...]]

Recovers one or more specific tablespaces or datafiles, respectively. You may specify more than one tablespace or datafile using a comma-delimited list. You may also recover a datafile by datafile number, rather than by name. The tablespace may be in normal or standby mode. The standby tablespace or datafile is reconstructed using archived redo logfiles copied from the primary database and a control file.

UNTIL [CONSISTENT WITH] CONTROLFILE

Tells Oracle to recover an older standby tablespace or datafile by using the current standby control file. *CONSISTENT WITH* are noise words.

LOGFILE *filename*[, ...]

Continues media recovery by applying one or more redo logfiles that you specify in a comma-delimited list.

TEST | ALLOW *int* CORRUPTION | [NO]PARALLEL *int*

TEST performs a trial recovery, allowing you to foresee any problems. *ALLOW int CORRUPTION* tells how many corrupt blocks (*int*) to tolerate before causing recovery to abort. *int* must be 1 for a real recovery, but may be any number you choose when paired with *TEST*. *[NO]PARALLEL* determines whether parallel recovery of media is used. *NOPARALLEL* is the default and enforces serial reading of the media. *PARALLEL* with no *int* value tells Oracle to choose the degree of parallelism to apply. Specifying *int* declares the degree of parallelism to apply.

CONTINUE [DEFAULT]

Determines whether multi-instance recovery continues after interruption. **CONTINUE DEFAULT** is the same as **RECOVER AUTOMATIC**, but it does not result in a prompt for a filename.

CANCEL

Cancels a managed recovery operation at the next archived log boundary, if it was started with the **USING CANCEL** clause.

MANAGED STANDBY DATABASE

Specifies managed physical standby recovery mode on an active component of a standby database. This command is used for media recovery only and not to construct a new database, using the following parameters:

USING CURRENT LOGFILE

Invokes *real time apply*, which allows recovery of redos from standby online logs as they are being filled, without first requiring that they be archived by the standby database.

DISCONNECT [FROM SESSION]

Causes the managed redo process to occur in the background, leaving the current process available for other tasks. **FROM SESSION** are noise words.

DISCONNECT is incompatible with **TIMEOUT**.

NODELAY

Overrides the *DELAY* attribute **LOG_ARCHIVE_DEST_n** parameter on the primary database. When omitted, Oracle delays the application of the archived redo log according to the attribute.

UNTIL CHANGE int

Conducts a managed recovery up to (but not including) the specified system change number *int*.

FINISH

Recovers all available online redo log files immediately in preparation of the standby assuming the primary database role. The **FINISH** clause is known as a *terminal recovery* and should be used only in the event of a failure of the primary database.

CANCEL

Stops application of redo applies immediately and returns control as soon as the redo apply stops.

TO LOGICAL STANDBY {database_name | KEEP IDENTITY}

Converts the physical standby database into a logical standby database. The *database_name* identifies the new logical standby database. The **KEEP IDENTITY** subclause tells Oracle that the logical standby is used for a rolling upgrade and is not usable as a general-purpose logical standby database.

{BEGIN | END} BACKUP

Controls the online backup mode for any datafiles. **BEGIN** places all datafiles into online backup mode. The database must be mounted and open, in archivelog mode with media recovery enabled. (Note that while the database is in online backup mode the instance cannot be shut down and individual tablespaces cannot be backed up, taken offline, or made read-only.) **END** takes all datafiles currently in online backup mode out of that mode. The database must be mounted but need not be open.

After that long discussion of specific syntax, it's important to establish some Oracle basics.

Oracle allows the use of *primary* and *standby* databases. A *primary* database is a mounted and open database accessible to users. The primary database regularly and frequently ships its redo logs to a *standby* database where they are recovered, thus making the *standby* database a very up-to-date copy of the primary.

Unique to the Oracle environment is the *INIT.ORA* file, which specifies the database name and a variety of other options that you can use when creating and starting up the database. You should always define startup parameters, such as the name of any control files, in the *INIT.ORA* file to identify the control files; otherwise, the database will not start. Starting in Oracle 9.1, you can use binary parameter files rather than *INIT.ORA* files.

When a group of logfiles is listed, they are usually shown in parentheses. The parentheses aren't needed when creating a group with only one member, but this is seldom done. Here's an example using a parenthetical list of logfiles:

```
CREATE DATABASE publications
  LOGFILE ('/s01/oradata/loga01','/s01/oradata/loga02') SIZE 5M
  DATAFILE;
```

That example creates a database called **publications** with an explicitly defined logfile clause and an automatically created datafile. The following example of an Oracle *CREATE DATABASE* command is much more sophisticated:

```
CREATE DATABASE sales_reporting
  CONTROLFILE REUSE
  LOGFILE
    GROUP 1 ('diskE:log01.log', 'diskF:log01.log') SIZE 15M,
    GROUP 2 ('diskE:log02.log', 'diskF:log02.log') SIZE 15M
  MAXLOGFILES 5
  MAXLOGHISTORY 100
  MAXDATAFILES 10
  MAXINSTANCES 2
  ARCHIVELOG
  CHARACTER SET AL32UTF8
  NATIONAL CHARACTER SET AL16UTF16
  DATAFILE
    'diskE:sales_rpt1.dbf' AUTOEXTEND ON,
    'diskF:sales_rpt2.dbf' AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED
  DEFAULT TEMPORARY TABLESPACE temp_tbldspc
  UNDO TABLESPACE undo_tbldspc
  SET TIME_ZONE = '-08:00';
```

This example defines logfiles and datafiles, as well as all appropriate character sets. We also define a few characteristics for the database, such as the use of *ARCHIVELOG* mode and *CONTROLFILE REUSE* mode, the time zone, the maximum number of instances and datafiles, etc. This example also assumes that the *INIT.ORA* parameter for *DB_CREATE_FILE_DEST* has already been set. Thus, we don't have to define file definitions for the *DEFAULT TEMPORARY TABLESPACE* and *UNDO TABLESPACE* clauses.

When issued by a user with *SYSDBA* privileges, this statement creates a database and makes it available to users in either exclusive or parallel mode, as defined by the value of the *CLUSTER_DATABASE* initialization parameter. Any data that exists in

predefined datafiles is erased. You will usually want to create tablespaces and rollback segments for the database. (Refer to the vendor documentation for details on the platform-specific commands *CREATE TABLESPACE* and *CREATE ROLLBACK SEGMENT*.)

Oracle has tightened up security around default database user accounts. Many default database user accounts are now locked and expired during initial installation. Only *SYS*, *SYSTEM*, *SCOTT*, *DBSNMP*, *OUTLN*, *AURORA\$JIS\$UTILITY\$*, *AURORA\$ORB\$UNAUTHENTICATED*, and *OSE\$HTTP\$ADMIN* are the same in 11g as they were in earlier versions. You *must* manually unlock and assign a new password to all locked accounts, as well as assign a password to *SYS* and *SYSTEM*, during the initial installation.

In the next example, we add more logfiles to the current database, and then add a datafile:

```
ALTER DATABASE ADD LOGFILE GROUP 3
  ('diskf: log3.sales_arch_log','diskg:log3.sales_arch_log')
SIZE 50M;
ALTER DATABASE sales_archive
CREATE DATAFILE 'diskF:sales_rpt4.dbf'
AUTOEXTEND ON NEXT 25M MAXSIZE UNLIMITED;
```

We can set a new default temporary tablespace, as shown in the next example:

```
ALTER DATABASE DEFAULT TEMPORARY TABLESPACE sales_tbl_spc_2;
```

Next, we'll perform a simple full database recovery:

```
ALTER DATABASE sales_archive RECOVER AUTOMATIC DATABASE;
```

In the next example, we perform a more elaborate partial database recovery:

```
ALTER DATABASE RECOVER STANDBY DATAFILE 'diskF:sales_rpt4.dbf'
UNTIL CONTROLFILE;
```

Now, we'll perform a simple recovery of a standby database in managed standby recovery mode:

```
ALTER DATABASE RECOVER sales_archive MANAGED STANDBY DATABASE;
```

In the following example, we gracefully switch over from a primary database to a logical standby, and promote the logical standby to primary:

```
-- Demotes the current primary to logical standby database.
ALTER DATABASE COMMIT TO SWITCHOVER TO LOGICAL STANDBY;
-- Applies changes to the new standby.
ALTER DATABASE START LOGICAL STANDBY APPLY;
-- Promotes the current standby to primary database.
ALTER DATABASE COMMIT TO SWITCHOVER TO PRIMARY;
```

PostgreSQL

PostgreSQL's implementation of the *CREATE DATABASE* command creates a database and a file location for the datafiles:

```
CREATE DATABASE database_name [ WITH ]
  [OWNER [=] database_owner]
  [TEMPLATE [=] tmp_name]
  [ENCODING [=] enc_value]
  [TABLESPACE [=] tablespace_name]
  [CONNECTION LIMIT [=] int]
```

PostgreSQL's syntax for *ALTER DATABASE* is:

```
ALTER DATABASE database_name [ WITH ]
  [CONNECTION LIMIT int]
  [OWNER TO new_database_owner]
  [RENAME TO new_database_name]
  [RESET parameter]
  [SET parameter {TO | =} {value | DEFAULT}]
```

where:

WITH

Is an optional keyword to further define the details of the database.

OWNER [=] *database_owner*

Specifies the name of the database owner if it is different from the name of the user executing the statement.

TEMPLATE [=] *tmp_name*

Names a template to use for creating the new database. You can omit this clause to accept the default template describing the location of the database files (or use the clause *TEMPLATE = DEFAULT*). The default is to copy the standard system database **template1**. You can get a pristine database (one that contains only required database objects) by specifying *TEMPLATE = template0*.

ENCODING [=] *enc_value*

Specifies the multibyte encoding method to use in the new database using either a string literal (such as 'SQL_ASCII'), an integer encoding number, or *DEFAULT* for the default encoding.

TABLESPACE [=] *tablespace_name*

Specifies the name of the tablespace associated with the database.

CONNECTION LIMIT [=] *int*

Specifies how many concurrent connections to the database are allowed. A value of -1 means no limit.

RENAME TO *new_database_name*

Assigns a new name to the database.

RESET *parameter* | SET *parameter* {TO | =} {value | DEFAULT}

Assigns (using *SET*) or reassigns (using *RESET*) a *value* for a *parameter* defining the database.

For example, to create the database **sales_revenue** in the */home/teddy/private_db* directory:

```
CREATE DATABASE sales_revenue
  WITH LOCATION = '/home/teddy/private_db';
```



Be careful using absolute file and pathnames, since they impact security and data integrity issues.

PostgreSQL advises that databases used as templates be treated as read-only. Template databases should not be used for general-purpose "copy database" functionality.



PostgreSQL does not support the ALTER DATABASE command.

SQL Server

SQL Server offers a lot of control over the OS filesystem structures that hold the database and its objects. SQL Server's *CREATE DATABASE* statement syntax looks like this:

```
CREATE DATABASE database_name
[ ON file_definition[, ...] ]
[, FILEGROUP filegroup_name file_definition[, ...] ]
[ LOG ON file_definition[, ...] ]
[ COLLATE collation_name ]
[ FOR { ATTACH [WITH {ENABLE_BROKER | NEW_BROKER | ERROR_BROKER_
CONVERSATIONS}] |
      ATTACH_REBUILD_LOG }
      [WITH {DB_CHAINING {ON | OFF}}[, TRUSTWORTHY {ON | OFF}]] ]
      [ AS SNAPSHOT OF source ] ]
```

Following is the syntax for *ALTER DATABASE*:

```
ALTER DATABASE database_name
{ADD FILE file_definition[, ...] [TO FILEGROUP filegroup_name]
| ADD LOG FILE file_definition[, ...]
| REMOVE FILE file_name
| ADD FILEGROUP filegroup_name
| REMOVE FILEGROUP filegroup_name
| MODIFY FILE file_definition
| MODIFY NAME = new_database_name
| MODIFY FILEGROUP filegroup_name
  {NAME = new_filegroup_name | filegroup_property
    {READONLY | READWRITE | DEFAULT}}
| SET {state_option | cursor_option
  | auto_option | sql_option | recovery_option
  [, ...] [WITH termination_option]
| COLLATE collation_name}
```

Parameter descriptions are as follows:

{*CREATE* | *ALTER*} *DATABASE* *database_name*

Creates a database (or alters an existing database) with the name *database_name*. The name cannot be longer than 128 characters. You should limit the database name to 123 characters when no logical filename is supplied, since SQL Server will create a logical filename by appending a suffix to the database name.

{*ON* | *ADD*} *file_definition*[, ...]

Defines the disk file(s) that store(s) the data components of the database for *CREATE DATABASE*, or adds disk file(s) for *ALTER DATABASE*. *ON* is required for *CREATE DATABASE* only if you wish to provide one or more file definitions.

The syntax for *file_definition* is:

```
{[PRIMARY] ( [NEW][NAME = file_name]
  [, FILENAME = {'os_file_name' | 'filestream_name'}]
  [, SIZE = int [KB | MD | GB | TB]][, MAXSIZE = { int |
  UNLIMITED }][, FILEGROWTH = int][, OFFLINE] )}{, ...]
```

where:

PRIMARY

Defines the *file_definition* as the primary file. Only one primary file is allowed per database. (If you don't define a primary file, SQL Server defaults primary status to the file that it autocreates, in the absence of any user-defined file, or to the first file that you define.) The primary file or group of files (also called a *filegroup*) contains the logical start of the database, all the database system tables, and all other objects not contained in user filegroups.

[NEW]NAME = *file_name*

Provides the logical name of the file defined by the *file_definition* for *CREATE DATABASE*. Use *NEWNAME* for *ALTER DATABASE* to define a new logical name for the file. In either case, the logical name must be unique within the database. This clause is optional when using *FOR ATTACH*.

FILENAME = { 'os_file_name' | 'filestream_name' }

Specifies the operating system path and filename for the file defined by *file_definition*. The file must be in a noncompressed directory on the filesystem.

For raw partitions, specify only the drive letter of the raw partition.

SIZE = *int* [KB | MB | GB | TB]

Sets the size of the file defined by the *file_definition*. This clause is optional, but it defaults to the file size for the primary file of the model database, which is usually very small. Logfiles and secondary datafiles default to a size of 1 MB. The value of *int* defaults to megabytes; however, you can explicitly define the size of the file using the suffixes for kilobyte (KB), megabyte (MB), gigabyte (GB), and terabyte (TB). The size cannot be smaller than 512 KB or the size of the primary file of the model database.

MAXSIZE = { *int* | UNLIMITED }

Defines the maximum size to which the file may grow. Suffixes, as described under the entry for *SIZE*, are allowed. The default, *UNLIMITED*, allows the file to grow until all available disk space is consumed. Not required for files on raw partitions.

FILEGROWTH = *int*

Defines the growth increment for the file each time it grows. Suffixes, as described under the entry for *SIZE*, are allowed. You may also use the percentage (%) suffix to indicate that the file should grow by a percentage of the total disk space currently consumed. If you omit the *FILEGROWTH* clause, the file will grow in 10% increments, but never less than 64 KB. Not required for files on raw partitions.

OFFLINE

Sets the file offline, making all objects in the filegroup inaccessible. This option should only be used when the file is corrupted.

[ADD] LOG {ON | FILE} *file_definition*

Defines the disk file(s) that store(s) the log component of the database for *CREATE DATABASE*, or adds disk file(s) for *ALTER DATABASE*. You can provide one or more *file_definitions* for the transaction logs in a comma-delimited list. Refer to the earlier section under the keyword *ON* for the full syntax of *file_definition*.

REMOVE FILE *file_name*

Removes a file from the database and deletes the physical file. The file must be emptied of all content first.

[ADD] FILEGROUP *filegroup_name* [*file_definition*]

Defines any user filegroups used by the database, and their file definitions. All databases have at least one primary filegroup (though many databases only use the primary filegroup that comes with SQL Server by default). Adding filegroups and then moving files to those filegroups can allow you greater control over disk I/O. (However, we recommend that you do not add filegroups without careful analysis and testing.)

REMOVE FILEGROUP *filegroup_name*

Removes a filegroup from the database and deletes all the files in the filegroup. The files and the filegroup must be empty first.

MODIFY FILE *file_definition*

Changes the definition of a file. This clause is very similar to the [ADD] LOG {ON | FILE} clause. For example: *MODIFY FILE (NAME = file_name, NEWNAME = new_file_name, SIZE = ...)*.

MODIFY NAME = *new_database_name*

Changes the database's name from its current name to *new_database_name*.

MODIFY FILEGROUP *filegroup_name* {NAME = *new_filegroup_name* | *filegroup_property*}

Used with ALTER DATABASE, this clause has two forms. One form allows you to change a filegroup's name, as in *MODIFY FILEGROUP filegroup_name NAME = new_filegroup_name*. The other form allows you to specify a *filegroup_property* for the filegroup, which must be one of the following:

READONLY

Sets the filegroup to read-only and disallows updates to all objects within the filegroup. *READONLY* can only be enabled by users with exclusive database access and cannot be applied to the primary filegroup. You may also use *READ_ONLY*.

READWRITE

Disables the *READONLY* property and allows updates to objects within the filegroup. *READWRITE* can only be enabled by users with exclusive database access. You may also use *READ_WRITE*.

DEFAULT

Sets the filegroup as the default filegroup for the database. All new tables and indexes are assigned to the default filegroup unless explicitly assigned elsewhere. Only one default filegroup is allowed per database. (By default, the CREATE DATABASE statement sets the primary filegroup as the default filegroup.)

SET {*state_option* | *cursor_option* | *auto_option* | *sql_option* | *recovery_option*}[, ...]

Controls a wide variety of behaviors for the database. These are discussed in the rules and information later in this section.

WITH *termination_option*

Used after the *SET* clause, *WITH* sets the rollback behavior for incomplete transactions whenever the database is in transition. When this clause is omitted, transactions must commit or roll back on their own with the database state changes. There are two *termination_option* settings:

ROLLBACK AFTER *int* [SECONDS] | ROLLBACK IMMEDIATE

Causes the database to roll back in *int* number of seconds, or immediately. *SECONDS* is a noise word and does not change the behavior of the *ROLLBACK AFTER* clause.

NO_WAIT

Causes database state or option changes to fail if a change cannot be completed immediately, without waiting for the current transaction to independently commit or roll back.

COLLATE *collation_name*

Defines or alters the default collation used by the database. *collation_name* can be either a SQL Server collation name or a Windows collation name. By default, all new databases receive the collation of the SQL Server instance. (You can execute the query *SELECT * FROM ::fn_helpcollations()* to see all the collation names available.) To change the collation of a database, you must be the only user in the database, no schema-bound objects that depend on the current collation may exist in the database, and the collation change must not result in the duplication of any object names in the database.

FOR { ATTACH [WITH {ENABLE_BROKER | NEW_BROKER | ERROR_BROKER_CONVERSATIONS}] | ATTACH_REBUILD_LOG }

Places the database in special startup mode. *FOR ATTACH* creates the database from a set of pre-existing operating system files (almost always database files created previously). Because of this, the new database must have the same code page and sort order as the previous database. You only need the *file_definition* of the first primary file or those files that have a different path from the last time the database was attached. The *FOR ATTACH_REBUILD_LOG* clause specifies that the database is created by attaching an existing set of OS files, rebuilding the log in the process in case any logfiles are missing. In general, you should use the *sp_attach_db* system stored procedure instead of the *CREATE DATABASE FOR ATTACH* statement unless you need to specify more than 16 *file_definitions*.

Service Broker options may be specified when using the *FOR ATTACH* clause:

ENABLE_BROKER

Specifies that Service Broker is enabled for the database.

NEW_BROKER

Creates a new **service_broker_guid** and ends all conversation endpoints with a cleanup.

ERROR_BROKER_CONVERSATIONS

Terminates all Service Broker conversations with an error indicating that a database has been attached or restored. The broker is disabled during the operation and then re-enabled afterward.

WITH {DB_CHAINING {ON | OFF}}[, TRUSTWORTHY {ON | OFF}]

Specifies that the database can be involved in a cross-database ownership chain (with *DB_CHAINING ON*). When omitted, the default is *OFF*, which disallows cross-database ownership chains. Setting *TRUSTWORTHY ON* specifies that database routines (such as views, functions, or procedures) that use an impersonation context can access resources outside of the database. When omitted, the default is *OFF*, which disallows accessing external resources from within a routine running in an impersonation context. *TRUSTWORTHY* is set *OFF* whenever a database is attached. *DB_CHAINING* and *TRUSTWORTHY* cannot be set *ON* for master, model, or *tempdb* databases.

AS SNAPSHOT OF *source*

Declares that the database being created is a snapshot of the *source* database. Both source and snapshot must exist on the same instance of SQL Server.

The *CREATE DATABASE* command should be issued from the *master* system database. You can, in fact, issue the command *CREATE DATABASE database_name*, with no other clauses, to get a very small, default database.

SQL Server uses *files*, formerly called *devices*, to act as a repository for databases. Files are grouped into one or more filegroups, with at least a *PRIMARY* filegroup assigned to each database. A file is a predefined block of space created on the disk structure. A database may be stored on one or more files or filegroups. SQL Server also allows the transaction log to be placed in a separate location from the database using the *LOG ON* clause. These functions allow sophisticated file planning for optimal control of disk I/O. For example, we can create a database called **sales_report** with a data and transaction logfile:

```
USE master
GO
CREATE DATABASE sales_report
ON
( NAME = sales_rpt_data, FILENAME =
  'c:\mssql\data\salerptdata.mdf',
  SIZE = 100, MAXSIZE = 500, FILEGROWTH = 25 )
LOG ON
( NAME = 'sales_rpt_log',
  FILENAME = 'c:\mssql\log\salesrptlog.ldf',
  SIZE = 25MB, MAXSIZE = 50MB,
  FILEGROWTH = 5MB )
GO
```

When a database is created, all objects in the model database are copied into the new database. All of the empty space within the file or files defined for the database is then initialized (i.e., emptied out), which means that creating a new and very large database can take a while, especially on a slow disk.

A database always has at least a primary datafile and a transaction logfile, but it may also have secondary files for both the data and log components of the database. SQL Server uses default filename extensions: *.mdf* for primary datafiles, *.ndf* for secondary files, and *.ldf* for transaction logfiles. The following example creates a database called **sales_archive** with several very large files that are grouped into a couple of filegroups:

```
USE master
GO
CREATE DATABASE sales_archive
ON
PRIMARY (NAME = sales_arch1, FILENAME = 'c:\mssql\data\archdata1.mdf',
  SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
( NAME = sales_arch2,
  FILENAME = 'c:\mssql\data\archdata2.ndf',
  SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
( NAME = sales_arch3,
  FILENAME = 'c:\mssql\data\archdat3.ndf',
  SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB)
```

```

FILEGROUP sale_rpt_grp1
  (NAME = sale_rpt_grp1_1_data,
  FILENAME = 'c:\mssql\data\SG1Fi1dt.ndf',
  SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
  (NAME = sale_rpt_grp1_1_data,
  FILENAME = 'c:\mssql\data\SG1Fi2dt.ndf',
  SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
FILEGROUP sale_rpt_grp2
  (NAME = sale_rpt_grp2_1_data, FILENAME = 'c:\mssql\data\SRG21dt.ndf',
  SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
  (NAME = sale_rpt_grp2_2_data, FILENAME = 'c:\mssql\data\SRG22dt.ndf',
  SIZE = 100GB, MAXSIZE = 200GB, FILEGROWTH = 20GB),
LOG ON
  (NAME = sales_archlog1,
  FILENAME = 'd:\mssql\log\archlog1.ldf',
  SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%),
  (NAME = sales_archlog2,
  FILENAME = 'd:\mssql\log\archlog2.ldf',
  SIZE = 100GB, MAXSIZE = UNLIMITED, FILEGROWTH = 25%)
GO

```

The *FOR ATTACH* clause is commonly used for situations like a salesperson traveling with a database on a CD. This clause tells SQL Server that the database is attached from an existing operating system file structure, such as a DVD-ROM, CD-ROM, or portable hard drive. When using *FOR ATTACH*, the new database inherits all the objects and data of the parent database, not the model database.

The following examples show how to change the name of a database, file, or filegroup:

```

-- Rename a database
ALTER DATABASE sales_archive MODIFY NAME = sales_history
GO
-- Rename a file
ALTER DATABASE sales_archive MODIFY FILE
NAME = sales_arch1,
NEWNAME = sales_hist1
GO
-- Rename a filegroup
ALTER DATABASE sales_archive MODIFY FILEGROUP
sale_rpt_grp1
NAME = sales_hist_grp1
GO

```

There may be times when you want to add new free space to a database, especially if you have not enabled it to auto-grow:

```

USE master
GO
ALTER DATABASE sales_report ADD FILE
( NAME = sales_rpt_added01, FILENAME = 'c:\mssql\data\salerptadded01.mdf',
  SIZE = 50MB, MAXSIZE = 250MB, FILEGROWTH = 25MB )
GO

```

When you alter a database, you can set many behavior options on the database. State options (shown as *state_option* in the earlier syntax diagram) control how users access the database. Following is a list of valid state options:

SINGLE_USER | RESTRICTED_USER | MULTI_USER

Sets the number and type of users with access to the database. *SINGLE_USER* mode allows only one user to access the database at a time. *RESTRICTED_USER* mode allows access only to members of the system roles *db_owner*, *dbcreator*, or *sysadmin*. *MULTI_USER*, the default, allows concurrent database access from all users who have permission.

OFFLINE | ONLINE

Sets the database to offline (unavailable) or online (available).

READ_ONLY | READ_WRITE

Sets the database to *READ_ONLY* mode, where no modifications are allowed, or to *READ_WRITE* mode, where data modifications are allowed. *READ_ONLY* databases can be very fast for query-intensive operations, since almost no locking is needed.

Cursor options control default behavior for cursors in the database. In the *ALTER DATABASE* syntax shown earlier, you can replace *cursor_option* with any of the following:

CURSOR_CLOSE_ON_COMMIT { ON | OFF }

When set to *ON*, any open cursors are closed when a transaction commits or rolls back. When set to *OFF*, any open cursors remain open when transactions are committed and close when a transaction rolls, back unless the cursor is *INSENSITIVE* or *STATIC*.

CURSOR_DEFAULT { LOCAL | GLOBAL }

Sets the default scope of all cursors in the database to either *LOCAL* or *GLOBAL*. (See later in this chapter for more details.)

In the *SET* clause, *auto_option* controls the automatic file-handling behaviors of the database. The following are valid replacements for *auto_option*:

AUTO_CLOSE { ON | OFF }

When set to *ON*, the database automatically shuts down cleanly and frees all resources when the last user exits. When set to *OFF*, the database remains open when the last user exits. The default is *OFF*.

AUTO_CREATE_STATISTICS { ON | OFF }

When set to *ON*, statistics are automatically created when SQL Server notices they are missing during query optimization. When set to *OFF*, statistics are not created during optimization. The default is *ON*.

AUTO_SHRINK { ON | OFF }

When set to *ON*, the database files may automatically shrink (the database periodically looks for an opportunity to shrink files, though the time is not always predictable). When set to *OFF*, files will shrink only when you explicitly and manually shrink them. The default is *OFF*.

AUTO_UPDATE_STATISTICS { ON | OFF }

When set to *ON*, out-of-date statistics are reassessed during query optimization.

When set to *OFF*, statistics are reassessed only by explicitly and manually recompiling them using the SQL Server command *UPDATE STATISTICS*.

The *sql_options* clause controls the ANSI compatibility of the database. You can use the standalone SQL Server command *SET ANSI_DEFAULTS ON* to enable all the ANSI SQL92 behaviors at one time, rather than using the individual statements below. In the *SET* clause, you can replace *sql_option* with any of the following:

ANSI_NULL_DEFAULT { ON | OFF }

When set to *ON*, the *CREATE TABLE* statement causes columns with no nullability setting to default to *NULL*. When set to *OFF*, the nullability of a column defaults to *NOT NULL*. The default is *OFF*.

ANSI_NULLS { ON | OFF }

When set to *ON*, comparisons to *NULL* yield *UNKNOWN*. When set to *OFF*, comparisons to *NULL* yield *NULL* if both non-UNICODE values are *NULL*. The default is *OFF*.

ANSI_PADDING { ON | OFF }

When set to *ON*, strings are padded to the same length for insert or comparison operations on *VARCHAR* and *VARBINARY* columns. When set to *OFF*, strings are not padded. The default is *ON*. (We recommend that you *do not* change this!)

ANSI_WARNINGS { ON | OFF }

When set to *ON*, the database warns when problems like “divide by zero” or “*NULL* in aggregates” occur. When set to *OFF*, these warnings are not raised. The default is *OFF*.

ARITHABORT { ON | OFF }

When set to *ON*, divide-by-zero and overflow errors cause a query or Transact-SQL batch to terminate and roll back any open transactions. When set to *OFF*, a warning is raised but processing continues. The default is *ON*. (We recommend that you *do not* change this!)

CONCAT_NULL_YIELDS_NULL { ON | OFF }

When set to *ON*, returns a *NULL* when a *NULL* is concatenated to a string. When set to *OFF*, *NULLs* are treated as empty strings when concatenated to a string. The default is *OFF*.

NUMERIC_ROUNDABORT { ON | OFF }

When set to *ON*, an error is raised when a numeric expression loses precision. When set to *OFF*, losses of precision result in rounding of the result from the numeric expression. The default is *OFF*.

QUOTED_IDENTIFIER { ON | OFF }

When set to *ON*, double quotation marks identify an object identifier that contains special characters or is a reserved word (e.g., a table named **SELECT**). When set to *OFF*, identifiers may not contain special characters or reserved words, and all occurrences of double quotation marks signify a literal string value. The default is *OFF*.

RECURSIVE_TRIGGERS { ON | OFF }

When set to *ON*, triggers can fire recursively. That is, the actions taken by one trigger may cause another trigger to fire, and so on. When set to *OFF*, triggers cannot cause other triggers to fire. The default is *OFF*.

Recovery options control the recovery model used by the database. Use any of the following in place of *recovery_option* in the *ALTER DATABASE* syntax:

RECOVERY { FULL | BULK_LOGGED | SIMPLE }

When set to *FULL*, database backups and transaction logs provide full recoverability even for bulk operations like *SELECT...INTO*, *CREATE INDEX*, etc. *FULL* is the default for SQL Server 2000 Standard Edition and Enterprise Edition. *FULL* provides the most recoverability, even from a catastrophic media failure, but uses more space. When set to *BULK_LOGGED*, logging for bulk operations

is minimized. Space is saved and fewer I/O operations are incurred, but risk of data loss is greater than under *FULL*. When set to *SIMPLE*, the database can only be recovered to the last full or differential backup. *SIMPLE* is the default for SQL Server 2000 Desktop Edition and Personal Edition.

TORN_PAGE_DETECTION { ON | OFF }

When set to *ON*, SQL Server can detect incomplete I/O operations at the disk level by checking each 512-byte sector per 8K database page. (Torn pages are usually detected in recovery.) The default is *ON*.

For example, we may want to change some behavior settings for the **sales_report** database without actually changing the underlying file structure:

```
ALTER DATABASE sales_report SET ONLINE, READ_ONLY,
AUTO_CREATE_STATISTICS ON
GO
```

This statement puts the database online and in read-only mode. It also sets the *AUTO_CREATE_STATISTICS* behavior to *ON*.

See Also

[CREATE SCHEMA](#)
[DROP](#)

CREATE/ALTER FUNCTION/PROCEDURE Statements

The *CREATE FUNCTION* and *CREATE PROCEDURE* statements are very similar in syntax and coding (as are the respective *ALTER* statements).

The *CREATE PROCEDURE* statement creates a *stored procedure*, which takes input arguments and performs conditional processing against various objects in the database. According to the ANSI standard, a stored procedure returns no result set (though it may return a value in an *OUTPUT* parameter). For example, you might use a stored procedure to perform all the processes that close an accounting cycle.

The *CREATE FUNCTION* statement creates a *user-defined function* (UDF), which takes input arguments and returns a single value output in the same way as a system-supplied function like *CAST()* or *UPPER()*. These functions, once created, can be called in queries and data-manipulation operations, such as *INSERT*, *UPDATE*, and the *WHERE* clause of *DELETE* statements. Refer to Chapter 4 for descriptions of built-in SQL functions and their individual vendor implementations.

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with variations

SQL2003 Syntax

Use the following syntax to create a stored procedure or function:

```
CREATE {PROCEDURE | FUNCTION} object_name
( {[IN | OUT | INOUT] [parameter_name] datatype [AS LOCATOR] [RESULT]}
[, ...] )
```

```

[ RETURNS datatype [AS LOCATOR]
  [CAST FROM datatype [AS LOCATOR]] ]
[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
[PARAMETER STYLE {SQL | GENERAL}]
[SPECIFIC specific_name]
[DETERMINISTIC | NOT DETERMINISTIC]
[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
[DYNAMIC RESULT SETS int]
[STATIC DISPATCH] code_block

```

Use the following syntax to alter a pre-existing UDF or stored procedure:

```

ALTER {PROCEDURE | FUNCTION} object_name
  [( {parameter_name datatype }[, ...] )]
[NAME new_object_name]
[LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}]
[PARAMETER STYLE {SQL | GENERAL}]
[NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
[RETURN NULL ON NULL INPUT | CALL ON NULL INPUT]
[DYNAMIC RESULT SETS int]
[CASCADE | RESTRICT]

```

Keywords

CREATE {PROCEDURE | FUNCTION} *object_name*

Creates a new stored procedure or user-defined function with the name *object_name*. A user-defined function returns a value, whereas a stored procedure (in ANSI SQL) does not.

({[IN | OUT | INOUT] [*parameter_name datatype* [AS LOCATOR] [RESULT]] [, ...]})

Declares one or more parameters to be passed into a stored procedure in a comma-delimited list enclosed in parentheses. Parameters used with a procedure may pass a value *IN*, *OUT*, or both in and out via *INOUT*.

The syntax for the parameter declaration is:

```

[ {IN | OUT | INOUT} ] parameter_name1 datatype,
[ {IN | OUT | INOUT} ] parameter_name2 datatype, [...]

```

When providing the optional *parameter_name*, make sure the name is unique within the stored procedure. The optional AS LOCATOR subclause is used to validate an external routine with a RETURNS parameter that is a *BLOB*, *CLOB*, *NCLOB*, *ARRAY*, or user-defined type. If you need to change the datatype of a RETURNS parameter on the fly, use the *CAST* clause (refer to the section on the function *CAST* in Chapter 4): for example, *RETURNS VARCHAR(12) CAST FROM DATE*. When used with *ALTER*, this clause adds parameters to a pre-existing stored procedure. Refer to Chapter 2 for details on datatypes.

RETURNS *datatype* [AS LOCATOR] [CAST FROM *datatype* [AS LOCATOR]]

Declares the datatype of the result returned by a function. (This clause is used only in the *CREATE FUNCTION* statement and is not used in stored procedures.) The key purpose of a user-defined function is to return a value.

LANGUAGE {ADA | C | FORTRAN | MUMPS | PASCAL | PLI | SQL}

Declares the language in which the function is written. Most database platforms do not support all of these languages and may support several not mentioned, such as Java. When omitted, the default is *SQL*. When used with *ALTER*, this clause changes the existing *LANGUAGE* value to the value that you declare.

PARAMETER STYLE {SQL | GENERAL}

Indicates, for external routines only, whether certain implicit and automatic parameters are passed explicitly, with the options *(SQL)* or not *(GENERAL)*. (The difference between SQL style and GENERAL style is that SQL style automatically passes SQL parameters, such as indicators, while GENERAL style does not automatically pass parameters.) The default is *PARAMETER STYLE SQL*. When used with *ALTER*, this clause changes the existing *PARAMETER STYLE* value to the value that you declare.

SPECIFIC *specific_name*

Uniquely identifies the function. Generally used with user-defined types.

DETERMINISTIC | NOT DETERMINISTIC

States the nature of values returned by a function. (This clause is used only in *CREATE* and *ALTER FUNCTION* statements.) *DETERMINISTIC* functions always return the same value when given the same parameter values. *NOT DETERMINISTIC* functions may return variable results when given the same parameter values. For example, *CURRENT_TIME* is not deterministic because it returns a constantly advancing value corresponding to the time.

NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA

Specifies, in conjunction with the *LANGUAGE* setting, the type of SQL contained in the user-defined function. When used with *ALTER*, this clause changes the existing *SQL* style value to the value that you declare.

NO SQL

Indicates that no SQL statements of any type are in the function. Used with a non-SQL *LANGUAGE* setting such as *LANGUAGE ADA...* *CONTAINS NO SQL*.

CONTAINS SQL

Indicates that SQL statements other than read or modify statements are in the function. This is the default.

READS SQL DATA

Indicates that the function contains *SELECT* or *FETCH* statements.

MODIFIES SQL DATA

Indicates that the function contains *INSERT*, *UPDATE*, or *DELETE* statements.

RETURN NULL ON NULL INPUT | CALL ON NULL INPUT

These options are for use with a host *LANGUAGE* that cannot support NULLs. The *RETURNS NULL ON NULL INPUT* setting causes the function to immediately return a NULL value if it is passed a NULL value. The *CALL ON NULL INPUT* setting causes the function to handle NULLs according to standard rules: for example, returning *UNKNOWN* when a comparison of two NULL values occurs. (This clause is used in the *CREATE* and *ALTER PROCEDURE* and *FUNCTION* statements.) When used with *ALTER*, this clause changes the existing NULL-style value to the value that you declare.

DYNAMIC RESULT SETS *int*

Declares that a certain number of cursors (*int*) can be opened by the stored procedure and that those cursors are visible after returning from the procedure. When omitted, the default is *DYNAMIC RESULT SETS 0*. (This clause is not used in *CREATE FUNCTION* statements.) When used with *ALTER*, this clause changes the existing *DYNAMIC RESULT SETS* value to the value that you declare.

STATIC DISPATCH

Returns the static values of a user-defined type or ARRAY datatype. Required for non-SQL functions that contain parameters that use user-defined types or ARRAYS. (This clause is not used in *CREATE PROCEDURE* statements.) This clause must be the last clause in the function or procedure declaration before the *code_block*.

code_block

Declares the procedural statements that handle all processing within the user-defined function or stored procedure. This is the most important, and usually largest, part of a function or procedure. We assume you're interested in a user-defined function of *LANGUAGE SQL*, so note that the *code_block* may not contain SQL-Schema statements, SQL-Transaction statements, or SQL-Connection statements.

While we assume that you're interested in SQL-based UDFs and stored procedures (this is a SQL book, after all), you can declare that the code block is derived externally. The syntax for external *code_blocks* is:

```
EXTERNAL [NAME external_routine_name] [PARAMETER STYLE
{SQL | GENERAL}] [TRANSFORM GROUP group_name]
```

where:

EXTERNAL [NAME *external_routine_name*]

Defines an external routine and assigns a name to it. When omitted, the unqualified routine name is used.

PARAMETER STYLE {SQL | GENERAL}

Same as for *CREATE PROCEDURE*.

TRANSFORM GROUP *group_name*

Transforms values between user-defined types and host variables in a user-defined function or a stored procedure. When omitted, the default is *TRANSFORM GROUP DEFAULT*.

NAME *new_object_name*

Declares the new name to use for a previously defined UDF or stored procedure. This clause is used only with *ALTER FUNCTION* and *ALTER PROCEDURE* statements.

CASCADE | RESTRICT

Allows you to cause changes to *CASCADE* down to all dependent UDFs or stored procedures, or to *RESTRICT* a change from happening if there are dependent objects. We strongly recommend that you *do not* issue an *ALTER* statement against UDFs or stored procedures that have dependent objects. This clause is used only with *ALTER FUNCTION* and *ALTER PROCEDURE* statements.

Rules at a Glance

With a user-defined function, you declare the input arguments and the single output argument that the function passes back out. You can then call the user-defined function just as you would any other function: for example, in *SELECT* statements, *INSERT* statements, or the *WHERE* clauses of *DELETE* statements.



User-defined functions and stored procedures, when referred to generically, are called *routines*.

With a stored procedure, you declare the input arguments that go into the stored procedure and the output arguments that come out from it. You invoke a stored procedure using the *CALL* statement.

The content of the *code_block* must conform to the rules of whatever procedural language the database platform supports. Some platforms do not have their own internal procedural languages, requiring you to use *EXTERNAL code_block* constructs.

For example, you might want to build a user-defined function on Microsoft SQL Server that returns the first and last name of a person as a single string:

```
CREATE FUNCTION formatted_name (@fname VARCHAR(30),
@lname VARCHAR(30) )
RETURNS VARCHAR(60)
AS
BEGIN
    DECLARE @full_name VARCHAR(60)
    SET @full_name = @fname + ' ' + @lname
    RETURN @full_name
END
```

You could then use this user-defined function just as you would any other function:

```
SELECT formatted_name(au_fname, au_lname) AS name, au_id AS id
FROM authors
```

The *SELECT* statement in this example will return a result set with the two columns **name** and **id**.



The ANSI statement *CREATE METHOD* is supported only by IBM's UDB DB2 platform at this time. A *method* is essentially a special-purpose user-defined function and follows the same syntax as that outlined here.

Stored procedures behave similarly. The Microsoft SQL Server stored procedure in the following example generates a unique 22-digit value (based on elements of the system date and time) and returns it to the calling process:

```
-- A Microsoft SQL Server stored procedure
CREATE PROCEDURE get_next_nbr
    @next_nbr CHAR(22) OUTPUT
AS
BEGIN
    DECLARE @random_nbr INT
    SELECT @random_nbr = RAND() * 1000000
    SELECT @next_nbr =
        RIGHT('000000' + CAST(ROUND(RAND(@random_nbr)*1000000,0))AS
        CHAR(6), 6) +
        RIGHT('0000' + CAST(DATEPART (yy, GETDATE() ) AS CHAR(4)), 2) +
        RIGHT('000' + CAST(DATEPART (dy, GETDATE() ) AS CHAR(3)), 3) +
        RIGHT('00' + CAST(DATEPART (hh, GETDATE() ) AS CHAR(2)), 2) +
        RIGHT('00' + CAST(DATEPART (mi, GETDATE() ) AS CHAR(2)), 2) +
        RIGHT('00' + CAST(DATEPART (ss, GETDATE() ) AS CHAR(2)), 2) +
        RIGHT('000' + CAST(DATEPART (ms, GETDATE() ) AS CHAR(3)), 3)
END
GO
```

In this next (and final) ANSI SQL2003 example, we change the name of an existing stored procedure:

```
ALTER PROCEDURE get_next_nbr
  NAME 'get_next_ID'
  RESTRICT;
```

Programming Tips and Gotchas

A key advantage of a stored procedure or function is the fact that it is *precompiled*, meaning that once it's been created, its query plans are already stored in the database. Precompiled routines can often (though not always) be cached in database memory to provide an additional boost in performance. A stored procedure or user-defined function can perform many statements with a single communication to the server, thus reducing network traffic.

Implementations of user-defined functions and stored procedures vary widely by platform. Some database platforms do not support internal *code_block* content. On these platforms, you can only write an external *code_block*. The following sections outline the variations and the capabilities of each platform.

If you execute an *ALTER PROCEDURE* or *FUNCTION* statement, dependent objects may become invalid after a change to an object on which they depend. Be careful to check all dependencies when altering UDFs or stored procedures on which other UDFs or stored procedures may depend.

MySQL

MySQL supports both the *ALTER* and *CREATE FUNCTION* statements, as well as the *ALTER* and *CREATE PROCEDURE* statements.

The *CREATE* syntax for functions and procedures follows:

```
CREATE
  [DEFINER = {user | CURRENT_USER}]
  { FUNCTION | PROCEDURE } [database_name.]routine_name
    ( [{IN | OUT | INOUT}] [parameter[, ...] )
  [RETURNS type]
  [NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
  [COMMENT 'string']
  routine_body;
```

where:

CREATE { FUNCTION | PROCEDURE } [database_name.]routine_name

Creates an external function or procedure with a *routine_name* of not more than 64 characters. The module is stored in the *proctable* in the MySQL database.

DEFINER

Assigns a *user*, in the format '*user_name@host_name*', as the owner of the routine. When omitted, the *current_user* is the default.

([{IN | OUT | INOUT}] parameter[, ...])

Defines one or more parameters for the routine. All function parameters must be *IN* parameters, but procedure parameters may be any of the three types. When this clause is omitted on a procedure, parameters are *IN* by default.

RETURNS type

Returns a value called *type* of any valid MySQL datatype. Only for use with *CREATE* or *ALTER FUNCTION*, where it is mandatory.

`COMMENT 'my_string'`

Adds a comment to the routine. The comment(s) may be displayed using the `SHOW CREATE PROCEDURE` and `SHOW CREATE FUNCTION` statements.

`routine_body`

Contains one or more valid SQL statements. Multiple SQL statements should be nested within `BEGIN` and `END`. The `routine_body` can also contain procedural language such as declarations, loops, and other control structures.

Procedures and functions may contain DDL statements such as `CREATE`, `ALTER`, or `DROP`. Procedures, but not functions, may contain transaction control statements such as `COMMIT` and `ROLLBACK`. Functions may not use statements that perform explicit or implicit rollbacks or commits, nor may functions contain statements that return uncontrolled result sets, such as a `SELECT` statement without the `INTO` clause. Neither procedures nor functions may contain the command `LOAD DATA INFILE`.

Once implemented, a MySQL function may be called just like any built-in function, such as `ABS()` or `SOUNDEX()`. Procedures, on the other hand, are invoked using the `CALL` statement.

The implementation of `CREATE FUNCTION` in MySQL has long supported user-defined functions through an implementation that depends on external procedural code in C/C++ under an operating system that supports dynamic loading. The C/C++ program is named in the `shared_program_library_name` option. The function may be compiled either directly into the MySQL server, making it permanently available, or as a dynamically callable program. For example, the code behind the UDF created in the following statement might be found on a Unix server:

```
CREATE FUNCTION find_radius RETURNS INT SONAME "radius.so";
```

Oracle

Oracle supports `ALTER` and `CREATE` for both the `FUNCTION` and `PROCEDURE` object types. (You may also wish to learn about Oracle packages, which can also be used to create UDFs and stored procedures. Check Oracle's documentation.) Oracle's `CREATE PROCEDURE` syntax is as follows:

```
CREATE [OR REPLACE] {FUNCTION | PROCEDURE} [schema.]object_name
[(parameter1 [IN | OUT | IN OUT] [NOCOPY] datatype[, ...])]
RETURN datatype
[DETERMINISTIC] [AUTHID {CURRENT_USER | DEFINER}] [IS | AS] [EXTERNAL]
[PARALLEL_ENABLE [( PARTITION prtn_name BY {ANY | {HASH | RANGE}
    (column[, ...]}) ) {[ORDER | CLUSTER} BY (column[, ...])}]]
{ {PIPELINED | AGGREGATE} [USING [schema.]implementation_type] |
    [PIPELINED] {IS | AS} }
{code_block | LANGUAGE {JAVA NAME external_program_name |
    C [NAME external_program_name]}
    LIBRARY lib_name [AGENT IN (argument[, ...])] [WITH CONTEXT]
    [PARAMETERS ( params[, ...] )]};
```

The `ALTER FUNCTION/PROCEDURE` statement shown next is used to recompile invalid UDFs or stored procedures:

```
ALTER {FUNCTION | PROCEDURE} [schema.]object_name
COMPILE [DEBUG] [compiler_param = value [...]] [REUSE SETTINGS]
```

Following are the parameter descriptions:

CREATE [OR REPLACE] {FUNCTION | PROCEDURE} [schema.]object_name

Creates a new UDF or stored procedure. Use *OR REPLACE* to replace an existing procedure or UDF without first dropping it and then having to reassign all permissions to it.

Certain clauses are only used with user-defined functions, including the *RETURN* clause, the *DETERMINISTIC* clause, and the entirety of the *USING* clause.

IN | OUT | IN OUT

Specifies whether a parameter is an input to the function, an output from the function, or both.

NOCOPY

Speeds up performance when an *OUT* or *IN OUT* argument is very large, as with a *VARRAY* or *RECORD* datatype.

AUTHID {CURRENT_USER | DEFINER}

Forces the routine to run in the permission context of either the current user or the person who owns the function, using *AUTHID CURRENT_USER* or *AUTHID DEFINER*, respectively.

AS EXTERNAL

Alternatively declares a C method. Oracle prefers the *AS LANGUAGE C* syntax except when PL/SQL datatypes need to be mapped to parameters.

PARALLEL_ENABLE

Enables the routine to be executed by a parallel query operation on a symmetric multi-processor (SMP) or parallel-processor server. This clause is used only for UDFs. (Do not use session state or package variables, because they can't be expected to be shared among parallel-execution servers.) Define the behavior of the *PARALLEL_ENABLE* query operation using these subclauses:

PARTITION prtn_name BY {ANY | {HASH | RANGE} (column[, ...])}

Defines partitioning of inputs on functions with *REF CURSOR* arguments. This may benefit table functions. *ANY* allows random partitioning. You can restrict partitioning to a specific *RANGE* or *HASH* partition on a comma-delimited *column* list.

{ORDER | CLUSTER} BY (column[, ...])

Orders or clusters parallel processing based on a comma-delimited *column* list. *ORDER BY* causes the rows to be locally ordered on the parallel-execution server according to the *column* list. *CLUSTER BY* restricts the rows on the parallel-execution server to the key values identified in the *column* list.

{PIPELINED | AGGREGATE} [USING [schema.]implementation_type]

PIPELINED iteratively returns the results of a table function, instead of the normal serial return of the *VARRAY* or nested table result set. This clause is used only for UDFs. The clause *PIPELINED USING implementation_type* is for an external UDF that uses a language such as C++ or Java. The *AGGREGATE USING implementation_type* clause defines a UDF as an aggregate function (a function that evaluates many rows but returns a single value).

IS | AS

Oracle treats *IS* and *AS* equally. Use either one to introduce the *code_block*.

code_block

Oracle allows a PL/SQL code block for user-defined functions and stored procedures. Alternately, you may use the *LANGUAGE* clause for stored procedures written in Java or C.

*LANGUAGE {JAVA NAME external_program_name | C [NAME external_program_name]
LIBRARY lib_name [AGENT IN (argument[, ...])] [WITH CONTEXT] [PARAMETERS
(params[, ...])] }*

Defines the Java or C implementation of the external program. The parameters and semantics of each declaration are specific to Java and C, not SQL.

ALTER {FUNCTION | PROCEDURE} [schema.]object_name

Recompiles an invalid standalone stored routine. Use *CREATE...OR REPLACE* to change the arguments, declarations, or definition of an existing routine.

COMPILE [DEBUG] [REUSE SETTINGS]

Recompiles the routine. Note that *COMPILE* is required. (You can see compile errors with the SQL*Plus command *SHOW ERRORS*.) The routine is marked valid if no compiler errors are encountered. The following optional subclauses may also be included with the *COMPILE* clause:

DEBUG

Generates and stores code used by the PL/SQL debugger.

compiler_param = value [...]

Specifies a PL/SQL compiler parameter. Allowable parameters include *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. Refer to Oracle's documentation on the PL/SQL compiler for more details.

REUSE SETTINGS

Maintains the existing compiler switch settings and reuses them for recompilation. Normally, Oracle drops and reacquires compiler switch settings.

In Oracle, UDFs and stored procedures are very similar in composition and structure. The primary difference is that a stored procedure cannot return a value to the invoking process, while a function may return a single value to the invoking process.

For example, you can pass in the name of a construction project to the following function to obtain the project's profit:

```
CREATE OR REPLACE FUNCTION project_revenue (project IN varchar2)
  RETURN NUMBER
  AS
    proj_rev NUMBER(10,2);
  BEGIN
    SELECT SUM(DECODE(action,'COMPLETED',amount,0)) -
           SUM(DECODE(action,'STARTED',amount,0)) +
           SUM(DECODE(action,'PAYMENT',amount,0))
    INTO proj_rev
    FROM construction_actions
    WHERE project_name = project;
    RETURN (proj_rev);
  END;
```

In this example, the UDF accepts the project name as an argument. Then it processes the project revenue, behind the scenes, by subtracting the starting costs from the completion payment and adding any other payments into the amount. The *RETURN (proj_rev);* line returns the amount to the invoking process.

In Oracle, UDFs *cannot* be used in the following situations:

- In a *CHECK* constraint or *DEFAULT* constraint of a *CREATE TABLE* or *ALTER TABLE* statement.
- In a *SELECT*, *INSERT*, *UPDATE*, or *DELETE*, the UDF cannot, either directly or indirectly (if invoked by another routine):
 - Have an *OUT* or *IN OUT* parameter. (Indirect calls may take *OUT* and *IN OUT* parameters.)
 - Terminate the transaction with *COMMIT*, *ROLLBACK*, *SAVEPOINT*, or a *CREATE*, *ALTER*, or *DROP* statement that implicitly issues a *COMMIT* or *ROLLBACK*.
 - Use session control (*SET ROLE*) or system control (the Oracle-specific statement *ALTER SYSTEM*) statements.
 - Write to a database (when a component of a *SELECT* statement or a parallelized *INSERT*, *UPDATE*, or *DELETE* statement).
 - Write to the same table that is modified by the statement that calls the UDF.

When you recompile a routine with the *ALTER* statement, it is marked valid if no compiler errors are encountered. If any errors are encountered, it is marked invalid. However, perhaps more importantly, any objects that depend upon the recompiled routine are marked invalid regardless of whether or not an error occurs. You can either recompile those dependent objects yourself, or allow Oracle to take some additional time to recompile them at runtime.

By way of example, the following statement recompiles the *project_revenue* function and maintains any compiler information for the PL/SQL debugger:

```
ALTER FUNCTION project_revenue COMPILE DEBUG;
```

PostgreSQL

PostgreSQL supports the *ALTER* and *CREATE FUNCTION* statements, but not *CREATE PROCEDURE* or *ALTER PROCEDURE*. This is because PostgreSQL's powerful functions can be used to simulate the processing performed by a procedure. The syntax to use to create a function is:

```
CREATE [OR REPLACE] FUNCTION routine_name
  ( [ parameter [{IN | OUT | INOUT}][, ...] ] )
  [RETURNS datatype]
  AS {code_block | object_file, link_symbol}
    [ LANGUAGE {C | SQL | internal} | {IMMUTABLE | STABLE | VOLATILE} |
      {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT} |
      [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER ]
    [WITH [{ISCACHABLE}[,] [ISSTRICT]]]
```

The syntax for *ALTER FUNCTION* follows:

```
ALTER FUNCTION routine_name
  ( [ parameter [{IN | OUT | INOUT}][, ...] ] )
  [RESTRICT]
  [RENAME TO new_routine_name]
  [OWNER TO new_owner_name]
  [SET SCHEMA new_schema_name]
```

The parameters are:

CREATE [OR REPLACE] FUNCTION *routine_name*

Creates a new function of the name you provide, or replaces an existing function. *OR REPLACE* does not enable you to change the name, input parameters, or output results of an existing function; you must drop and recreate a function to change any of those settings.

RETURNS *datatype*

Specifies the type of data required by the function. Not used with a procedure.

code_block* | *object_file*, *link_symbol

Defines the composition of the user-defined function. The *code_block* can be a string defining the function (dependent on the *LANGUAGE* setting), such as an internal function name, the path and name of an object file, a SQL query, or the text of a procedural language. The definition also can be an object file and link symbol for a C-language function.

LANGUAGE {C | SQL | *internal*}

Defines a call to an external program or an internal SQL routine. Since all of the language options except *SQL* are programs compiled in other languages, they are beyond the scope of this book. However, the *LANGUAGE SQL* clause should be used when writing SQL user-defined functions.



You can add a new language not installed by default on PostgreSQL by using the *CREATE LANGUAGE* statement.

IMMUTABLE | STABLE | VOLATILE

Describes the behavior of a function. *IMMUTABLE* describes a deterministic function that does not modify or access data stored within the database (i.e., no *SELECT* statements). *STABLE* describes a deterministic function that may modify or access data stored within the database (i.e., a *SELECT* statement that always returns the same data given the same inputs). When all of these descriptors are omitted, PostgreSQL assumes *VOLATILE*. *VOLATILE* indicates that the function is not deterministic (i.e., may give different results even when the inputs are always the same).

CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT

STRICT is a synonym for *RETURNS NULL ON NULL INPUT*. Both *RETURNS NULL ON NULL INPUT* and *CALLED ON NULL INPUT* are part of the ANSI standard and are described in that section.

[EXTERNAL] SECURITY {INVOKER | DEFINER}

INVOKER tells PostgreSQL to execute using the privileges of the user that called the routine, while *DEFINER* tells PostgreSQL to execute using the privileges of the user that created the routine. *EXTERNAL* is a noise word and is included only for ANSI compliance.

[WITH {[ISCACHABLE][,] [ISSTRICT]}]

Optimizes PostgreSQL performance by indicating that the function always returns the same value when provided with the same parameter values. *WITH ISCACHABLE* is similar to the ANSI setting *DETERMINISTIC*, except that this setting then allows the optimizer to pre-evaluate the call of the function. *WITH ISSTRICT* is similar to the ANSI setting *RETURNS NULL ON NULL INPUT*.

When omitted, the default behavior is similar to *CALLED ON NULL INPUT*.

Note that you may include both keywords in a single function declaration.

[RESTRICT] [RENAME TO *new_routine_name*] [OWNER TO *new_owner_name*] [SET SCHEMA *new_schema_name*]

Assigns a new name, owner, or schema to the routine. The *RESTRICT* keyword is noise.

PostgreSQL also allows function *overloading*, where the same function name is allowed for different functions, as long as they accept distinct input parameters.



Whenever you drop an existing function, all dependent objects are invalidated and must be recompiled.

Here's an example of a simple SQL function in PostgreSQL:

```
CREATE FUNCTION max_project_nbr
RETURNS int4
AS "SELECT MAX(title_ID) FROM titles AS RESULT"
LANGUAGE 'sql';
```

In this example, we created a UDF that returns the maximum *title_ID* from the *titles* table. PostgreSQL uses *CREATE FUNCTION* as a substitute for *CREATE PROCEDURE*, as well as to define actions for *CREATE TRIGGER*.

SQL Server

SQL Server supports *CREATE* and *ALTER* for both types of routines. By default, SQL Server stored procedures can return result sets and SQL Server UDFs can return single or multirow result sets using the *TABLE* datatype on *RETURNS* arguments, contrary to the ANSI standard. However, this makes SQL Server routines more flexible and powerful. Use the following syntax to create a user-defined function or stored procedure:

```
CREATE {FUNCTION | PROCEDURE} [owner_name.]object_name[;int]
( [ {@parameter datatype [VARYING] [=default] [OUTPUT]} ] [READONLY][, ...] ] )
[RETURNS {datatype | TABLE}]
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE | RECOMPILE, ENCRYPTION | 
      RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT |
      EXEC[UTE] AS {CALLER | SELF | OWNER | 'user_name'}}]
[FOR REPLICATION]
[AS]
code_block
```

Use the following syntax to alter an existing user-defined function or stored procedure:

```
ALTER {FUNCTION | PROCEDURE} [owner_name.]object_name[;int]
( [ {@parameter datatype [VARYING] [=default] [OUTPUT]} ] [, ...] ] )
[RETURNS {datatype | TABLE}]
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE | RECOMPILE, ENCRYPTION | 
      RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT |
      EXEC[UTE] AS {CALLER | SELF | OWNER | 'user_name'}}]
[FOR REPLICATION]
[AS]
code_block
```

Following are the parameter descriptions:

CREATE {FUNCTION | PROCEDURE} [owner_name.]object_name[;int]

Creates a new UDF or stored procedure in the current database. For SQL Server stored procedures, you may optionally specify a version number in the format *procedure_name;1*, where 1 is an integer indicating the version number. This facility allows you to have multiple versions of a single stored procedure.

{@parameter datatype [VARYING] [=default] [OUTPUT]} [READONLY][, ...]

Defines one or more input arguments for a UDF or stored procedure. SQL Server parameters are always declared with an at sign (@) as the first character.

VARYING

Used in stored procedures with a CURSOR datatype parameter. Indicates that the procedure constructs the result set dynamically.

=default

Assigns a default value to the parameter. The default value is used whenever the stored procedure or UDF is invoked without a value being supplied for the parameter.

OUTPUT

Used for stored procedures, OUTPUT is functionally equivalent to the ANSI OUT clause in the CREATE FUNCTION statement. The value stored in the return parameter is passed back to the calling procedure through the return variables of the SQL Server EXEC[UTE] command. Output parameters can be any datatype except TEXT and IMAGE.

READONLY

Used in functions to indicate that the parameter cannot be updated or modified within the routine code body. This is especially useful for user-defined TABLE types.

RETURNS {datatype | TABLE}

Allows SQL Server UDFs to return a single *datatype* value or to return multiple values via the TABLE datatype. The TABLE datatype is considered *inline* if it has no accompanying column list and is defined with a single SELECT statement. If the RETURNS clause returns multiple values via the TABLE datatype, and if the TABLE has defined columns and datatypes, the function is considered a *multi-statement, table-valued* function.

WITH

Allows the assignment of additional characteristics to a SQL Server UDF or stored procedure.

ENCRYPTION

Tells SQL Server to encrypt the column in the system table that stores the text of the function, thus preventing unwarranted review of the function code. Usable by both UDFs and stored procedures.

SCHEMABINDING

Specifies that the function is bound to a specific database object, such as a table or view. That database object cannot be altered or dropped as long as the function exists (or maintains the SCHEMABINDING option). Usable only by UDFs.

RECOMPILE

Tells SQL Server not to store a cache plan for the stored procedure, but instead to recompile the cache plan each time the stored procedure is executed. This is useful when using atypical or temporary values in the procedure, but it can cause serious degradation in performance. Usable only by stored procedures. Note that *RECOMPILE* and *ENCRYPTION* can be invoked together.

EXEC[UTE] AS {CALLER | SELF| OWNER | 'user_name'}

Optional clause, for both procedures and functions, to specify the privileges under which the routine will execute. *CALLER* indicates the routine will run with the privileges of the user invoking the routine. When this clause is omitted, SQL Server assumes *CALLER*. *SELF* indicates the routine will run with the privileges of the creator of the routine. *OWNER* indicates the routine will run with the privileges of the current owner of the routine. *'user_name'* indicates the routine will run with the privileges of the named, pre-existing user.

FOR REPLICATION

Disables execution of the stored procedure on a subscribing server. This clause is used primarily to create a filtering stored procedure that is executed only by SQL Server's built-in replication engine. It is incompatible with *WITH RECOMPILE*.

Like tables (see *CREATE TABLE*), local and global temporary stored procedures may be declared by prefixing a pound symbol (#) or double pound symbol (##) to the name of the procedure, respectively. Temporary procedures exist only for the duration of the user or process session that created them. When that session ends, the temporary procedures automatically delete themselves.

A SQL Server stored procedure or UDF may have as many as 1,024 input parameters, specified by the at sign (@@). Parameters are defined using SQL Server datatypes. (Parameters of the *CURSOR* datatype must be defined with both *VARYING* and *OUTPUT*.) The user or calling process must supply values for any input parameters. However, a default value can be supplied for any input parameter to allow the procedure to execute without a user- or process-supplied value. The default must be a constant or NULL, but it may contain wildcard characters.

SQL Server requires that one or more user-supplied parameters be declared for a given user-defined function. All SQL Server datatypes are supported as parameters, except *TIMESTAMP*. Values returned by the function can be any datatype except *TIME-STAMP*, *TEXT*, *NTEXT*, or *IMAGE*. If an inline table value is required, the *TABLE* option without an accompanying column list may be used.



The *ALTER FUNCTION* and *ALTER PROCEDURE* statements support the full syntax provided by the corresponding *CREATE* statements. You can use the *ALTER* statements to change any of the attributes of an existing routine without changing permissions or affecting any dependent objects.

For UDFs, the *code_block* is either a single *SELECT* statement for an inline function, in the format *RETURN (SELECT...)*, or a series of Transact-SQL statements following the *AS* clause for a multistatement operation. When using *RETURN (SELECT)*, the *AS* clause is optional. Here are some other rules for SQL Server UDFs:

- When using the *AS* clause, the *code_body* should be enclosed in *BEGIN...END* delimiters.
- UDFs cannot make any permanent changes to data or cause other lasting side effects. A number of other restrictions exist as a result. For example, *INSERT*, *UPDATE*, and *DELETE* statements may modify only *TABLE* variables local to a function.
- When returning a scalar value, a SQL Server UDF must contain the clause *RETURN datatype*, where *datatype* is the same as that identified in the *RETURNS* clause.
- The last statement of the *code_block* must be an unconditional *RETURN* that returns a single datatype value or *TABLE* value.
- The *code_block* may not contain any global variables that return a perpetually changing value, such as *@@CONNECTIONS* or *GETDATE*. However, it may contain variables that return a single, unchanging value, such as *@@SERVERNAME*.

The following is an example of a *scalar* function that returns a single value. Once created, the scalar UDF can then be utilized in a query just like a system-supplied function:

```
CREATE FUNCTION metric_volume -- Input dimensions in centimeters.
    (@length decimal(4,1),
     @width decimal(4,1),
     @height decimal(4,1) )
RETURNS decimal(12,3) -- Cubic centimeters.
AS BEGIN
    RETURN ( @length * @width * @height )
END
GO
SELECT project_name,
    metric_volume(construction_height,
        construction_length,
        construction_width)
FROM housing_construction
WHERE metric_volume(construction_height,
        construction_length,
        construction_width) >= 300000
GO
```

An inline table-valued UDF supplies values via a single *SELECT* statement using an *AS RETURN* clause. For example, we can supply a store ID and find all of that store's **titles**:

```
CREATE FUNCTION stores_titles(@stor_id varchar(30))
RETURNS TABLE
AS
RETURN (SELECT title, qty
    FROM sales AS s
    JOIN titles AS t ON t.title_id = s.title_id
    WHERE s.stor_id = @storeid )
```

Now, let's alter the UDF just a bit by changing the input argument datatype length and adding another condition to the *WHERE* clause (changes indicated in boldface):

```
ALTER FUNCTION stores_titles(@stor_id VARCHAR(4))
RETURNS TABLE
```

```

AS
RETURN (SELECT title, qty
       FROM sales AS s
      JOIN titles AS t ON t.title_id = s.title_id
     WHERE s.store_id = @storeid
       AND s.city = 'New York')

```

User-defined functions that return *TABLE* values are often selected as result set values or are used in the *FROM* clause of a *SELECT* statement, just as a regular table is used. These *multi-statement, table-valued functions* can have very elaborate code bodies since the *code_block* is composed of many Transact-SQL statements that populate a *TABLE* return variable.

Here is an example invoking a multistatement, table-valued function in a *FROM* clause. Notice that a table alias is assigned, just as for a regular table:

```

SELECT co.order_id, co.order_price
  FROM construction_orders AS co,
       fn_construction_projects('Cancelled') AS fcp
 WHERE co.construction_id = fcp.construction_id
 ORDER BY co.order_id
 GO

```

For stored procedures, the *code_block* clause contains one or more Transact-SQL commands, up to a maximum size of 128 MB, delimited by *BEGIN* and *END* clauses. Some rules about Microsoft SQL Server stored procedures include:

- The *code_block* allows most valid Transact-SQL statements, but *SET SHOWPLAN_TEXT* and *SET SHOWPLAN_ALL* are prohibited.
- Some other commands have restricted usages within stored procedures, including *ALTER TABLE*, *CREATE INDEX*, *CREATE TABLE*, all *DBCC* statements, *DROP TABLE*, *DROP INDEX*, *TRUNCATE TABLE*, and *UPDATE STATISTICS*.
- SQL Server allows deferred name resolution, meaning that a stored procedure compiles without an error even though it references an object that has not yet been created. SQL Server creates the execution plan and fails only when the object is actually invoked (for instance, in a stored procedure), if the object still doesn't exist.
- Stored procedures can be nested easily in SQL Server. Whenever a stored procedure invokes another stored procedure, the system variable *@@NESTLEVEL* is incremented by 1. It is decreased by 1 when the called procedure completes. Use *SELECT @@NESTLEVEL* inside a procedure or from an ad hoc query session to find the current nesting depth.

In the following example, a SQL Server stored procedure generates a unique 22-digit value (based on elements of the system date and time) and returns it to the calling process:

```

-- A Microsoft SQL Server stored procedure
CREATE PROCEDURE get_next_nbr
    @next_nbr CHAR(22) OUTPUT
AS
BEGIN
    DECLARE @random_nbr INT
    SELECT @random_nbr = RAND() * 1000000

```

```

SELECT @next_nbr =
    RIGHT('000000' + CAST(ROUND(RAND(@random_nbr)*1000000,0))
        AS CHAR(6), 6) +
    RIGHT('0000' + CAST(DATEPART (yy, GETDATE() )
        AS CHAR(4)), 2) +
    RIGHT('000' + CAST(DATEPART (dy, GETDATE() )
        AS CHAR(3)), 3) +
    RIGHT('00' + CAST(DATEPART (hh, GETDATE() )
        AS CHAR(2)), 2) +
    RIGHT('00' + CAST(DATEPART (mi, GETDATE() )
        AS CHAR(2)), 2) +
    RIGHT('00' + CAST(DATEPART (ss, GETDATE() )
        AS CHAR(2)), 2) +
    RIGHT('000' + CAST(DATEPART (ms, GETDATE() )
        AS CHAR(3)), 3)
END
GO

```

SQL Server supports functions and procedures written in Microsoft .NET Framework common language runtime (CLR) methods that can take and return user-supplied parameters. These routines have similar *CREATE* and *ALTER* declarations to regular SQL routines and functions; however, the code bodies are external assemblies. Refer to the SQL Server documentation if you want to learn more about programming routines using the CLR.

See Also

CALL
RETURN

CREATE/ALTER INDEX Statement

Indexes are special objects built on top of tables that speed many data-manipulation operations, such as *SELECT*, *UPDATE*, and *DELETE* statements. The selectivity of a given *WHERE* clause and the available query plans the database query optimizer can choose from are usually based upon the quality of the indexes that have been placed on the table in a given database.

The *CREATE INDEX* command is not a part of the ANSI SQL standard, and thus its syntax varies greatly among vendors.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

Common Vendor Syntax

```
CREATE [UNIQUE] INDEX index_name ON table_name
    (column_name[, ...])
```

Keywords

`CREATE [UNIQUE] INDEX index_name`

Creates a new index named *index_name* in the current database and schema context. Since indexes are associated with specific tables (or sometimes views), the *index_name* need only be unique to the table it is dependent on. The *UNIQUE* keyword defines the index as a unique constraint for the table and disallows any duplicate values into the indexed column or columns of the table. (Refer to “Constraints.”)

`table_name`

Declares the pre-existing table with which the index is associated. The index is dependent upon the table: if the table is dropped, so is the index.

`column_name[, ...]`

Defines one or more columns in the table that are indexed. The pointers derived from the indexed column or columns enable the database query optimizer to greatly speed up data-manipulation operations such as *SELECT* and *DELETE* statements. All major vendors support *composite indexes*, also known as *concatenated indexes*, which are used when two or more columns are best searched as a unit (for example, `last_name` and `first_name` columns).

Rules at a Glance

Indexes are created on tables to speed data-manipulation operations against those tables, such as those in a *WHERE* or *JOIN* clause. Indexes may also speed other operations, including:

- Identifying a *MIN()* or *MAX()* value in an indexed column.
- Sorting or grouping columns of a table.
- Searching based on *IS NULL* or *IS NOT NULL*.
- Fetching data quickly when the indexed data is all that is requested. A *SELECT* statement that retrieves data from an index and not directly from the table itself is called a *covering query*. An index that answers a query in this way is a *covering index*.

After creating a table, you can create indexes on columns within the table. It is a good idea to create indexes on columns that are frequently part of the *WHERE* clauses or *JOIN* clauses of the queries made against a table. For example, the following statement creates an index on a column in the `sales` table that is frequently used in the *WHERE* clauses of queries against that table:

```
CREATE INDEX ndx_ord_date ON sales(ord_date);
```

In another case, we want to set up the `pub_name` and `country` as a unique index on the `publishers` table:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country);
```

Since the index is unique, any new record entered into the `publishers` table must have a unique combination of publisher name and country.



Some vendor platforms allow you to create indexes on views as well as tables.

Programming Tips and Gotchas

Concatenated indexes are most useful when queries address the columns of the index starting from the left. If you omit left-side columns in a query against a concatenated key, the query may not perform as well. For example, assume that we have a concatenated index on `(last_name, first_name)`. If we query only by `first_name`, the concatenated index that starts with `last_name` and includes `first_name` may not be any good to us. That said, some of the vendor platforms have now advanced their query engines to the point where this is much less of a problem than it used to be.



Creating an index on a table may cause that table to take up as much as 1.2 to 1.5 times more space than the table currently occupies. Make sure you have enough room! Most of that space is released after the index has been created.

You should be aware that there are situations in which too many indexes can actually slow down system performance. In general, indexes greatly speed lookup operations against a table or view, especially in *SELECT* statements. However, every index you create adds overhead whenever you perform an *UPDATE*, *DELETE*, or *INSERT* operation, because the database must update all dependent indexes with the values that have changed in the table. As a rule of thumb, 6 to 12 indexes are about the most you'll want to create on a single table.

In addition, indexes take up extra space within the database. The more columns there are in an index, the more space it consumes. This is not usually a problem, but it sometimes catches the novices off guard when they're developing a new database.

Most databases use indexes to create statistical samplings (usually just called *statistics*), so the query engine can quickly determine which, if any, index or combination of indexes will be most useful for a query. These indexes are always fresh and useful when the index is first created, but they may become stale and less useful over time as records in the table are deleted, updated, and inserted. Consequently, indexes, like day-old bread, are not guaranteed to be useful as they age. You need to be sure to refresh, rebuild, and maintain your databases regularly to keep index statistics fresh.

MySQL

MySQL supports a form of the *CREATE INDEX* statement, but not the *ALTER INDEX* statement. The types of indexes you can create in MySQL are determined by the engine type, and the indexes are not necessarily stored in B-tree structures on the filesystem. Strings within an index are automatically prefix- and end-space-compressed. MySQL's *CREATE INDEX* syntax is:

```
CREATE [ONLINE | OFFLINE] [UNIQUE | FULLTEXT] INDEX index_name
  [USING {BTREE | HASH | RTREE}]
  ON table_name (column_name(length)[, ...])
  [KEY BLOCK SIZE int]
  [WITH PARSER parser_name]
```

where:

FULLTEXT

Creates a full-text search index against a column. Full-text indexes are only supported on MyISAM table types and *CHAR*, *VARCHAR*, or *TEXT* datatypes. They do not support the *(length)* clause.

ONLINE | OFFLINE

ONLINE indicates that creating the index does not require the subject table to be copied first. *OFFLINE* copies the table before creating the index.

USING {BTREE | HASH | RTREE}

Specifies a specific type of index to use. *RTREE* indexes may be used only for *SPATIAL* indexes. Use this hint sparingly, since different storage engines allow different index types: MyISAM allows *BTREE* and *RTREE*, InnoDB allows only *BTREE*, NDB allows only *HASH* (and allows the *USING* clause only for unique keys and primary keys), and MEMORY/HEAP allows *HASH* and *BTREE*. This clause deprecates the *TYPE type_name* clause found in MySQL 5.1.10 and earlier.

KEY BLOCK SIZE int

Provides a hint to the storage engine about the size to use for index key blocks, where *int* is the value in kilobytes to use. A value of 0 means that the default for the storage engine should be used.

WITH PARSER parser_name

Used only with *FULLTEXT* indexes, this clause associates a parser plug-in with the index. Plug-ins are fully documented in the MySQL documentation.

MySQL supports the basic industry standard syntax for the *CREATE INDEX* statement. Interestingly, MySQL also lets you build an index on the first *length* characters of a *CHAR* or *VARCHAR* column. MySQL requires the *length* clause for *BLOB* and *TEXT* columns. Specifying a length can be useful when selectivity is sufficient in the first, say, 10 characters of a column, and in those situations where saving disk space is very important. This example indexes only the first 25 characters of the *pub_name* column and the first 10 characters of the *country* column:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name(25),
                                         country(10))
```

As a general rule, MySQL allows at least 16 keys per table, with a total maximum length of at least 256 bytes. This can vary by storage engine, however.

Oracle

Oracle allows the creation of indexes on tables, partitioned tables, clusters, and index-organized tables, as well as on scalar type object attributes of a typed table or cluster, and on nested table columns using the *CREATE INDEX* statement. Oracle also allows several types of indexes, including normal B-tree indexes, *BITMAP* indexes (useful for columns that have each value repeated 100 or more times), partitioned indexes, function-based indexes (based on an expression rather than a column value), and domain indexes.



Oracle index names must be unique within a schema, not just to the table to which they are assigned.

Oracle also supports the *ALTER INDEX* statement, which is used to change or rebuild an existing index without forcing the user to drop and recreate the index. Oracle's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE | BITMAP] INDEX index_name
{ON
  {table_name ({column | expression} [ASC | DESC][, ...])
  [{INDEXTYPE IS index_type [PARALLEL [int] | NOPARALLEL]}}
```

```
[PARAMETERS ('values')] }] |  
CLUSTER cluster_name |  
FROM table_name WHERE condition [LOCAL partitioning]  
[ {LOCAL partitioning | GLOBAL partitioning} ]  
[physical_attributes_clause] [{LOGGING | NOLOGGING}] [ONLINE]  
[COMPUTE STATISTICS] [{TABLESPACE tablespace_name | DEFAULT}]  
[{COMPRESS int | NOCOMPRESS}] [{NOSORT | SORT}] [REVERSE]  
[{VISIBLE | INVISIBLE}] [{PARALLEL [int] | NOPARALLEL}] }
```

and the syntax for ALTER INDEX is:

```
ALTER INDEX index_name  
{ {ENABLE | DISABLE} | UNUSABLE | {VISIBLE | INVISIBLE} |  
RENAME TO new_index_name | COALESCE |  
[NO]MONITORING USAGE | UPDATE BLOCK REFERENCES |  
PARAMETERS ('ODCI_params') | alter_index_partitioning_clause |  
rebuild_clause |  
[DEALLOCATE UNUSED [KEEP int [K | M | G | T]]]  
[ALLOCATE EXTENT ( [SIZE int [K | M | G | T]] [DATAFILE 'filename']  
[INSTANCE int])]  
[SHRINK SPACE [COMPACT] [CASCADE]]  
[{PARALLEL [int] | NOPARALLEL}]  
[{LOGGING | NOLOGGING}]  
[physical_attributes_clause] }
```

where the non-ANSI clauses are:

BITMAP

Creates an index bitmap for each index value, rather than indexing each individual row. Bitmaps are best for low-concurrency tables (e.g., read-intensive tables). *BITMAP* indexes are incompatible with global partitioned indexes, the *INDEXTYPE* clause, and index-organized tables without a mapping table association.

ASC | DESC

Specifies that the values in the index be kept in either ascending (ASC) or descending (DESC) order. When *ASC* or *DESC* is omitted, *ASC* is used by default. However, be aware that Oracle treats *DESC* indexes as function-based indexes, so there is some difference in functionality between *ASC* and *DESC* indexes. You may not use *ASC* or *DESC* when you are using the *INDEXTYPE* clause. *DESC* is ignored on *BITMAP* indexes.

INDEXTYPE IS *index_type* [PARAMETERS ('*values*')]

Creates an index on a user-defined type of *index_type*. Domain indexes require that the user-defined type already exists. If the user-defined type requires arguments, pass them in using the optional *PARAMETERS* clause. You may also optionally parallelize the creation of the type index using the *PARALLEL* clause (explained in more detail later in this list).

CLUSTER *cluster_name*

Declares a clustering index based on the specified pre-existing *cluster_name*. On Oracle, a clustering index physically co-locates two tables that are frequently queried on the same columns, usually a primary key and a foreign key. (Clusters are created with the Oracle-specific command *CREATE CLUSTER*.) You do not declare a table or columns on a *CLUSTER* index, since both the tables involved and the columns indexed are already declared with the previously issued *CREATE CLUSTER* statement.

GLOBAL partitioning

Includes the full syntax:

```
GLOBAL PARTITION BY
  {RANGE (column_list) ( PARTITION [partition_name] VALUE LESS THAN
    (value_list) [physical_attributes_clause]
    [TABLESPACE tablespace_name] [LOGGING | NOLOGGING][, ...] )} |
  {HASH (column_list) ( PARTITION [partition_name] )
    {[TABLESPACE tablespace_name] [[OVERFLOW] TABLESPACE
      tablespace_name] [VARARRAY varray_name STORE AS LOB
      lob_segment_name] [LOB (lob_name) STORE AS [lob_segment_name]] |
      [TABLESPACE tablespace_name]]} |
    [STORE IN (tablespace_name[, ...])] [OVERFLOW STORE IN
      (tablespace_name [,...])]}[, ...]
```

The *GLOBAL PARTITION* clause declares that the global index is manually partitioned via either range or hash partitioning onto *partition_name*. (The default is to partition the index equally in the same way the underlying table is partitioned, if at all.) You can specify a maximum of 32 columns, though none may be *ROWID*. You may also apply the *[NO]LOGGING* clause, the *TABLESPACE* clause, and the *physical_attributes_clause* (defined earlier) to a specific partition. You cannot partition on *ROWID*. You may include one or more partitions, along with any attributes, in a comma-delimited list, according to the following:

RANGE

Creates a range-partitioned global index based on the range of values from the table columns listed in the *column_list*.

VALUE LESS THAN (value_list)

Sets an upper bound for the current partition of the global index. The values in the *value_list* correspond to the columns in the *column_list*, both of which are comma-delimited lists of columns. Both lists are prefix-dependent, meaning that for a table with columns *a*, *b*, and *c*, you could define partitioning on *(a, b)* or *(a, b, c)*, but not *(b, c)*. The last value in the list should always be the keyword *MAXVALUE*.

HASH

Creates hash-partitioned global index, assigning rows in the index to each partition based on a hash function of the values of the columns in the *column_list*. You may specify the exact tablespace to store special database objects such as *VARARRAYs* and *LOBs*, and for any *OVERFLOW* of the specified (or default) tablespaces.

LOCAL partitioning

Supports local index partitioning on *range*-partitioned indexes, *list*-partitioned indexes, *hash*-partitioned indexes, and *composite*-partitioned indexes. You may include zero or more partitions, along with any attributes, in a comma-delimited list. When this clause is omitted, Oracle generates one or more partitions consistent with those of the table partition. Index partitioning is done in one of three ways:

Range- and list-partitioned indexes

Applied to regular or equipartitioned tables. Range- and list-partitioned indexes (synonyms for the same thing) follow the syntax:

```
LOCAL [ (PARTITION [partition_name]
  { [physical_attributes_clause] [TABLESPACE tablespace_name]
  [LOGGING | NOLOGGING] |
  [COMPRESS | NOCOMPRESS] }[, ...]) ]
```

All of the options are the same as for *GLOBAL PARTITION* (see earlier), except that the scope is for a local index.

Hash-partitioned indexes

Applied to hash-partitioned tables. Hash-partitioned indexes allow you to choose between the earlier syntax and the following optional syntax:

```
LOCAL {STORE IN (tablespace_name[, ...]) |  
  (PARTITION [partition_name] [TABLESPACE tablespace_name])}
```

to store the index partition on a specific tablespace. When you supply more tablespace names than index partitions, Oracle will cycle through the tablespaces when it partitions the data.

Composite-partitioned indexes

Applied on composite-partitioned tables, using the following syntax:

```
LOCAL [STORE IN (tablespace_name[, ...])]  
PARTITION [partition_name]  
  { [physical_attributes_clause] [TABLESPACE tablespace_name]  
  [LOGGING | NOLOGGING] |  
  [COMPRESS | NOCOMPRESS]}  
  [ {STORE IN (tablespace_name[, ...]) |  
  (SUBPARTITION [subpartition_name] [TABLESPACE tablespace_  
  name])} ]
```

You may use the *LOCAL STORE* clause shown under the hash-partitioned indexes entry, or the *LOCAL* clause shown under the range- and list-partitioned indexes entry. (When using the *LOCAL* clause, substitute the keyword *SUBPARTITION* for *PARTITION*.)

physical_attributes_clause

Establishes values for one or more of the following settings: *PCTFREE int*, *PCTUSED int*, and *INITRANS int*. When this clause is omitted, Oracle defaults to *PCTFREE 10*, *PCTUSED 40*, and *INITRANS 2*.

PCTFREE int

Designates the percentage of free space to leave on each block of the index as it is created. This speeds up new entries and updates on the table. However, *PCTFREE* is applied only when the index is created. It is not maintained over time. Therefore, the amount of free space can erode over time as records are inserted, updated, and deleted from the index. This clause is not allowed on index-organized tables.

PCTUSED int

Designates the minimum percentage of used space to be maintained on each data block. A block becomes available to row insertions when its used space falls below the value specified for *PCTUSED*. The default is 40. The sum of *PCTFREE* and *PCTUSED* must be equal to or less than 100.

INITRANS int

Designates the initial number of concurrent transactions allocated to each data block of the database. The value may range from 1 to 255.



In versions prior to 11g the *MAXTRANS* parameter was used to define the maximum allowed number of concurrent transactions on a data block, but this parameter has now been deprecated. Oracle 11g automatically sets *MAXTRANS* to 255, silently overriding any other value that you specify for this parameter (although existing objects retain their established *MAXTRANS* settings).

LOGGING | NOLOGGING

Tells Oracle to log the creation of the index on the redo logfile (*LOGGING*), or not to log it (*NOLOGGING*). This clause also sets the default behavior for subsequent bulk loads using Oracle SQL*Loader. For partitioned indexes, this clause establishes the default values of all partitions and segments associated with the partitions, and the defaults used on any partitions or subpartitions added later with an *ALTER TABLE...ADD PARTITION* statement. (When using *NOLOGGING*, we recommend that you take a full backup after the index has been loaded in case the index has to be rebuilt due to a failure.)

ONLINE

Allows data manipulation on the table while the index is being created. Even with *ONLINE*, there is a very small window at the end of the index creation operation where the table will be locked while the operation completes. Any changes made to the base table at that time will then be reflected in the newly created index. *ONLINE* is incompatible with bitmap, cluster, or parallel clauses. It also cannot be used on indexes on a *UROWID* column or on index-organized tables with more than 32 columns in their primary keys.

COMPUTE [STATISTICS]

Collects statistics while the index is being created, when it can be done with relatively little cost. Otherwise, you will have to collect statistics after the index is created.

TABLESPACE {tablespace_name | DEFAULT}

Assigns the index to a specific tablespace. When omitted, the index is placed in the default tablespace. Use the *DEFAULT* keyword to explicitly place an index into the default tablespace. (When local partitioned indexes are placed in *TABLESPACE DEFAULT*, the index partition (or subpartition) is placed in the corresponding tablespace of the base table partition (or subpartition).

COMPRESS [int] | NOCOMPRESS

Enables or disables key compression, respectively. Compression eliminates repeated occurrences of key column values, yielding a substantial space savings at the cost of speed. The integer value, *int*, defines the number of prefix keys to compress. The value can range from 1 to the number of columns in the index for nonunique indexes, and from 1 to $n-1$ columns for unique indexes. The default is *NOCOMPRESS*, but if you specify *COMPRESS* without an *int* value, the default is *COMPRESS n* (for nonunique indexes) or *COMPRESS n-1* (for unique indexes), where *n* is the number of columns in the index. *COMPRESS* can only be used on nonpartitioned and nonbitmapped indexes.

NOSORT | REVERSE

NOSORT allows an index to be created quickly for a column that is already sorted in ascending order. If the values of the column are not in perfect ascending order, the operation aborts, allowing a retry without the *NOSORT* option. *REVERSE*, by contrast, places the index blocks in storage by reverse order (excluding *ROWID*).

REVERSE is mutually exclusive of **NOSORT** and cannot be used on a bitmap index or an index-organized table. **NOSORT** is most useful for creating indexes immediately after a base table is loaded with data in presorted order.

VISIBLE | INVISIBLE

Declares whether the index is visible or invisible to the optimizer. Invisible indexes are maintained by DML operations but are not normally used by the optimizer for query performance. This is very useful when you cannot alter an index to disable it, but you really need Oracle to ignore the index.

PARALLEL [int] | NOPARALLEL

Allows the parallel creation of the index using multiple server process, each operating on a distinct subset of the index, to speed up the operation. An optional integer value, *int*, may be supplied to define the exact number of parallel threads used in the operation. When omitted, Oracle calculates the number of parallel threads to use. **NOPARALLEL**, the default, causes the index to be created serially.

ENABLE | DISABLE

Enables or disables a pre-existing function-based index, respectively. You cannot specify any other clauses of the **ALTER INDEX** statement with **ENABLE** or **DISABLE**.

UNUSABLE

Marks the index (or index partition or subpartition) as unusable. When **UNUSABLE**, an index (or index partition or subpartition) may only be rebuilt or dropped and recreated before it can be used.

RENAME TO *new_index_name*

Renames the index from *index_name* to *new_index_name*.

COALESCE

Merges the contents of index blocks used to maintain the index-organized table so that the blocks can be reused. **COALESCE** is similar to **SHRINK**, though **COALESCE** compacts the segments less densely than **SHRINK** and does not release unused space.

[NO]MONITORING USAGE

Declares that Oracle should clear existing information on index usage and monitor the index, posting information in the **V\$OBJECT_USAGE** dynamic performance view, until **ALTER INDEX...NOMONITORING USAGE** is executed. The **NOMONITORING USAGE** clause explicitly disables this behavior.

UPDATE BLOCK REFERENCES

Updates all stale guess data block addresses stored as part of the index row on normal or domain indexes of an index-organized table. The guess data blocks contain the correct database addresses for the corresponding blocks identified by the primary key. This clause cannot be used with other clauses of the **ALTER INDEX** statement.

PARAMETERS ('*ODCI_params*')

Specifies a parameter string passed, without interpretation, to the ODCI index-type routine of a domain index. The parameter string, called '*ODCI_params*', may be up to 1,000 characters long. Refer to the vendor documentation for more information on ODCI parameter strings.

alter_index_partitioning_clause

Refer to in the section “Oracle partitioned and subpartitioned tables” under “Oracle” in the section on **CREATE/ALTER TABLE** for more details.

rebuild clause

Rebuilds the index, or a specific partition (or subpartition) of the index. A successful rebuild marks an *UNUSABLE* index as *USABLE*. The syntax for the *rebuild clause* is:

```
REBUILD {[NO]REVERSE | [SUB]PARTITION partn_name}  
[{PARALLEL int | NOPARALLEL}] [TABLESPACE tablespace_name]  
[PARAMETERS ('ODCI_params')] [ONLINE] [COMPUTE STATISTICS]  
[COMPRESS int | NOCOMPRESS] [[NO]LOGGING]  
[physical_attributes_clause]
```

where:

[NO]REVERSE

Stores the bytes of the index block in reverse order and excludes rows when the index is rebuilt (*REVERSE*), or stores the bytes of the index blocks in regular order (*NOREVERSE*).

DEALLOCATE UNUSED [KEEP *int* [K | M | G | T]]

Deallocates unused space at the end of the index (or at the end of each range or hash partition of a partitioned index) and frees the space for other segments in the tablespace. The optional *KEEP* keyword defines how many bytes (*int*) above the high-water mark the index will keep after deallocation. You can append a suffix to the *int* value to indicate that the value is expressed in kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T). When the *KEEP* clause is omitted, all unused space is freed.

ALLOCATE EXTENT ([SIZE *int* [K | M | G | T]] [DATAFILE '*filename*'] [INSTANCE *int*])

Explicitly allocates a new extent for the index using the specified parameters. You may mix and match any of the parameters. *SIZE* specifies the size of the next extent, in bytes (no suffix), kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T). *DATAFILE* allocates an entirely new datafile to the index extent. *INSTANCE*, used only on Oracle RACs, makes a new extent available to a freelist group associated with the specified instance.

SHRINK SPACE [COMPACT] [CASCADE]

Shrinks the index segments, though only segments in tablespaces with automatic segment management may be shrunk. Shrinking a segment moves rows in the table, so make sure *ENABLE ROW MOVEMENT* is also used in the *ALTER TABLE...SHRINK* statement. Oracle compacts the segment, releases the emptied space, and adjusts the high-water mark, unless the optional keywords *COMPACT* and/or *CASCADE* are applied. The *COMPACT* keyword only defragments the segment space and compacts the index; it does not readjust the high-water mark or empty the space immediately. The *CASCADE* keyword performs the same shrinking operation (with some restrictions and exceptions) on all dependent objects of the index. The statement *ALTER INDEX...SHRINK SPACE COMPACT* is functionally equivalent to *ALTER INDEX...COALESCE*.

By default, Oracle indexes are non-unique. It is also important to know that Oracle's regular B-tree indexes do not include records that have a NULL key value.

Oracle does not support indexes on columns with the following datatypes: *LONG*, *LONG RAW*, *REF* (with the *SCOPE* attribute), or any user-defined datatype. You may create indexes on functions and expressions, but they cannot allow NULL values or aggregate functions. When you create an index on a function, if it has no parameters

the function should show an empty set (for example, `function_name()`). If the function is a UDF, it must be *DETERMINISTIC*.

Oracle supports a special index structure called an *index-organized table* (IOT) that combines the table data and primary key index on a single physical structure, instead of having separate structures for the table and the index. IOTs are created using the `CREATE TABLE...ORGANIZATION INDEX` statement. Refer to the section on the `CREATE/ALTER TABLE` statement for more information on making an IOT.

Oracle automatically creates any additional indexes on an index-organized table as secondary indexes. Secondary indexes do not support the *REVERSE* clause.

Oracle allows the creation of partitioned indexes and tables with the *PARTITION* clause. Consequently, Oracle's indexes also support partitioned tables. The *LOCAL* clause tells Oracle to create separate indexes for each partition of a table. The *GLOBAL* clause tells Oracle to create a common index for all the partitions.

Note that any time an object name is referenced in the syntax diagram, you may optionally supply the schema. This applies to indexes, tables, etc., but not to tablespaces. You must have the explicitly declared privilege to create an index in a schema other than the current one.

As an example, you can use a statement such as the following to create an Oracle index that is compressed and created in parallel, with compiled statistics, but without logging the creation:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
  COMPRESS 1 PARALLEL NOLOGGING COMPUTE STATISTICS;
```

As with other Oracle object creation statements, you can control how much space the index consumes and in what increments it grows. The following example constructs an index in Oracle on a specific tablespace with specific instructions for how the data is to be stored:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
  STORAGE (INITIAL 10M NEXT 5M PCTINCREASE 0)
  TABLESPACE publishers;
```

For example, when you create the **housing_construction** table as a partitioned table on an Oracle server, you should also create a partitioned index with its own index partitions:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
  ON housing_construction(project_id)
  GLOBAL PARTITION BY RANGE (project_id)
    (PARTITION part1 VALUES LESS THAN ('H')
      TABLESPACE construction_part1_ndx_ts,
    PARTITION part2 VALUES LESS THAN ('P')
      TABLESPACE construction_part2_ndx_ts,
    PARTITION part3 VALUES LESS THAN (MAXVALUE)
      TABLESPACE construction_part3_ndx_ts);
```

If in fact the **housing_construction** table used a composite partition, we could accommodate that here:

```
CREATE UNIQUE CLUSTERED INDEX project_id_ind
  ON housing_construction(project_id)
  STORAGE (INITIAL 10M MAXEXTENTS UNLIMITED)
  LOCAL (PARTITION part1 TABLESPACE construction_part1_ndx_ts,
    PARTITION part2 TABLESPACE construction_part2_ndx_ts
      (SUBPARTITION subpart10, SUBPARTITION subpart20,
```

```

        SUBPARTITION subpart30, SUBPARTITION subpart40,
        SUBPARTITION subpart50, SUBPARTITION subpart60),
    PARTITION part3 TABLESPACE construction_part3_ndx_ts);

```

In the following example, we rebuild the `project_id_ind` index that was created earlier by using parallel execution processes to scan the old index and build the new index in reverse order:

```

ALTER INDEX project_id_ind
REBUILD REVERSE PARALLEL;

```

Similarly, we can split out an additional partition on `project_id_ind`:

```

ALTER INDEX project_id_ind
SPLIT PARTITION part3 AT ('S')
INTO (PARTITION part3_a TABLESPACE constr_p3_a LOGGING,
      PARTITION part3_b TABLESPACE constr_p3_b);

```

PostgreSQL

PostgreSQL allows the creation of ascending-order indexes, as well as *UNIQUE* indexes. Its implementation also includes a performance enhancement under the `USING` clause. PostgreSQL's `CREATE INDEX` syntax is:

```

CREATE [UNIQUE] INDEX [CONCURRENTLY] index_name ON table_name
[USING {BTREE | RTREE | HASH | GIST}]
{function_name | (column_name[, ...])}
[WITH FillFactor = int]
[TABLESPACE tablespace_name]
[WHERE condition]

```

and the syntax for `ALTER INDEX` is:

```

ALTER INDEX index_name
[RENAME TO new_index_name]
[SET TABLESPACE new_tablespace_name]
[SET FillFactor = int]
[RESET FillFactor = int]

```

where:

CONCURRENTLY

Builds the index without acquiring any locks that would prevent concurrent inserts, updates, or deletes on the table. Normally, PostgreSQL locks the table to writes (but *not* reads) until the operation completes. Building an index concurrently is *not recommended* unless you can be sure that the table is not being used for write activity.

USING {BTREE | RTREE | HASH | GIST}

Specifies one of three dynamic access methods to optimize performance. Of particular importance is the fact that the indexes are fully dynamic and do not require periodic refreshes of their statistics. The `USING` options are as follows:

BTREE

Uses Lehman-Yao's high-concurrency B-tree structures to optimize the index. This is the default method when no other is specified. B-tree indexes can be invoked for comparisons using `=`, `<`, `<=`, `>`, and `>=`. B-tree indexes can be multicolumn.

RTREE

Uses Guttman's quadratic-split algorithm R-tree structures to optimize the index. R-tree indexes can be invoked for comparison using `<<`, `&<`, `&>`, `>>`, `@`, `~`, and `&&`. R-tree indexes must be single-column indexes. This keyword is deprecated and has been removed from the latest version of PostgreSQL. If this value is declared, PostgreSQL will interpret it as `USING GIST`.

HASH

Uses Litwin's linear hashing algorithm to optimize the index. Hash indexes can be invoked for comparisons using `=`. Hash indexes must be single-column indexes.

GIST

Uses Generalized Index Search Trees (GISTS) to optimize the index. GIST indexes may be multicolumn.

function_name

Defines a user-defined function, rather than a column from the base table, as the basis of the index values. A function-based index is mutually exclusive of a regular column-based index.

WITH FILLFACTOR = *int*

Defines a percentage for PostgreSQL to fill each index page during the creation process. For B-tree indexes, this is during the initial index creation process and when extending the index. The default is 90. PostgreSQL does not maintain the fillfactor over time, so it is advisable to rebuild the index at regular intervals to avoid excessive fragmentation and page splits.

TABLESPACE *tablespace_name*

Defines the tablespace where the index is created.

WHERE *condition*

Defines a `WHERE` clause search condition, which is then used to generate a *partial* index. A partial index contains entries for a select set of records in the table, not all records. You can get some interesting effects from this clause. For example, you can pair `UNIQUE` and `WHERE` to enforce uniqueness for a subset of the table rather than for the whole table. The `WHERE` clause must:

- Reference columns in the base table (though they need not be columns of the index itself).
- Not make use of aggregate functions.
- Not use subqueries.

[RENAME TO *new_index_name*] [SET TABLESPACE *new_tablespace_name*] [SET FILLFACTOR = *int*] [RESET FILLFACTOR]

Allows you to alter the properties of an existing index, for example to rename the index, specify a new tablespace for the index, specify a new fillfactor for the index, or reset the fillfactor to its default value. Note that for `SET FILLFACTOR` and `RESET FILLFACTOR`, we recommend that you rebuild the index with the `REINDEX` command because changes do not immediately take effect.

In PostgreSQL, a column may have an associated *operator class* based on the datatype of the column. An operator class specifies the operators for a particular index. Although users are free to define any valid operator class for a given column, the default operator class is the appropriate operator class for that column type.

In the following example, we create an index using the *GIST* indextype and make sure that uniqueness is enforced only for publishers outside of the USA:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(pub_name, country)
  USING GIST
  WHERE country <> 'USA';
```

SQL Server

SQL Server's *CREATE INDEX* syntax is:

```
CREATE [UNIQUE] [[NON]CLUSTERED] INDEX index_name
  ON {table_name | view_name} (column [ASC | DESC][, ...])
  [INCLUDE (column [ASC | DESC][, ...])]
  [WITH [PAD_INDEX = {ON | OFF}] [FILLCODE = int] [IGNORE_DUP_KEY = {ON | OFF}]
    [STATISTICS_NORECOMPUTE = {ON | OFF}] [DROP_EXISTING = {ON | OFF}]
    [ONLINE = {ON | OFF}] [SORT_IN_TEMPDB = {ON | OFF}]
    [ALLOW_ROW_LOCKS = {ON | OFF}] [ALLOW_PAGE_LOCKS = {ON | OFF}]
    [MAXDOP = int][, ...]]
  [ON {filegroup | ptn (column) | DEFAULT}]
  [FILESTREAM_ON {filestream_filegroup_name | ptn | "NULL"}]
```

and the syntax for *ALTER INDEX* is:

```
ALTER INDEX {index_name | ALL} ON {object_name}
  { DISABLE |
    REBUILD [PARTITION = ptn_nbr] [WITH
      ( [ SORT_IN_TEMPDB = {ON | OFF} ][MAXDOP = int][, ...] )
      [WITH [PAD_INDEX = {ON | OFF}][FILLCODE = int]
        [IGNORE_DUP_KEY = {ON | OFF}]
        [STATISTICS_NORECOMPUTE = {ON | OFF}] [SORT_IN_TEMPDB = {ON | OFF}]
        [ALLOW_ROW_LOCKS = {ON | OFF}] [ALLOW_PAGE_LOCKS = {ON | OFF}]
        [MAXDOP = int][, ...]] |
      REORGANIZE [PARTITION = ptn_nbr] [WITH (LOB_COMPACTION = {ON | OFF})] |
      SET [ALLOW_ROW_LOCKS = {ON | OFF}] [ALLOW_PAGE_LOCKS = {ON | OFF}]
        [IGNORE_DUP_KEY = {ON | OFF}]
        [STATISTICS_NORECOMPUTE = {ON | OFF}][, ...] }
```

where:

/NON]CLUSTERED

Controls the physical ordering of data for the table using either a *CLUSTERED* or a *NONCLUSTERED* index. The columns of a *CLUSTERED* index determine the order in which the records of the table are physically written. Thus, if you create an ascending clustered index on column **A** of table **Foo**, the records will be written to disk in ascending alphabetical order. The *NONCLUSTERED* clause (the default when a value is omitted) creates a secondary index containing only pointers and has no impact on how the actual rows of the table are written to disk.

ASC | DESC

Specifies that the values in the index be kept in either ascending (*ASC*) or descending (*DESC*) order. When *ASC* or *DESC* is omitted, *ASC* is used by default.

WITH

Allows the specification of one or more optional attributes for the index.

PAD_INDEX = {ON | OFF}

Specifies that space should be left open on each 8K index page, according to the value established by the *FILLFACTOR* setting.

FILLFACTOR = *int*

Declares a percentage value, *int*, from 1 to 100 that tells SQL Server how much of each 8K data page should be filled at the time the index is created. This is useful to reduce I/O contention and page splits when a data page fills up. Creating a clustered index with an explicitly defined fillfactor can increase the size of the index, but it can also speed up processing in certain circumstances.

IGNORE_DUP_KEY = {ON | OFF}

Controls what happens when a duplicate record is placed into a unique index through an insert or update operation. If this value is set for a column, only the duplicate row is excluded from the operation. If this value is not set, all records in the operation (even nonduplicate records) are rejected as duplicates.

DROP_EXISTING = {ON | OFF}

Drops any pre-existing indexes on the table and rebuilds the specified index.

STATISTICS_NORECOMPUTE = {ON | OFF}

Stops SQL Server from recomputing index statistics. This can speed up the *CREATE INDEX* operation, but it may mean that the index is less effective.

ONLINE = {ON | OFF}

Specifies whether underlying tables and associated indexes are available for queries and data-manipulation statements during the index operation. The default is *OFF*. When set to *ON*, long-term write locks are not held, only shared locks.

SORT_IN_TEMPDB = {ON | OFF}

Stores any intermediate results used to build the index in the system database, *TEMPDB*. This increases the space needed to create the index, but it can speed processing if *TEMPDB* is on a different disk than the table and index.

ALLOW_ROW_LOCKS = {ON | OFF}

Specifies whether row locks are allowed. When omitted, the default is *ON*.

ALLOW_PAGE_LOCKS = {ON | OFF}

Specifies whether page locks are allowed. When omitted, the default is *ON*.

MAXDOP = *int*

Specifies the maximum degrees of parallelism for the duration of the indexing operation. 1 suppresses parallelism. A value greater than 1 restricts the operation to the number of processors specified. 0, the default, allows SQL Server to choose up to the actual number of processors on the system.

ON *filegroup*

Creates the index on a given pre-existing filegroup. This enables the placing of indexes on a specific hard disk or RAID device. Issuing a *CREATE CLUSTERED INDEX...ON FILEGROUP* statement effectively moves a table to the new filegroup, since the leaf level of the clustered index is the same as the actual data pages of the table.

DISABLE

Disables the index, making it unavailable for use in query execution plans. Disabled nonclustered indexes do not retain underlying data in the index pages. Disabling a clustered index makes the underlying table unavailable to user access. You can re-enable an index with *ALTER INDEX REBUILD* or *CREATE INDEX WITH DROP_EXISTING*.

REBUILD [PARTITION = *prtn_nbr*]

Rebuilds an index using the pre-existing properties, including columns in the index, indextype, uniqueness attributes, and sort order. You may optionally specify a new partition. This clause will *not* automatically rebuild associated nonclustered indexes unless you include the keyword *ALL*. When using this clause to rebuild an XML or spatial index, you may not also use the *ONLINE = ON* or *IGNORE_DUP_KEY = ON* clauses. Equivalent to *DBCC DBREINDEX*.

REORGANIZE [PARTITION = *prtn_nbr*]

Performs an online reorganization of the leaf level of the index (i.e., no long-term blocking table locks are held and queries and updates to the underlying table can continue). You may optionally specify a new partition. Not allowed with *ALLOW_PAGE_LOCKS=OFF*. Equivalent to *DBCC INDEXDEFRAG*.

WITH (LOB_COMPACTION = {ON | OFF})

Compacts all pages containing *LOB* datatypes, including *IMAGE*, *TEXT*, *NTEXT*, *VARCHAR(MAX)*, *NVARCHAR(MAX)*, *VARBINARY(MAX)*, and *XML*. When omitted, the default is *ON*. The clause is ignored if no *LOB* columns are present. When *ALL* is specified, all indexes associated with the table or view are reorganized.

SET

Specifies index options without rebuilding or reorganizing the index. *SET* cannot be used on a disabled index.

SQL Server allows the creation of a unique clustered index on a view, effectively materializing the view. This can greatly speed up data-retrieval operations against the view. Once a view has a unique clustered index, nonclustered indexes can be added to the view. Note that the view also must be created using the *SCHEMABINDING* option. Indexed views are allowed only in SQL Server 2000 Enterprise Edition, unless you add a *NOEXPAND* hint to the view. Indexed views support data retrieval, but not data modification.

SQL Server allows up to 249 nonclustered indexes (unique or non-unique) on a table, as well as one primary key index. Index columns may not be of the datatypes *NTEXT*, *TEXT*, or *IMAGE*.

SQL Server automatically parallelizes the creation of the index according to the configuration option *max degree of parallelism*.

It is often necessary to build indexes that span several columns—i.e., a *concatenated key*. Concatenated keys may contain up to 16 columns and/or a total of 900 bytes across all fixed-length columns. Here is an example:

```
CREATE UNIQUE INDEX project2_ind
  ON housing_construction(project_name, project_date)
  WITH PAD_INDEX, FILLFACTOR = 80
  ON FILEGROUP housing_fg
  GO
```

Adding the *PAD_INDEX* clause and setting the *FILLCATOR* to 80 tells SQL Server to leave the index and data pages 80% full, rather than 100% full. This example also tells SQL Server to create the index on the **housing_fg** filegroup, rather than the default filegroup.

See Also

[CREATE/ALTER TABLE](#)
[DROP](#)

CREATE/ALTER METHOD Statement

The *CREATE/ALTER METHOD* statements allow the creation of a new database method or the alteration of an already existing database method. An easy (but loose) way to think of a method is that it is a user-defined function associated with a user-defined type. For example, a method called *Office*, of type *Address*, can accept a *VARCHAR* input parameter and passes out a result *Address*.

An implicitly defined method is created every time a structured type is created. Using a combination of the *CREATE TYPE* statement and the *CREATE METHOD* statement creates user-defined methods.

Platform	Command
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
{CREATE | ALTER} [INSTANT | STATIC] METHOD method_name
  ( [{IN | OUT | INOUT}] param datatype [AS LOCATOR] [RESULT][, ...] )
  RETURNS datatype
  FOR udt_name
  [SPECIFIC specific_name] code_body
```

Keywords

{*CREATE | ALTER*} [*INSTANT | STATIC*] *method_name*

Creates a new method or alters an existing method and, optionally, specifies it as an *INSTANT* or *STATIC* method.

({*IN | OUT | INOUT*} *param datatype*[, ...])

Declares one or more parameters to be passed into the method in a comma-delimited list enclosed in parentheses. Parameters used with a method may pass a value *IN*, *OUT*, or both in and out via *INOUT*. The syntax for the parameter declaration is:

```
[{IN | OUT | INOUT}] parameter_name1 datatype,
[ {IN | OUT | INOUT}] parameter_name2 datatype,[...]
```

Make sure the name is unique within the method. When used with *ALTER*, this clause adds parameters to a pre-existing method. Refer to Chapter 2 for details on datatypes.

AS LOCATOR

The optional *AS LOCATOR* clause is used to validate an external routine with *RETURNS* parameter that is a *BLOB*, *CLOB*, *NCLOB*, *ARRAY*, or user-defined type. In other words, the locator (i.e., a pointer) for the *LOB* is returned, but not the entire value of the *LOB*.

RESULT

Designates a user-defined type. Not needed for standard datatypes.

RETURNS datatype

Declares the datatype of the results returned by a method. The key purpose of a user-defined method is to return a value. If you need to change the datatype of a *RETURNS* parameter on the fly, use the *CAST* clause (refer to the function *CAST*); for example, *RETURNS VARCHAR(12) CAST FROM DATE*.

FOR udt_name

Associates the method with a specific, pre-existing user-defined type, created using *CREATE TYPE*.

SPECIFIC specific_name

Uniquely identifies the function; generally used with user-defined types.

Rules at a Glance

User-defined methods are essentially a different approach to obtaining the same output provided by user-defined functions. For example, consider the following two pieces of code:

```
CREATE FUNCTION my_fcn (order_udt)
RETURNS INT;
CREATE METHOD my_mthd ()
RETURNS INT
FOR order_udt;
```

Although the code sections for the function and method are different, they do exactly the same thing.

The rules for use and invocation of methods are otherwise the same as for functions.

Programming Tips and Gotchas

The main difficulty with *CREATE METHOD* statements is that they are an object-oriented approach to the same sort of functionality provided by user-defined functions. Since they accomplish the same work using a different approach, it can be hard to decide which approach to use.

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

Not supported.

See Also

[CREATE/ALTER TYPE](#)

CREATE ROLE Statement

CREATE ROLE allows the creation of a named set of privileges that may be assigned to users of a database. When a role is granted to a user, that user gets all the privileges and permissions of that role. Roles are generally accepted as one of the best means for maintaining security and controlling privileges within a database.

Platform	Command
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

`CREATE ROLE role_name [WITH ADMIN {CURRENT_USER | CURRENT_ROLE}]`

Keywords

CREATE ROLE *role_name*

Creates a new role and differentiates that role from a host DBMS user and other roles. A role can be assigned any permission that a user can be assigned. The important difference is that a role can then be assigned to one or more users, thus giving them all the permissions of that role.

WITH ADMIN {CURRENT_USER | CURRENT_ROLE}

Assigns a role immediately to the currently active user or currently active role along with the privilege to pass the use of the role on to other users. By default, the statement defaults to *WITH ADMIN CURRENT_USER*.

Rules at a Glance

Using roles for database security can greatly ease administration and user maintenance. The general steps for using roles in database security are:

1. Assess the needs for roles and pick the role names (e.g., *administrator*, *manager*, *data_entry*, *report_writer*, etc.).
2. Assign permissions to each role as if it were a standard database user, using the *GRANT* statement. For example, the *manager* role might have permission to read from and write to all user tables in a database, while the *report_writer* role might only have permission to execute read-only reports against certain reporting tables in the database.
3. Use the *GRANT* statement to grant roles to users of the system according to the types of work they will perform.

Permissions can be disabled using the *REVOKE* command.

Programming Tips and Gotchas

The main problem with roles is that occasionally a database administrator will provide redundant permissions to a role and separately to a user. If you ever need to prevent a user's access to a resource in situations like this, you usually will need to *REVOKE* the permissions twice: the role must be revoked from the user, and then the specific user-level privilege must also be revoked from the user.

MySQL

Not supported.

Oracle

Although it is not currently supported by the ANSI standard, Oracle also offers an *ALTER ROLE* statement. Oracle supports the concept of roles, though its implementation of the commands is very different from the ANSI SQL standard:

```
{CREATE | ALTER} ROLE role_name
  [NOT IDENTIFIED |
   IDENTIFIED {BY password | EXTERNALLY | GLOBALLY |
   USING package_name}]
```

where:

{CREATE | ALTER} ROLE *role_name*

Identifies the name of the new role being created or the pre-existing role being modified.

NOT IDENTIFIED

Declares that no password is needed for the role to receive authorization to the database. This is the default.

IDENTIFIED

Declares that the users of the role must authenticate themselves by the method indicated before the role is enabled via the *SET ROLE* command, where:

BY password

Creates a local role authenticated by the string value of *password*. Only single-byte characters are allowed in the password, even when using a multi-byte character set.

EXTERNALLY

Creates an external role that is authenticated by the operating system or a third-party provider. In either case, the external authentication service will likely require a password.

GLOBALLY

Creates a global role that is authenticated by an enterprise directory service, such as an LDAP directory.

USING package_name

Creates an application role that enables a role only through an application that uses a PL/SQL package of *package_name*. If you omit the schema, Oracle assumes that the package is in your schema.

In Oracle, the role is created first, then granted privileges and permissions as if it is a user via the *GRANT* command. When users want to get access to the permissions of a role protected by a password, they use the *SET ROLE* command. If a password is placed on the role, any user wishing to access it must provide the password with the *SET ROLE* command.

Oracle ships with several preconfigured roles. *CONNECT*, *DBA*, and *RESOURCE* are available in all versions of Oracle. *EXP_FULL_DATABASE* and *IMP_FULL_DATABASE* are newer roles used for import and export operations. The *GRANT* statement reference has a more detailed discussion of all of the preconfigured roles available in Oracle.

The following example uses *CREATE* to specify a new role in Oracle, *GRANTS* it privileges, assigns it a password with *ALTER ROLE*, and *GRANTS* the new role to a couple of users:

```
CREATE ROLE boss;
GRANT ALL ON employee TO boss;
GRANT CREATE SESSION, CREATE DATABASE LINK TO boss;
ALTER ROLE boss IDENTIFIED BY le_grand_fromage;
GRANT boss TO emily, jake;
```

PostgreSQL

PostgreSQL supports both the *ALTER* and *CREATE ROLE* statements, and it offers a nearly identical extension of its own called *ALTER/CREATE GROUP*. The syntax for *CREATE ROLE* follows:

```
{CREATE | ALTER} ROLE name
[ [WITH] [[NO]SUPERUSER] [[NO]CREATEDB] [[NO]CREATEUSER] [[NO]INHERIT]
[[NO]LOGIN]
[ [CONNECTION LIMIT int] [ {ENCRYPTED | UNENCRYPTED} PASSWORD 'password' ]
[VALID UNTIL 'date_and_time' ] [IN ROLE rolename[, ...]]
[IN GROUP groupname[, ...]] [ROLE rolename[, ...]]
[ADMIN rolename[, ...]] [USER rolename[, ...]] [SYSID int][...]
[RENAME TO new_name]
[SET parameter {TO | =} {value | DEFAULT}]
[RESET parameter]
```

where:

*{CREATE | ALTER} ROLE *name**

Specifies the new role to create or the existing role to modify, where *name* is the name of the role to create or modify.

[NO]SUPERUSER

Specifies whether the role is a superuser or not. The superuser may override all access restrictions within the database. *NOSUPERUSER* is the default.

[NO]CREATEDB

Specifies whether the role may create databases or not. *NOCREATEDB* is the default.

[NO]CREATEROLE

Specifies whether the role may create new roles and alter or drop other roles. *NOCREATEROLE* is the default.

[NO]CREATEUSER

Specifies whether the role may create a superuser. This clause is deprecated in favor of *[NO]SUPERUSER*.

[NO]INHERIT

Specifies whether the role inherits the privileges of the roles of which it is a member. A role with *INHERIT* automatically may use the privileges that are granted to the roles that of which it is (directly or indirectly) a member. *INHERIT* is the default.

[NO]LOGIN

Specifies whether the role may log in. With *LOGIN*, a role is essentially a user. With *NOLOGIN*, a role provides mapping to specific database privileges but is not an actual user. The default is *NOLOGIN*.

CONNECTION LIMIT *int*

Specifies how many concurrent connections a role can make, if it has *LOGIN* privileges. The default is -1; that is, unlimited.

{ENCRYPTED | UNENCRYPTED} PASSWORD '*password*'

Sets a password for the role, if it has *LOGIN* privileges. The password may be stored in plain text (*UNENCRYPTED*) or encrypted in MD5-format in the system catalogs (*ENCRYPTED*). Older clients may not support MD5 authentication, so be careful.

VALID UNTIL '*date_and_time*'

Sets a date and time when the role's password will expire, if it has *LOGIN* privileges. When omitted, the default is no time limit.

IN ROLE, IN GROUP

Specifies one or more existing roles (or groups, though this clause is deprecated) of which the role is a member.

ROLE, GROUP

Specifies one or more existing roles (or groups, though this clause is deprecated) that are automatically added as members of the new or modified role.

ADMIN *rolename*

Similar to the *ROLE* clause, except new roles are added *WITH ADMIN OPTION*, giving them the right to grant membership in this role to others.

SYSID *int*, USER *username*

Deprecated clauses that are still accepted for backward compatibility. *USER* is equivalent to the *ROLE* clause and *SYSID* is equivalent to the *GROUP* clause.

[RENAME TO *new_name*] [SET *parameter* {TO | =} {*value* | DEFAULT}] [RESET *parameter*]

Renames an existing role to a new name, sets a configuration parameter, or resets a configuration parameter to the default value. Configuration parameters are fully detailed within PostgreSQL's documentation.

Use the *DROP ROLE* clause to drop a role you no longer want.

SQL Server

Microsoft SQL Server supports the *ALTER* and *CREATE ROLE* statements, as well as equivalent capabilities via the system stored procedure **sp_add_role**. SQL Server's syntax follows:

```
CREATE ROLE role_name [AUTHORIZATION owner_name]
    [WITH NAME = new_role_name]
```

where:

AUTHORIZATION *owner_name*

Specifies the database user or role that owns the new role. When omitted, the role is owned by the user that created it. Only used with *CREATE ROLE*.

WITH NAME = *new_role_name*

Specifies the new name of the role. Only used with *ALTER ROLE*.

See Also

GRANT
REVOKE

CREATE SCHEMA Statement

This statement creates a *schema*—i.e., a named group of related objects. A schema is a collection of tables, views, and permissions granted to specific users or roles. According to the ANSI standard, specific object permissions are not schema objects in themselves and do not belong to a specific schema. However, roles are sets of privileges that do belong to a schema.

Platform	Command
MySQL	Supported (as <i>CREATE DATABASE</i>)
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
CREATE SCHEMA [schema_name] [AUTHORIZATION owner_name]
  [DEFAULT CHARACTER SET char_set_name]
  [PATH schema_name[, ...]]
  [ ANSI CREATE statements [...] ]
  [ ANSI GRANT statements [...] ]
```

Keywords

CREATE SCHEMA [*schema_name*]

Creates a schema called *schema_name*. When omitted, the database will create a schema name for you using the name of the user who owns the schema.

AUTHORIZATION *owner_name*

Specifies the user who will be the owner of the schema. When this clause is omitted, the current user is set as the owner. The ANSI standard allows you to omit either the *schema_name* or the *AUTHORIZATION* clause, or to use them both together.

DEFAULT CHARACTER SET *char_set_name*

Declares a default character set of *char_set_name* for all objects created within the schema.

PATH *schema_name*[, ...]

Optionally declares a file path and filename for any unqualified routines (i.e., stored procedures, user-defined functions, user-defined methods) in the schema.

ANSI CREATE statements [...]

Contains one or more *CREATE* statements. No commas are used between the *CREATE* statements.

ANSI GRANT statements [...]

Contains one or more *GRANT* statements that apply to previously defined objects. Usually, the objects were created earlier in the same *CREATE SCHEMA* statement, but they may be pre-existing objects. No commas are used between the *GRANT* statements.

Rules at a Glance

The *CREATE SCHEMA* statement is a container that can hold many other *CREATE* and *GRANT* statements. The most common way to think of a schema is as all of the objects that a specific user owns. For example, the user *jake* may own several tables and views in his own schema, including a table called **publishers**. Meanwhile, the user *dylan* may own several other tables and views in his schema, but may also own his own separate copy of the **publishers** table.

The ANSI standard requires that all *CREATE* statements are allowed in a *CREATE SCHEMA* statement. In practice, however, most implementations of *CREATE SCHEMA* allow only three subordinate statements: *CREATE TABLE*, *CREATE VIEW*, and *GRANT*. The order of the commands is not important, meaning that (although it is not recommended) you can grant privileges on tables or views whose *CREATE* statements appear later in the *CREATE SCHEMA* statement.

Programming Tips and Gotchas

It is not required, but it is considered a best practice to arrange the objects and grants within a *CREATE SCHEMA* statement in an order that will execute naturally. In other words, *CREATE VIEW* statements should follow the *CREATE TABLE* statements that they depend on, and the *GRANT* statements should come last.

If your database system uses schemas, we recommend that you always reference your objects by schema and then object name (as in *jake.publishers*). If you do not include a schema qualifier, the database platform will typically assume the default schema for the current user connection.

Some database platforms do not explicitly support the *CREATE SCHEMA* command. However, they implicitly create a schema when a user creates database objects. For example, Oracle creates a schema whenever a user is created. The *CREATE SCHEMA* command is simply a single-step method of creating all the tables, views, and other database objects along with their permissions.

MySQL

MySQL supports the *CREATE SCHEMA* statement as a synonym of the *CREATE DATABASE* statement. Refer to that section for more information on MySQL's implementation.

Oracle

In Oracle, the *CREATE SCHEMA* statement does not actually create a schema; only the *CREATE USER* statement does that. What *CREATE SCHEMA* does is allow a user to perform multiple *CREATEs* and *GRANTs* in a previously created schema in one SQL statement:

```
CREATE SCHEMA AUTHORIZATION schema_name
  [ ANSI CREATE statements [...] ]
  [ ANSI GRANT statements [...] ]
```

Note that Oracle only allows ANSI-standard *CREATE TABLE*, *CREATE VIEW*, and *GRANT* statements within a *CREATE SCHEMA* statement. You *should not* use any of Oracle's extensions to these commands when using the *CREATE SCHEMA* statement.

The following Oracle example places the permissions before the objects within the *CREATE SCHEMA* statement:

```
CREATE SCHEMA AUTHORIZATION emily
  GRANT SELECT, INSERT ON view_1 TO sarah
  GRANT ALL ON table_1 TO sarah
```

```

CREATE VIEW view_1 AS
  SELECT column_1, column_2
  FROM table_1
  ORDER BY column_2
CREATE TABLE table_1(column_1 INT, column_2 CHAR(20));

```

As this example shows, the order of the statements within the *CREATE SCHEMA* statement is unimportant; Oracle commits the *CREATE SCHEMA* statement only if all *CREATE* and *GRANT* statements in the statement complete successfully.

PostgreSQL

PostgreSQL supports both *ALTER* and *CREATE SCHEMA* statements without support for the *PATH* and *DEFAULT CHARACTER SET* clauses. The *CREATE SCHEMA* syntax follows:

```

CREATE SCHEMA { schema_name [AUTHORIZATION user_name] | AUTHORIZATION user_name }
  [ ANSI CREATE statements [...] ]
  [ ANSI GRANT statements [...] ]

```

When the *schema_name* is omitted, the *user_name* is used to name the schema. Currently, PostgreSQL supports only the following *CREATE* statements within a *CREATE SCHEMA* statement: *CREATE TABLE*, *CREATE VIEW*, *CREATE INDEX*, *CREATE SEQUENCE*, and *CREATE TRIGGER*. Other *CREATE* statements must be handled separately from the *CREATE SCHEMA* statement.

The *ALTER SCHEMA* syntax follows:

```
ALTER SCHEMA schema_name [RENAME TO new_schema_name] [OWNER TO new_user_name]
```

The *ALTER SCHEMA* statement allows you to rename a schema or to specify a new owner, who must be a pre-existing user on the database.

SQL Server

SQL Server supports the basic *CREATE SCHEMA* statement, without support for the *PATH* clause or the *DEFAULT CHARACTER SET* clause:

```

CREATE SCHEMA AUTHORIZATION owner_name
  [ ANSI CREATE statements [...] ]
  [ ANSI GRANT statements [...] ]

```

If any statement fails within the *CREATE SCHEMA* statement, the entire statement fails.

SQL Server does not require that the *CREATE* or *GRANT* statements be in any particular order, except that nested views must be declared in their logical order. That is, if *view_100* references *view_10*, *view_10* must appear in the *CREATE SCHEMA* statement before *view_100*.

For example:

```

CREATE SCHEMA AUTHORIZATION katie
GRANT SELECT ON view_10 TO public
CREATE VIEW view_10(col1) AS SELECT col1 FROM foo
CREATE TABLE foo(col1 INT)
CREATE TABLE foo
  (col1 INT PRIMARY KEY,
  col2 INT REFERENCES foo2(col1))

```

CREATE/ALTER TABLE Statement

```
CREATE TABLE foo
  (col1 INT PRIMARY KEY,
   col2 INT REFERENCES foo(col1));
```

The syntax for *ALTER ROLE* follows:

```
ALTER ROLE owner_name WITH NAME = new_owner_name;
```

The *ALTER ROLE* statement merely allows the assignment of a new owner to an existing schema.

See Also

[CREATE/ALTER TABLE](#)
[CREATE/ALTER VIEW](#)
[GRANT](#)

CREATE/ALTER TABLE Statement

Manipulating tables is one of the most common activities that database administrators and programmers perform when working with database objects. This section details how to create and modify tables.

The ANSI standard represents a sort of least common denominator among the vendors. Although not all vendors offer every option of the ANSI-standard version of *CREATE TABLE* and *ALTER TABLE*, the ANSI standard does represent the basic form that can be used across all of the platforms. Conversely, the vendor platforms offer a variety of extensions and additions to the ANSI standards for *CREATE* and *ALTER TABLE*.



Typically, a great deal of consideration goes into the design and creation of a table. This discipline is known as *database design*. The discipline of analyzing the relationship of a table to its own data and to other tables within the database is known as *normalization*. We recommend that database programmers and administrators alike study both database design and normalization principles thoroughly before issuing *CREATE TABLE* commands.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 statement *CREATE TABLE* creates a permanent or temporary table within the database where the command is issued. The syntax is as follows:

```
CREATE [{LOCAL TEMPORARY| GLOBAL TEMPORARY}] TABLE table_name
  (column_name datatype attributes[, ...]) |
  [column_name WITH OPTIONS options] |
  [LIKE table_name] |
```

```
[REF IS column_name
  {SYSTEM GENERATED | USER GENERATED | DERIVED}]
[CONSTRAINT constraint_type [constraint_name][, ...]]
[OF type_name [UNDER super_table] [table_definition]]
[ON COMMIT {PRESERVE ROWS | DELETE ROWS}]
```

The SQL2003 statement *ALTER TABLE* allows many useful modifications to be made to an existing table without dropping any existing indexes, triggers, or permissions assigned to it. Following is the *ALTER TABLE* syntax:

```
ALTER TABLE table_name
[ADD [COLUMN] column_name datatype attributes]
| [ALTER [COLUMN] column_name SET DEFAULT default_value]
| [ALTER [COLUMN] column_name DROP DEFAULT]
| [ALTER [COLUMN] column_name ADD SCOPE table_name]
| [ALTER [COLUMN] column_name DROP SCOPE {RESTRICT | CASCADE}]
| [DROP [COLUMN] column_name {RESTRICT | CASCADE}]
| [ADD table_constraint]
| [DROP CONSTRAINT table_constraint_name {RESTRICT | CASCADE}]
```

Keywords

CREATE {[LOCAL TEMPORARY | GLOBAL TEMPORARY]} TABLE

Declares a permanent table or a TEMPORARY table of LOCAL or GLOBAL scope. *Local temporary tables* are accessible only to the session that created them and are automatically dropped when that session terminates. *Global temporary tables* are accessible by all active sessions but are also automatically dropped when the session that created them terminates. Do not qualify a temporary table with a schema name.

(*column_name datatype attributes*[, ...])

Defines a comma-delimited list of one or more columns, their datatypes, and any additional attributes (such as nullability). Every table declaration must contain at least one column, which may include:

column_name

Specifies a name for a column. The name must be a valid identifier according to the rules of the specific DBMS. Make sure the name makes sense!

datatype

Associates a specific datatype with the column identified by *column_name*. An optional *length* may be specified for datatypes that allow user-defined lengths, for example *VARCHAR(255)*. It must be a valid datatype on the specific DBMS. Refer to Chapter 2 for a full discussion of acceptable datatypes and vendor variations.

attributes

Associates specific constraint attributes with the *column_name*. Many attributes may be applied to a single *column_name*, no commas required. Typical ANSI attributes include:

NOT NULL

Tells the column to reject NULL values or, when omitted, to accept them. Any *INSERT* or *UPDATE* statement that attempts to place a NULL value in a NOT NULL column will fail and roll back.

DEFAULT *expression*

Tells the column to use the value of *expression* when no value is supplied by an *INSERT* or *UPDATE* statement. The expression must be acceptable according to the datatype of the column; for example, no alphabetic characters may be used in an *INTEGER* column. *expression* may be a string or numeric literal, but you may also define a user-defined function or system function. SQL2003 allows these system functions in a *DEFAULT* expression: *NULL*, *USER*, *CURRENT_USER*, *SESSION_USER*, *SYSTEM_USER*, *CURRENT_PATH*, *CURRENT_DATE*, *CURRENT_TIME*, *LOCALTIME*, *CURRENT_TIMESTAMP*, *LOCALTIMESTAMP*, *ARRAY*, or *ARRAY[]*.

COLLATE *collation_name*

Defines a specific collation, or sort order, for the column to which it is assigned. The name of the collation is platform-dependent. If you do not define a collation, the columns of the table default to the collation of the character set used for the column.

REFERENCES ARE [NOT] CHECKED [ON DELETE {RESTRICT | SET NULL}]

Indicates whether references are to be checked on a *REF* column defined with a scope clause. The optional *ON DELETE* clause tells whether any values in records referenced by a deleted record should set to *NULL*, or whether the operation should be restricted.

CONSTRAINT *constraint_name* [*constraint_type* [*constraint*]]

Assigns a constraint and, optionally a constraint name, to the specific column. Constraint types are discussed in Chapter 2. Because the constraint is associated with a specific column, the constraint declaration assumes that the column is the only one in the constraint. After the table is created, the constraint is treated as a table-level constraint.

***column_name* [WITH OPTIONS *options*]**

Defines a column with special options, such as a scope clause, a default clause, a column-level constraint, or a *COLLATE* clause. For many implementations, the *WITH OPTIONS* clause is restricted to the creation of typed tables.

LIKE *table_name*

Creates a new table with the same column definitions as the pre-existing table named *table_name*.

REF IS *column_name* {SYSTEM GENERATED | USER GENERATED | DERIVED}

Defines the object identifier column (OID) for a typed table. An OID is necessary for the root table of a table hierarchy. Based on the option specified, the *REF* might be automatically generated by the system (SYSTEM GENERATED), manually provided by the user when inserting the row (USER GENERATED), or derived from another *REF* (DERIVED). Requires the inclusion of a REFERENCES column attribute for *column_name*.

CONSTRAINT *constraint_type* [*constraint_name*][, ...]

Assigns one or more constraints to the table. This option is noticeably different from the *CONSTRAINT* option at the column level, because column-level constraints are assumed to apply only to the column with which they are associated. Table-level constraints, however, give the option of associating multiple columns with a constraint. For example, in a *sales* table you might wish to declare a unique constraint on a concatenated key of *store_id*, *order_id*, and *order_date*. This can only be done using a table-level constraint. Refer to Chapter 2 for a full discussion of constraints.

OF type_name [UNDER super_table] [table_definition]

Declares that the table is based upon a pre-existing user-defined type. In this situation, the table may have only a single column for each attribute of the structured type, plus an additional column, as defined in the *REF IS* clause. This clause is incompatible with the *LIKE table_name* clause.

[UNDER super_table] [table_definition]

Declares the direct supertable of the currently declared table within the same schema, if any exists. The table being created is thus a direct subtable of the supertable. You may optionally provide a complete *table_definition* of the new subtable, complete with columns, constraints, etc.

ON COMMIT {PRESERVE ROWS | DELETE ROWS}

ON COMMIT PRESERVE ROWS preserves data rows in a temporary table on issuance of a *COMMIT* statement. *ON COMMIT DELETE ROWS* deletes all data rows in a temporary table on *COMMIT*.

ADD [COLUMN] column_name datatype attributes

Adds a column to a table, along with the appropriate datatype and attributes.

ALTER [COLUMN] column_name SET DEFAULT default_value

Adds a default value to the column if none exists, and resets the default value if a previous one exists.

ALTER [COLUMN] column_name DROP DEFAULT

Completely removes a default from the named column.

ALTER [COLUMN] column_name ADD SCOPE table_name

Adds a *scope* to the named column. A scope is a reference to a user-defined datatype.

ALTER [COLUMN] column_name DROP SCOPE {RESTRICT | CASCADE}

Drops a scope from the named column. The *RESTRICT* and *CASCADE* clauses are defined at the end of this list.

DROP COLUMN column_name {RESTRICT | CASCADE}

Drops the named column from the table.

ADD table_constraint

Adds a table constraint of the specified name and characteristics to the table.

DROP CONSTRAINT constraint_name {RESTRICT | CASCADE}

Drops a previously defined constraint from the table.

RESTRICT

Tells the DBMS to abort the command if it finds any other objects in the database that depend on the object.

CASCADE

Tells the DBMS to drop any other objects that depend on the object.

Rules at a Glance

The typical *CREATE TABLE* statement is very simple. Generally, it names the table and any columns contained in the table. Many table definitions also include a nullability constraint for most of the columns, as in this SQL Server example:

```
CREATE TABLE housing_construction
  (project_number      INT      NOT NULL,
   project_date        DATE     NOT NULL,
   project_name        VARCHAR(50) NOT NULL,
```

```
construction_color  NCHAR(20)  ,
construction_height  DECIMAL(4,1),
construction_length  DECIMAL(4,1),
construction_width  DECIMAL(4,1),
construction_volume  INT      )
```

This example adds a foreign key to the example table:

```
-- Creating a column-level constraint
CREATE TABLE favorite_books
  (isbn      CHAR(100)  PRIMARY KEY,
  book_name  VARCHAR(40) UNIQUE,
  category   VARCHAR(40) ,
  subcategory  VARCHAR(40) ,
  pub_date   DATETIME   NOT NULL,
  purchase_date DATETIME  NOT NULL,
  CONSTRAINT fk_categories FOREIGN KEY (category)
    REFERENCES category(cat_name));
```

The foreign key on the **categories** column relates it to the **cat_name** table in the **category** table. All the vendors discussed in this book support this syntax.



Examples for creating a table with each constraint type are shown in Chapter 2.

Similarly, the foreign key could be added after the fact as a multicolumn key including both the **category** and **subcategory** columns:

```
ALTER TABLE favorite_books ADD CONSTRAINT fk_categories
  FOREIGN KEY (category, subcategory)
    REFERENCES category(cat_name, subcat_name);
```

Now, we can use an *ALTER TABLE* statement to drop the constraint altogether:

```
ALTER TABLE favorite_books DROP CONSTRAINT fk_categories RESTRICT;
```

Listed here are more full examples from **pubs**, the sample database that ships with Microsoft SQL Server and Sybase Adaptive Server:

```
-- For a Microsoft SQL Server database
CREATE TABLE jobs
  (job_id  SMALLINT IDENTITY(1,1) PRIMARY KEY CLUSTERED,
  job_desc VARCHAR(50) NOT NULL DEFAULT 'New Position',
  min_lvl  TINYINT NOT NULL CHECK (min_lvl >= 10),
  max_lvl  TINYINT NOT NULL CHECK (max_lvl <= 250))
-- For a MySQL database
CREATE TABLE employee
  (emp_id INT AUTO_INCREMENT CONSTRAINT PK_emp_id PRIMARY KEY,
  fname VARCHAR(20) NOT NULL,
  minit CHAR(1) NULL,
  lname VARCHAR(30) NOT NULL,
  job_id SMALLINT NOT NULL DEFAULT 1
    REFERENCES jobs(job_id),
  job_lvl TINYINT DEFAULT 10,
  pub_id CHAR(4) NOT NULL DEFAULT ('9952')
    REFERENCES publishers(pub_id),
  hire_date DATETIME NOT NULL DEFAULT (CURRENT_DATE()));
```

```
CREATE TABLE publishers
  (pub_id char(4) NOT NULL
    CONSTRAINT UPKCL_pubind PRIMARY KEY CLUSTERED
    CHECK (pub_id IN ('1389', '0736', '0877', '1622', '1756')
    OR pub_id LIKE '99[0-9][0-9]'),
  pub_name varchar(40) NULL,
  city varchar(20) NULL,
  state char(2) NULL,
  country varchar(30) NULL DEFAULT('USA'))
```

Once you get into the vendor extensions, the *CREATE TABLE* command is no longer portable between database platforms. The following is an example of an Oracle *CREATE TABLE* statement with many storage properties:

```
CREATE TABLE classical_music_cds
  (music_id          INT,
  composition       VARCHAR2(50),
  composer          VARCHAR2(50),
  performer         VARCHAR2(50),
  performance_date DATE DEFAULT SYSDATE,
  duration          INT,
  cd_name           VARCHAR2(100),
  CONSTRAINT pk_class_cds PRIMARY KEY (music_id)
  USING INDEX TABLESPACE index_ts
  STORAGE (INITIAL 100K NEXT 20K),
  CONSTRAINT uq_class_cds UNIQUE
  (composition, performer, performance_date)
  USING INDEX TABLESPACE index_ts
  STORAGE (INITIAL 100K NEXT 20K))
  TABLESPACE tabledata_ts;
```

When issuing a *CREATE* or *ALTER* statement, we recommend that it be the only statement in your transaction. For example, do not attempt to create a table and select from it in the same batch. Instead, first create the table, then verify the operation, issue a *COMMIT*, and finally perform any subsequent operations against the table.

table_name is the name of a new or existing table. New table names should start with an alphabetic character and in general should contain no other special symbol besides the underscore (_). Rules for the length of the name and its exact composition differ according to the vendor.

When creating or altering a table, the list of column definitions is always encapsulated within parentheses, and the individual column definitions are separated by commas.

Programming Tips and Gotchas

The user issuing the *CREATE TABLE* command must have the appropriate permissions. Similarly, any user wishing to *ALTER* or *DROP* a table should own the table within his own schema or have adequate permissions to alter or drop the table. Since the ANSI standard does not specify the privileges required, expect some variation between vendors.

You can encapsulate a *CREATE TABLE* or *ALTER TABLE* statement within a transaction, using a *COMMIT* or *ROLLBACK* statement to explicitly conclude the transaction. We recommend that the *CREATE/ALTER TABLE* statement be the *only* command in the transaction.

Extensions to the ANSI standard can give you a great deal of control over the way that the records of a table are physically written to disk. SQL Server uses a technique called *clustered indexes* to control the way that records are written to disk. Oracle uses a technique that is functionally similar, called an *index-organized table*.

Some databases will lock a table that is being modified by an *ALTER TABLE* statement. It is therefore wise to issue this command only on tables that are not in use on a production database.

Furthermore, some databases will lock the target *and* source tables when using the *CREATE TABLE...LIKE* statement.

MySQL

The MySQL syntax for *CREATE TABLE* creates a permanent or local temporary table within the database in which the command is issued:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] table_name
  {column_name datatype attributes
   constraint_type constraint_name[, ...]
   [constraint_type [constraint_name][, ...]]
   [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
   [ON {DELETE | UPDATE} {RESTRICT | CASCADE | SET NULL | NO ACTION}]
   LIKE other_table_name}
  {[TABLESPACE tablespace_name STORAGE DISK] |
   [ENGINE = {ISAM | MyISAM | HEAP | BDB | InnoDB | MERGE | MRG_MyISAM}] |
   [AUTO_INCREMENT = int] |
   [AVG_ROW_LENGTH = int] |
   [ [DEFAULT] CHARACTER SET charset_name ] |
   [CHECKSUM = {0 | 1}] |
   [ [DEFAULT] COLLATE collation_name ] |
   [COMMENT = "string"] |
   [CONNECTION = 'connection_string'] |
   [DATA DIRECTORY = "path_to_directory"] |
   [DELAY_KEY_WRITE = {0 | 1}] |
   [INDEX DIRECTORY = "path_to_directory"] |
   [INSERT_METHOD = {NO | FIRST | LAST}] |
   [KEY_BLOCK_SIZE = int] |
   [MAX_ROWS = int] |
   [MIN_ROWS = int] |
   [PACK_KEYS = {0 | 1}] |
   [PASSWORD = "string"] |
   [ROW_FORMAT= { DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT |
   COMPACT }] [...]}
  [partition_definition[, ...]]
  [ [IGNORE | REPLACE] select_statement ]
```

In most cases, MySQL alters a table by creating a temporary work copy of the table, manipulating it, deleting the original, and finally renaming the copy. The MySQL syntax for *ALTER TABLE* allows modifications to a table or even renaming of a table:

```
ALTER [IGNORE] TABLE table_name
  { [ADD [COLUMN] (column_name datatype attributes)
   [FIRST | AFTER column_name][, ...]]
  | [ADD [CONSTRAINT] [UNIQUE | FOREIGN KEY | FULLTEXT | PRIMARY KEY | SPATIAL]
   [INDEX | KEY] [index_name] (index_col_name[, ...])]
  | [ALTER [COLUMN] column_name {SET DEFAULT literal | DROP DEFAULT}]}
```

```

| [CHANGE | MODIFY] [COLUMN] old_col_name new_column_name column_definition
| [FIRST | AFTER column_name]
| [DROP [COLUMN | FOREIGN KEY | PRIMARY KEY | INDEX | KEY] [object_name]]
| [{ENABLE | DISABLE} KEYS]
| [RENAME [TO] new_tbl_name]
| [ORDER BY column_name[, ...]]
| [CONVERT TO CHARACTER SET charset_name [COLLATE collation_name]]
| [{DISCARD | IMPORT} TABLESPACE]
| [{ADD | DROP | COALESCE int | ANALYZE | CHECK | OPTIMIZE | REBUILD | REPAIR}
  PARTITION]
| [REORGANIZE PARTITION prtn_name INTO (partition_definition)]
| [REMOVE PARTITIONING]
| [table_options] }[, ...]

```

Keywords and parameters are as follows:

TEMPORARY

Creates a table that persists for the duration of the connection under which it was created. Once that connection closes, the temporary table is automatically deleted.

IF NOT EXISTS

Prevents an error if the table already exists. A schema specification is not required.

constraint_type

Allows standard ANSI SQL constraints to be assigned at the column or table level. MySQL fully supports the following constraints: primary key, unique, and default (must be a constant). MySQL provides syntax support for check, foreign key, and references constraints, but they are not functional except on InnoDB tables. MySQL also has six specialty constraints:

FULLTEXT [{INDEX | KEY}]

Creates a fulltext-search index to facilitate rapid searches of large blocks of text. Note that only MyISAM tables support *FULLTEXT* indexes, and that they can only be made on *CHAR*, *VARCHAR*, and *TEXT* columns.

SPATIAL [{INDEX | KEY}]

Creates a spatial index or key on the column. Only MyISAM supports *SPATIAL* indexes.

AUTO_INCREMENT

Sets up an integer column so that it automatically increases its value by 1 (starting with a value of 1). MySQL only allows one *AUTO_INCREMENT* column per table. When all records are deleted (using *DELETE* or *TRUNCATE*), the values may start over. This option is allowed on MyISAM, MEMORY, ARCHIVE, and InnoDB tables.

[*UNIQUE*] *INDEX*

When an *INDEX* characteristic is assigned to a column, a name for the index can also be included. (A MySQL synonym for *INDEX* is *KEY*.) If a name is not assigned to the primary key, MySQL assigns a name of *column_name* plus a numeric suffix (_2, _3, ...) to make it unique. All table types except ISAM support indexes on NULL columns, or on *BLOB* or *TEXT* columns. Note that *UNIQUE* without *INDEX* is valid syntax on MySQL.

COLUMN_FORMAT {FIXED | DYNAMIC | DEFAULT}

Specifies a data storage format for individual columns in NDB tables. *FIXED* specifies fixed-width storage; *DYNAMIC* specifies variable-width storage; and *DEFAULT* specifies that either *FIXED* or *DYNAMIC* be used, according to the datatype for the column. This clause is not available in versions before 5.1.19-ndb.

STORAGE {DISK | MEMORY}

Specifies whether to store the column in an NDB table on the *DISK* or in *MEMORY* (the default). Not available in versions before 5.1.19-ndb.

ENGINE

Describes how the data should be physically stored. You can convert tables between types using the *ALTER TABLE* statement. Only InnoDB and BDB tables offer recoverability with the *COMMIT* or *ROLLBACK* statements. All other table types will result in the loss of data if the database crashes, but are much faster and take up less space. The default is MyISAM. Following is the list of MySQL table types:

ARCHIVE

Utilizes the ARCHIVE storage engine, which is good for storing large amounts of data without indexes in a small footprint. When creating an ARCHIVE table, the filename is the table's name with an *.FRM* extension. Data and metadata tables have the table name as the filename and extensions of *.ARX* and *.ARM*, respectively. A file with an *.ARN* extension may appear occasionally during optimizations.

CSV

Stores rows in comma-separated values (CSV) format. When creating a CSV table, the filename is the table name with an *.FRM* extension. Data is stored in a file with the table name as the filename and an extension of *.CSV*. The data is stored in plain text, so be careful with security on these tables.

EXAMPLE

EXAMPLE is a stub engine that does nothing. No data can be stored in an EXAMPLE table.

FEDERATED

Stores data in such a way that it is accessible from other remote MySQL databases without using replication or clustering technology. No data is stored in the local tables.

HEAP

Creates a memory-resident table that uses hashed indexes. Synonymous with MEMORY. Since they are memory-resident, these tables are not transaction-safe. Any data they contain will be lost if the server crashes. Think of HEAP tables as an alternative to temporary tables. If you use HEAP tables, always specify the *MAX_ROWS* option so that you do not use all available memory. HEAP tables also do not support *BLOB* columns, *TEXT* columns, *AUTO_INCREMENT* constraints, *ORDER BY* clauses, or the variable-length record format.

InnoDB

Creates a transaction-safe table with row-level locking. It also provides an independently managed tablespace, checkpoints, non-locking reads, and fast reads from large datafiles for heightened concurrency and performance.

Requires the `innodb_data_file_path` startup parameter. InnoDB supports all ANSI constraints, including `CHECK` and `FOREIGN KEY`. Well-known developer websites like Slashdot.org run on MySQL with InnoDB because of its excellent performance. InnoDB tables and indexes are stored together in their own tablespace (unlike other table formats, such as MyISAM, which store tables in separate files).

ISAM

Creates a table in the older ISAM style. ISAM has been deprecated and is obsolete and unavailable in MySQL v5.1. When upgrading, convert ISAM tables to MyISAM before performing the upgrade.

MEMORY

Creates a memory-resident table that uses hashed indexes. Synonymous with `HEAP`. Since they are memory-resident, the indexes are not transaction-safe; any data they contain will be lost if the server crashes. `MEMORY` tables can have up to 32 indexes per table and 16 columns per index, for a maximum index length of 500 bytes. Think of `MEMORY` tables as an alternative to temporary tables; like temporary tables, they are shared by all clients. If you use `MEMORY` tables, always specify the `MAX_ROWS` option so that you do not use all available memory. `MEMORY` tables do not support `BLOB` or `TEXT` columns, `ORDER BY` clauses, or the variable-length record format, and there are many additional rules concerning `MEMORY` tables. Be sure to read the vendor documentation before implementing `MEMORY` tables.

MERGE

Collects several identically structured MyISAM tables for use as one table, providing some of the performance benefits of a partitioned table. `SELECT`, `DELETE`, and `UPDATE` statements operate against the collection of tables as if they were one table. Think of a merge table as the collection, but not as the table(s) containing the data. Dropping a merge table only removes it from the collection; it does not erase the table or its data from the database. Specify a `MERGE` table by using the statement `UNION=(table1, table2, ...)`. The two keywords `MERGE` and `MRG_MyISAM` are synonyms.

MyISAM

Stores data in `.MYD` files and indexes in `.MYI` files. MyISAM is based on ISAM code, with several extensions. MyISAM is a binary storage structure that is more portable than ISAM. MyISAM supports `AUTO_INCREMENT` columns, compressed tables (with the `myisampack` utility), and large table sizes. Under MyISAM, `BLOB` and `TEXT` columns can be indexes, and up to 64 indexes are allowed per table, with up to 16 columns per index and a maximum key length of 1,000 bytes.

NDBCLUSTER

Creates clustered, fault-tolerant, memory-based tables called `NDBs`. This is MySQL's special high-availability table format. Refer to the vendor documentation for additional information on implementing NDBs.

UNION

Refer to `MERGE`.

TABLESPACE...STORAGE DISK

Assigns the table to a Cluster Disk Data tablespace when using NDB Cluster tables. The tablespace named in the clause must already have been created using `CREATE TABLESPACE`.

AUTO_INCREMENT = int

Sets the auto-increment value (*int*) for the table (MyISAM only).

AVG_ROW_LENGTH = int

Sets an approximate average row length for tables with variable-size records. MySQL uses *AVG_ROW_LENGTH* * *MAX_ROWS* to determine how big a table may be.

[DEFAULT] CHARACTER SET

Specifies the *CHARACTER SET* (or *CHARSET*) for the table, or for specific columns.

CHECKSUM = {0 | 1}

When set to 1, maintains a checksum for all rows in the table (MyISAM only). This makes processing slower but leaves your data less prone to corruption.

[DEFAULT] COLLATE

Specifies the collation set for the table, or for specific columns.

COMMENT = "string"

Allows a comment of up to 60 characters.

CONNECTION = 'connection_string'

The connection string required to connect to a FEDERATED table. Otherwise, this is a noise word. Older versions of MySQL used the *COMMENT* option for the connection string.

DATA DIRECTORY = "path_to_directory"

Overrides the default path and directory that MySQL should use for MyISAM table storage.

DELAY_KEY_WRITE = {0 | 1}

When set to 1, delays key table updates until the table is closed (MyISAM only).

INDEX DIRECTORY = "path_to_directory"

Overrides the default path and directory that MySQL should use for MyISAM index storage.

INSERT_METHOD = {NO | FIRST | LAST}

Required for MERGE tables. If no setting is specified for a MERGE table or the value is *NO*, *INSERTs* are disallowed. *FIRST* inserts all records to the first table in the collection, while *LAST* inserts them all into the last table of the collection

KEY_BLOCK_SIZE = int

Allows the storage engine to change the value used for the index key block size. 0 tells MySQL to use the default.

MAX_ROWS = int

Sets a maximum number of rows to store in the table. The default maximum is 4 GB of space.

MIN_ROWS = int

Sets a minimum number of rows to store in the table.

PACK_KEYS = {0 | 1}

When set to 1, compacts the indexes of the table, making reads faster but updates slower (MyISAM and ISAM only). By default, only strings are packed. When set to 1, both strings and numeric values are packed.

PASSWORD = "string"

Encrypts the .FRM file (but not the table itself) with a password, *"string"*.

ROW_FORMAT = { DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT }

Determines how future rows should be stored in a MySQL table. *DEFAULT* varies by storage engine. *DYNAMIC* allows the rows to be of variable sizes (i.e., using *VARCHAR*), while *FIXED* expects fixed-size columns (i.e., *CHAR*, *INT*, etc.). *REDUNDANT* is used only on InnoDB tables and maximizes the value of an index, even if some redundant data is stored. *COMPRESSED* tables are read-only and compresses the data by about 20% compared to the *REDUNDANT* format. *COMPRESSED* is also allowed only on InnoDB.

partition_definition

Specifies a partition or subpartition for a MySQL table. Refer to the section below for more details on partitioning and subpartitioning MySQL tables. Note that all of the definition options are usable for subpartitions, with the exception of the *VALUE* subclause.

[IGNORE | REPLACE] select_statement

Creates a table with columns based upon the elements listed in the *SELECT* statement. The new table will be populated with the results of the *SELECT* statement if the statement returns a result set.

ALTER [IGNORE]

The altered table will include all duplicate records unless the *IGNORE* keyword is used. If it is not used, the statement will fail if a duplicate row is encountered and if the table has a unique index or primary key.

{ADD | COLUMN} [FIRST | AFTER column_name]

Adds or moves a column, index, or key to the table. When adding or moving columns, the new column appears as the last column in the table unless it is placed *AFTER* another named column.

ALTER COLUMN

Allows the definition or resetting of a default value for a column. If you reset a default, MySQL will assign a new default value to the column.

CHANGE

Renames a column, or changes its datatype.

MODIFY

Changes a column's datatype or attributes such as *NOT NULL*. Existing data in the column is automatically converted to the new datatype.

DROP

Drops the column, key, index, or tablespace. A dropped column is also removed from any indexes in which it participated. When dropping a primary key, MySQL will drop the first unique key if no primary key is present.

{ENABLE | DISABLE} KEYS

Enables or disables all non-unique keys on a MyISAM table simultaneously. This can be useful for bulk loads where you want to temporarily disable constraints until after the load is finished. It also speeds performance by finishing all index block flushing at the end of the operation.

RENAME [TO] new_tbl_name

Renames a table.

ORDER BY column_name[, ...]

Orders the rows in the specified order.

CONVERT TO CHARACTER SET *charset_name* [COLLATE *collation_name*]

Converts the table to a character set (and optionally a collation) that you specify.

DISCARD | IMPORT TABLESPACE

Deletes the current .IDB file (using *DISCARD*), or makes a tablespace available after restoring from a backup (using *IMPORT*).

{ADD | DROP | COALESCE *int* | ANALYZE | CHECK | OPTIMIZE | REBUILD | REPAIR} PARTITION

Adds or drops a partition on a table. Other options perform preventative maintenance behaviors analogous to those available for MySQL tables (i.e., *CHECK TABLE* and *REPAIR TABLE*). Only *COALESCE PARTITION* has a unique behavior, in which MySQL reduces the number of *KEY* or *HASH* partitions to the number specified by *int*.

REORGANIZE PARTITION *prtn_name* INTO (*partition_definition*)

Alters the definition of an existing partition according to the new *partition_definition* specified.

REMOVE PARTITIONING

Removes a table's partitioning without otherwise affecting the table or its data.

For example:

```
CREATE TABLE test_example
  (column_a INT NOT NULL AUTO_INCREMENT,
   PRIMARY KEY(column_a),
   INDEX(column_b))
   TYPE=HEAP
   IGNORE
   SELECT column_b,column_c FROM samples;
```



In MySQL v5.1 and later, only the MyISAM, INNODB, and MEMORY storage engines support indexes on columns that contain NULL values. For all other indexed columns, the column must be declared *NOT NULL*. Spatial types are supported only on MyISAM and, when indexed, must be declared as *NOT NULL*. Note that MySQL v5.1 is limited to 4,096 columns per table, but it may not even be able to store that many due to space constraints.

This creates a heap table with three columns: **column_a**, **column_b**, and **column_c**. Later, we could change this table to the MyISAM type:

```
ALTER TABLE example TYPE=MyISAM;
```

Three operating-system files are created when MySQL creates a MyISAM table: a table definition file with the extension *.FRM*, a datafile with the extension *.MYD*, and an index file with the extension *.MYI*. The *.FRM* datafile is used for all other tables.

The following example creates two base MyISAM tables and then creates a *MERGE* table from them:

```
CREATE TABLE message1
  (message_id  INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20));
CREATE TABLE message2
  (message_id  INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20));
```

```
CREATE TABLE all_messages
  (message_id  INT AUTO_INCREMENT PRIMARY KEY,
   message_text CHAR(20))
  TYPE=MERGE UNION=(message1, message2) INSERT_METHOD=LAST;
```

Partitioned tables

MySQL allows partitioning of tables for greater control of I/O and space management. The syntax for the partitioning clause is:

```
PARTITION BY function
[ [SUB]PARTITION prtn_name
  [VALUES {LESS THAN {(expr) | MAXVALUE} | IN (value_list)}]
  [[STORAGE] ENGINE [=] engine_name]
  [COMMENT [=] 'comment_text']
  [DATA DIRECTORY [=] 'data_path']
  [INDEX DIRECTORY [=] 'index_path']
  [MAX_ROWS [=] max_rows]
  [MIN_ROWS [=] min_rows]
  [TABLESPACE [=] (tablespace_name)]
  [NODEGROUP [=] node_group_id]
  [(subprtn[, subprtn] ...)] [, ...]]
```

where (values not described below are redundant to the list of table options and are presented in the earlier listing):

function

Specifies the function used to create the partition. Allowable values include: *HASH(expr)*, where *expr* is a hash of one or more columns in an allowable SQL format (including function calls that return any single integer value); *LINEAR KEY(column_list)*, where MySQL's hashing function more evenly distributes data; *RANGE(expr)*, where *expr* is one or more columns in an allowable SQL format with the *VALUES* clause telling exactly which partition holds which values; and *LIST(expr)*, where *expr* is one or more columns in an allowable SQL format with the *VALUES* clause telling exactly which partition holds which values.

[SUB]PARTITION *prtn_name*

Names the partition or subpartition.

VALUES {LESS THAN {(*expr*) | MAXVALUE} | IN (*value_list*)}

Specifies which values are assigned to which partitions.

NODEGROUP [=] *node_group_id*

Makes the partition or subpartition act as part of a node group identified by *node_group_id*. Only applicable for NDB tables.

Note that partitions and subpartitions must all use the same storage engine.

The following example creates three tables, each with a different partitioning function:

```
CREATE TABLE employee (emp_id INT, emp_fname VARCHAR(30), emp_lname
  VARCHAR(50))
  PARTITION BY HASH(emp_id);
```

```
CREATE TABLE inventory (prod_id INT, prod_name VARCHAR(30), location_code
  CHAR(5))
  PARTITION BY KEY(location_code)
  PARTITIONS 4;
```

```
CREATE TABLE inventory (prod_id INT, prod_name VARCHAR(30), location_code
CHAR(5))
PARTITION BY LINEAR KEY(location_code)
PARTITIONS 5;
```

The following two examples show somewhat more elaborate examples of partitioning using *RANGE* partitioning and *LIST* partitioning:

```
CREATE TABLE employee (emp_id INT,
emp_fname VARCHAR(30),
emp_lname VARCHAR(50),
hire_date DATE)
PARTITION BY RANGE(hire_date)
(PARTITION prtn1 VALUES LESS THAN ('01-JAN-2004'),
PARTITION prtn2 VALUES LESS THAN ('01-JAN-2006'),
PARTITION prtn3 VALUES LESS THAN ('01-JAN-2008'),
PARTITION prtn4 VALUES LESS THAN MAXVALUE);

CREATE TABLE inventory (prod_id INT, prod_name VARCHAR(30), location_code
CHAR(5))
PARTITION BY LIST(prod_id)
(PARTITION prtn0 VALUES IN (10, 50, 90, 130, 170, 210),
PARTITION prtn1 VALUES IN (20, 60, 100, 140, 180, 220),
PARTITION prtn2 VALUES IN (30, 70, 110, 150, 190, 230),
PARTITION prtn3 VALUES IN (40, 80, 120, 160, 200, 240));
```

The following example renames both a table and a column:

```
ALTER TABLE employee RENAME AS emp;
ALTER TABLE employee CHANGE employee_ssn emp_ssn INTEGER;
```

Since MySQL allows the creation of indexes on a portion of a column (for example, on the first 10 characters of a column), you can also build short indexes on very large columns.

MySQL can redefine the datatype of an existing column, but to avoid losing any data, the values contained in the column must be compatible with the new datatype. For example, a date column could be redefined to a character datatype, but a character datatype could not be redefined to an integer. Here's an example:

```
ALTER TABLE mytable MODIFY mycolumn LONGTEXT
```

MySQL offers some additional flexibility in the *ALTER TABLE* statement by allowing users to issue multiple *ADD*, *ALTER*, *DROP*, and *CHANGE* clauses in a comma-delimited list in a single *ALTER TABLE* statement. However, be aware that the *CHANGE column_name* and *DROP INDEX* clauses are MySQL extensions not found in SQL2003. MySQL supports the clause *MODIFY column_name* to mimic the same feature found in Oracle.

Oracle

The Oracle syntax for *CREATE TABLE* creates a relational table either by declaring the structure or by referencing an existing table. You can modify a table after it is created using the *ALTER TABLE* statement. Oracle also allows the creation of a relational table that uses user-defined types for column definitions, an *object table* that is explicitly defined to hold a specific UDT (usually a *VARARRAY* or *NESTED TABLE* type), or an *XMLType* table.

The standard ANSI-style *CREATE TABLE* statement is supported, but Oracle has added many sophisticated extensions to the command. For example, Oracle allows significant control over the storage and performance parameters of a table. In both the *CREATE TABLE* and *ALTER TABLE* statements, you'll see a great deal of nesting and reusable clauses. To make this somewhat easier to read, we have broken Oracle's *CREATE TABLE* statement into three distinct variations (relational table, object table, and XML table) so that you can more easily follow the syntax.

The *CREATE TABLE* syntax for a standard relational table, which has no object or XML properties, is as follows:

```
CREATE [GLOBAL] [TEMPORARY] TABLE table_name
[ ( {column | virtual_column | attribute} [SORT] [DEFAULT expression]
[ {column_constraint |
  inline_ref_constraint}] |
{table_constraint_clause | table_ref_constraint} |
{GROUP log_group (column [NO LOG][, ...]) [ALWAYS] | DATA
  (constraints[, ...]) COLUMNS} ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[table_constraint_clause]
{ [physical_attributes_clause] [TABLESPACE tablespace_name]
  [storage_clause] [[NO]LOGGING] |
  [CLUSTER (column[, ...])] |
  {[ORGANIZATION
    {HEAP [physical_attributes_clause][TABLESPACE
      tablespace_name] [storage_clause]
      [COMPRESS | NOCOMPRESS] [[NO]LOGGING] |
      INDEX [physical_attributes_clause] [TABLESPACE tablespace_name]
        [storage_clause]
        [PCTTHRESHOLD int] [COMPRESS [int] | NOCOMPRESS]
        [MAPPING TABLE | NOMAPPING][...] [[NO]LOGGING]
        [[INCLUDING column] OVERFLOW
          [physical_attributes_clause] [TABLESPACE tablespace_name]
          [storage_clause] [[NO]LOGGING]]] |
    EXTERNAL ( [TYPE driver_type] ) DEFAULT DIRECTORY directory_name
      [ACCESS PARAMETERS {USING CLOB subquery | ( opaque_format )}]
      LOCATION ( [directory_name:]'location_spec'[, ...] )
      [REJECT LIMIT {int | UNLIMITED}] } }
  {[ENABLE | DISABLE] ROW MOVEMENT}
  {[NO]CACHE} {[NO]MONITORING} {[NO]ROWDEPENDENCIES} {[NO]FLASHBACK ARCHIVE}
  [PARALLEL int | NOPARALLEL] [NOSORT] [[NO]LOGGING]
  [COMPRESS [int] | NOCOMPRESS]
  {[ENABLE | DISABLE} {[NO]VALIDATE}
    {[UNIQUE (column[, ...]) | PRIMARY KEY | CONSTRAINT constraint_name} ]
    {[USING INDEX {index_name | CREATE_INDEX_statement}] [EXCEPTIONS INTO]
      [CASCADE] [{KEEP | DROP} INDEX]}] |
  [partition_clause]
  [AS subquery]
```

The relational table syntax contains a large number of optional clauses. However, the table definition must contain, at a minimum, either column names and datatypes specifications or the *AS subquery* clause.

The Oracle syntax for an object table follows:

```

CREATE [GLOBAL] [TEMPORARY] TABLE table_name
AS object_type [[NOT] SUBSTITUTABLE AT ALL LEVELS]
[ ( {column | attribute} [DEFAULT expression] [{column_constraint |
    inline_ref_constraint}] | 
    {table_constraint_clause | table_ref_constraint} | 
    {GROUP log_group (column [NO LOG][, ...]) [ALWAYS] | DATA
        (constraints[, ...]) COLUMNS} ) ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX index_name ([physical_attributes_clause] [storage_clause])]
[physical_attributes_clause] [TABLESPACE tablespace_name] [storage_clause]

```

Oracle allows you to create, and later alter, XMLType tables. XMLType tables may have regular columns or virtual columns. The Oracle syntax for an XMLType table follows:

```

CREATE [GLOBAL] [TEMPORARY] TABLE table_name
OF XMLTYPE
[ ( {column | attribute} [DEFAULT expression] [{column_constraint |
    inline_ref_constraint}] | 
    {table_constraint_clause | table_ref_constraint} | 
    {GROUP log_group (column [NO LOG][, ...]) [ALWAYS] | DATA
        (constraints[, ...]) COLUMNS} ) ]
[XMLTYPE {OBJECT RELATIONAL [xml_storage_clause] |
    [ {SECUREFILE | BASICFILE} ]
    [ {CLOB | BINARY XML} [lob_segname] [lob_params]}] [xml_schema_spec]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[OBJECT IDENTIFIER IS {SYSTEM GENERATED | PRIMARY KEY}]
[OIDINDEX index_name ([physical_attributes_clause] [storage_clause])]
[physical_attributes_clause] [TABLESPACE tablespace_name] [storage_clause]

```

The Oracle syntax for *ALTER TABLE* changes the table or column properties, storage characteristics, LOB or VARRAY properties, partitioning characteristics, and integrity constraints associated with a table and/or its columns. The statement can also do other things, like move an existing table into a different tablespace, free recuperated space, compact the table segment, and adjust the “high-water mark.”

The ANSI SQL standard uses the *ALTER* keyword to modify existing elements of a table, while Oracle uses *MODIFY* for the same purpose. Since they are essentially the same thing, please consider the behavior of otherwise identical clauses (for example, ANSI’s *ALTER TABLE...ALTER COLUMN* and Oracle’s *ALTER TABLE...MODIFY COLUMN*) to be functionally equivalent.

Oracle’s *ALTER TABLE* syntax is:

```

ALTER TABLE table_name
-- Alter table characteristics
[physical_attributes_clause] [storage_clause]
[ {READ ONLY | READ WRITE} ]
[[NO]LOGGING] [[NO]CACHE] [[NO]MONITORING] [[NO]COMPRESS]
[[NO]FLASHBACK ARCHIVE] [SHRINK SPACE [COMPACT] [CASCADE]]
[UPGRADE [[NOT] INCLUDING DATA] column_name datatype attributes]
[[NO]MINIMIZE RECORDS_PER_BLOCK]
[PARALLEL int | NOPARALLEL]
[ {ENABLE | DISABLE} ROW MOVEMENT]
[ {ADD | DROP} SUPPLEMENTAL LOG

```

```

{GROUP log_group [(column_name [NO LOG][, ...]) [ALWAYS]] | 
 DATA ( {ALL | PRIMARY KEY | UNIQUE | FOREIGN KEY}[, ...] ) COLUMNS}
[ALLOCATE EXTENT
 [([ [SIZE int [K | M | G | T]] [DATAFILE 'filename'] [INSTANCE int] )]
 [DEALLOCATE UNUSED [KEEP int [K | M | G | T]]]
 [ORGANIZATION INDEX ...
 [COALESCE] [MAPPING TABLE | NOMAPPING] [PCTTHRESHOLD int]
 [COMPRESS int | NOCOMPRESS]
 [ { ADD OVERFLOW [TABLESPACE tablespace_name] [[NO]LOGGING]
 [physical_attributes_clause] } |
 OVERFLOW [ [ALLOCATE EXTENT ( [SIZE int [K | M | G | T]] [DATAFILE
 'filename'] [INSTANCE int] ) |
 [DEALLOCATE UNUSED [KEEP int [K | M | G | T]]]] } ]]]
[RENAME TO new_table_name]
-- Alter column characteristics
[ADD (column_name datatype attributes[, ...])]
[DROP { {UNUSED COLUMNS | COLUMNS CONTINUE} [CHECKPOINT int] |
 {COLUMN column_name | (column_name[, ...])} [CHECKPOINT int]
 [{CASCADE CONSTRAINTS | INVALIDATE}] }]
[SET UNUSED {COLUMN column_name | (column_name[, ...])}
 [{CASCADE CONSTRAINTS | INVALIDATE}]]
[MODIFY { (column_name datatype attributes[, ...]) |
 COLUMN column_name [NOT] SUBSTITUTABLE AT ALL LEVELS [FORCE] }]
[RENAME COLUMN old_column_name TO new_column_name]
[MODIFY {NESTED TABLE | VARRAY} collection_item [RETURN AS {LOCATOR |
 VALUE}]]
-- Alter constraint characteristics
[ADD CONSTRAINT constraint_name table_constraint_clause]
[MODIFY CONSTRAINT constraint_name constraint_state_clause]
[RENAME CONSTRAINT old_constraint_name TO new_constraint_name]
[DROP { { PRIMARY KEY | UNIQUE (column[, ...]) } [CASCADE]
 [{KEEP | DROP} INDEX] |
 CONSTRAINT constraint_name [CASCADE] } ]
-- Alter table partition characteristics
[alter_partition_clauses]
-- Alter external table characteristics
DEFAULT DIRECTORY directory_name
[ACCESS PARAMETERS {USING CLOB subquery | ( opaque_format )}]
LOCATION ( [directory_name:]'location_spec'[, ...] )
[ADD (column_name ...)][DROP column_name ...][MODIFY (column_name ...)]
[PARALLEL int | NOPARALLEL]
[REJECT LIMIT {int | UNLIMITED}]
[PROJECT COLUMN {ALL | REFERENCED}]
-- Move table clauses
[MOVE [ONLINE] [physical_attributes_clause]
 [TABLESPACE tablespace_name] [[NO]LOGGING] [PCTTHRESHOLD int]
 [COMPRESS int | NOCOMPRESS] [MAPPING TABLE | NOMAPPING]
 [[INCLUDING column] OVERFLOW
 [physical_attributes_clause] [TABLESPACE tablespace_name]
 [[NO]LOGGING]]
 [LOB ...] [VARRAY ...] [PARALLEL int | NOPARALLEL]]
-- Enable/disable attributes and constraints
[{ {ENABLE | DISABLE} [[NO]VALIDATE] {UNIQUE (column[, ...])} |
 PRIMARY KEY | CONSTRAINT constraint_name}]

```

```
[USING INDEX {index_name | CREATE_INDEX_statement} |
 [TABLESPACE tablespace_name] [physical_attributes_clause]
 [storage_clause]
 [NOSORT] [[NO]LOGGING] [ONLINE] [COMPUTE STATISTICS]
 [COMPRESS | NOCOMPRESS] [REVERSE]
 [{LOCAL | GLOBAL} partition_clause]
 [EXCEPTIONS INTO table_name] [CASCADE] [{KEEP | DROP} INDEX]]} |
 [{ENABLE | DISABLE}] [{TABLE LOCK | ALL TRIGGERS}] }]
```

The parameters are as follows:

virtual_column

Allows the creation or alteration of a *virtual column* (i.e., a column whose value is derived from a calculation rather than directly from a physical storage location). For example, a virtual column **income** might be derived by summing the **salary**, **bonus**, and **commission** columns.

column_constraint

Specifies a column constraint using the syntax described later.

GROUP log_group (column [NO LOG][, ...]) [ALWAYS | DATA (constraints[, ...])]
COLUMNS

Specifies a log group rather than a single logfile for the table.

ON COMMIT {DELETE | PRESERVE} ROWS

Declares whether a declared temporary table should keep the data in the table active for the entire session (*PRESERVE*) or only for the duration of the transaction in which the temporary table is created (*DELETE*).

table_constraint_clause

Specifies a table constraint using the syntax described later.

physical_attributes_clause

Specifies the physical attributes of the table using the syntax described later.

TABLESPACE *tablespace_name*

Specifies the name of the tablespace where the table you are creating will be stored. If omitted, the default tablespace for the schema owner will be used. See below for specifics. See the Oracle Concepts manual to learn about tablespaces and their use.

storage_clause

Specifies physical storage characteristics of the table using the syntax described later.

[NO]LOGGING

Specifies whether redo log records will be written during object creation (*LOGGING*) or not (*NOLOGGING*). *LOGGING* is the default. *NOLOGGING* can speed the creation of database objects. However, in case of database failure under the *NOLOGGING* option, the operation cannot be recovered by applying logfiles, and the object must be recreated. The *LOGGING* clause replaces the older *RECOVERABLE* clause, which is deprecated.

CLUSTER(*column*[, ...])

Declares that the table is part of a clustered index. The *column* list should correspond, one to one, with the columns in a previously declared clustered index. Because it uses the clustered index's space allocation, the *CLUSTER* clause is compatible with the *physical_attributes_clause*, *storage_clause*, or *TABLESPACE* clause. Tables containing *LOBs* are incompatible with the *CLUSTER* clause.

ORGANIZATION HEAP

Declares how the data of the table should be recorded to disk. *HEAP*, the default for an Oracle table, declares that no order should be associated with the storage of rows of data (i.e., the physical order in which records are written to disk) for this table. The *ORGANIZATION HEAP* clause allows several optional clauses, described in detail elsewhere in this list, that control storage, logging, and compression for the table.

ORGANIZATION INDEX

Declares how the data of the table should be recorded to disk. *INDEX* declares that the records of the table should be physically written to disk in the sort order defined by the primary key of the table. Oracle calls this an *index-organized table*. A primary key is required. Note that the *physical_attributes_clause*, the *TABLESPACE* clause, and the *storage_clause* (all described in greater detail elsewhere in this section) and the *[NO]LOGGING* keyword may all be associated with the new *INDEX* segment as you create it. In addition, the following subclauses may also be associated with an *ORGANIZATION INDEX* clause:

PCTTHRESHOLD *int*

Declares the percentage (*int*) of space in each index block to be preserved for data. On a record-by-record basis, data that cannot fit in this space will be placed in the overflow segment.

INCLUDING *column*

Declares the point at which a record will split between index and overflow portions. All columns that follow the specified *column* will be stored in the overflow segment. The *column* cannot be a primary key column.

MAPPING TABLE | NOMAPPING

Tells the database to create a mapping of local to physical *ROWIDs*. This mapping is required to create a bitmap index on an IOT. Mappings are also partitioned identically if the table is partitioned. *NOMAPPING* tells the database not to create the *ROWID* map.

[INCLUDING *column*] OVERFLOW

Declares that a record that exceeds the *PCTTHRESHOLD* value be placed in a segment described in this clause. The *physical_attributes_clause*, the *TABLESPACE* clause, the *storage_clause* (all described elsewhere in the list in greater detail) and the *[NO]LOGGING* keyword may all be associated with a specific *OVERFLOW* segment when you create it. The optional *INCLUDING column* clause defines a column at which to divide an IOT row into index and overflow portions. Primary key columns are always stored in the index. However, all non-primary key columns that follow *column* are stored in the overflow data segment.

ORGANIZATION EXTERNAL

Declares how the data of the table should be recorded to disk. *EXTERNAL* declares that the table stores its data outside of the database and is usually read-only (its metadata is stored in the database, but its data is stored outside of the database). There are some restrictions on external tables: they cannot be temporary; they cannot have constraints; they can only have column, datatype, and attribute column-descriptors; and *LOB* and *LONG* datatypes are disallowed. No other *ORGANIZATION* clauses are allowed with *EXTERNAL*. The following subclauses may be used with the *ORGANIZATION EXTERNAL* clause:

TYPE *driver_type*

Defines the access driver API for the external table. The default is ORACLE_LOADER.

DEFAULT DIRECTORY *directory_name*

Defines the default directory on the filesystem where the external table resides.

ACCESS PARAMETERS {USING CLOB *subquery* | (*opaque_format*)}

Assigns and passes specific parameters to the access driver. Oracle does not interpret this information. *USING CLOB subquery* tells Oracle to derive the parameters and their values from a subquery that returns a single row with a single column of the datatype CLOB. The subquery cannot contain an ORDER BY, UNION, INTERSECT, or MINUS/EXCEPT clause. The *opaque_format* clause allows you to list parameters and their values, as described in the ORACLE LOADER section of the “Oracle9i Database Utilities” guide.

LOCATION (*directory_name*:'*location_spec*'[, ...])

Defines one or more external data sources, usually as files. Oracle does not interpret this information.

REJECT LIMIT {*int* | UNLIMITED}

Defines the number of conversion errors (*int*) that are allowed during the query to the external data source before Oracle aborts the query and returns an error. UNLIMITED tells Oracle to continue with the query no matter how many errors are encountered. The default is 0.

{ENABLE | DISABLE} ROW MOVEMENT

Specifies that a row may be moved to a different partition or subpartition if required due to an update of the key (ENABLE), or not (DISABLE). The DISABLE keyword also specifies that Oracle return an error if an update to a key would require a move.

[NO]CACHE

Buffers a table for rapid reads (CACHE), or turns off this behavior (NOCACHE). Index-organized tables offer CACHE behavior.

[NO]MONITORING

Specifies whether modification statistics can be collected for this table (MONITORING) or not (NOMONITORING). NOMONITORING is the default.

[NO]ROWDEPENDENCIES

Specifies whether a table will use *row-level dependency tracking*, a feature that applies a system change number (SCN) greater than or equal to the time of the last transaction affecting the row. The SCN adds 6 extra bytes of space to each record. Row-level dependency tracking is most useful in replicated environments with parallel data propagation. NOROWDEPENDENCIES is the default.

[NO]FLASHBACK ARCHIVE

Enables or disables historical tracking for the table, if a flashback archive for the table already exists. NO FLASHBACK ARCHIVE is the default.

PARALLEL [*int*] | NOPARALLEL

The PARALLEL clause allows for the parallel creation of the table by distinct CPUs to speed the operation. It also enables parallelism for queries and other data-manipulation operations against the table after its creation. An optional integer value may be supplied to define the exact number of parallel threads used

to create the table in parallel, as well as the number of parallel threads allowed to service the table in the future. (Oracle calculates the best number of threads to use in a given parallel operation, so the *int* argument is optional.) *NOPARALLEL*, the default, creates the table serially and disallows future parallel queries and data-manipulation operations.

COMPRESS [int] | NOCOMPRESS

Specifies whether the table should be compressed or not. On index-organized tables, only the key is compressed; on heap-organized tables, the entire table is compressed. This can greatly reduce the amount of space consumed by the table. *NOCOMPRESS* is the default. In index-organized tables, you can specify the number of prefix columns (*int*) to compress. The default value for *int* is the number of keys in the primary key minus one. You need not specify an *int* value for other clauses, such as *ORGANIZATION*. When you omit the *int* value, Oracle will apply compression to the entire table.

{ENABLE | DISABLE} {[NO]VALIDATE} {UNIQUE (column[, ...]) | PRIMARY KEY | CONSTRAINT constraint_name}

Declares whether the named key or constraint applies to all of the data in the new table or not. *ENABLE* specifies that the key or constraint applies to all new data in the table while *DISABLE* specifies that the key or constraint is disabled for the new table, with the following options:

[NO]VALIDATE

VALIDATE verifies that all existing data in the table complies with the key or constraint. When *NOVALIDATE* is specified with *ENABLE*, Oracle does not verify that existing data in the table complies with the key or constraint, but ensures that new data added to the table does comply with the constraint.

UNIQUE (column[, ...]) | PRIMARY KEY | CONSTRAINT constraint_name

Declares the unique constraint, primary key, or constraint that is enabled or disabled.

USING INDEX index_name | CREATE_INDEX_statement

Declares the name (*index_name*) of a pre-existing index (and its characteristics) used to enforce the key or constraint, or creates a new index (*CREATE_INDEX_statement*). If neither clause is declared, Oracle creates a new index.

EXCEPTIONS INTO table_name

Specifies the name of a table into which Oracle places information about rows violating the constraint. Run the *utlexpt1.sql* script before using this keyword to explicitly create this table.

CASCADE

Cascades the disablement/enablement to any integrity constraints that depend on the constraint named in the clause. Usable only with the *DISABLE* clause.

{KEEP | DROP} INDEX

Lets you keep (*KEEP*) or drop (*DROP*) an index used to enforce a unique or primary key. You can drop the key only when disabling it.

partition_clause

Declares partitioning and subpartitioning of a table. Partitioning syntax can be quite complex; refer to the material later in this section under “Oracle partitioned and subpartitioned tables” for the full syntax and examples.

AS subquery

Declares a *subquery* that inserts rows into the table upon creation. The column names and datatypes used in the subquery can act as substitutes for column name and attribute declarations for the table.

AS object_type

Declares that the table is based on a pre-existing object type.

[NOT] SUBSTITUTABLE AT ALL LEVELS

Declares whether row objects corresponding to subtypes can be inserted into the type table or not. When this clause is omitted, the default is *SUBSTITUTABLE AT ALL LEVELS*.

inline_ref_constraint and table_ref_constraint

Declares a reference constraint used by an object-type table or XMLType table. These clauses are described in greater detail later in this section.

OBJECT IDENTIFIER IS (SYSTEM GENERATED | PRIMARY KEY)

Declares whether the object ID (OID) of the object-type table is *SYSTEM GENERATED* or based on the *PRIMARY KEY*. When omitted, the default is *SYSTEM GENERATED*.

OIDINDEX [index_name]

Declares an index, and possibly a name for the index, if the OID is system-generated. You may optionally apply a *physical_attributes_clause* and a *storage_clause* to the *OIDINDEX*. If the OID is based on the primary key, this clause is unnecessary.

OF XMLTYPE

Declares that the table is based on Oracle's *XMLTYPE* datatype.

XMLTYPE {OBJECT RELATIONAL [xml_storage_clause] | [{SECUREFILE | BASICFILE}] [{CLOB | BINARY XML} [lob_segname] [lob_params]]}

Declares how the underlying data of the *XMLTYPE* is stored: either in *LOB*, object-relational, or binary XML format. *OBJECT RELATIONAL* stores the data in object-relational columns and allows indexing for better performance. This subclause requires an *xml_schema_spec* and a schema that has been preregistered using the *DBMS_XMLSCHEMA* package. *CLOB* specifies that the *XMLTYPE* data will be stored in a *LOB* column for faster retrieval. You may optionally specify the *LOB* segment name and/or the *LOB* storage parameters, but you cannot specify *LOB* details and XMLSchema specifications in the same statement. *BINARY XML* stores the data in a compact binary XML format, with any *LOB* parameters applied to the underlying *BLOB* column.

xml_schema_spec

Allows you to specify the URL of one or more registered XML schemas, and an XML element name. The element name is required, but the URL is optional. Multiple schemas are allowed only when using the *BINARY XML* storage format. You may further specify *ALLOW ANYSCHEMA* to store any schema-based document in the *XMLType* column, *ALLOW NONSCHEMA* to store non-schema-based documents, or *DISALLOW NONSCHEMA* to prevent storage of non-schema-based documents.

READ ONLY | READ WRITE

Places the table in read-only mode, which disallows all DML operations including *SELECT...FOR UPDATE*. Regular *SELECT* statements are allowed, as are operations on indexes associated with a read-only table. *READ WRITE* re-enables normal DML operations.

ADD ...

Adds a new column, virtual column, constraint, overflow segment, or supplemental log group to an existing table. You may also alter an XMLType table by adding (or removing) one or more XMLSchemas.

MODIFY ...

Changes an existing column, constraint, or supplemental log group on an existing table.

DROP ...

Drops an existing column, constraint, or supplemental log group from an existing table. You can explicitly drop columns marked as unused from a table with *DROP UNUSED COLUMNS*; however, Oracle will also drop all unused columns when *any* other column is dropped. The *INVALIDATE* keyword causes any object that depends on the dropped object, such as a view or stored procedure, to become invalid and unusable until the dependent object is recompiled or reused. The *COLUMNS CONTINUE* clause is used only when a *DROP COLUMN* statement failed with an error and you wish to continue where it left off.

RENAME ...

Renames an existing table, column, or constraint on an existing table.

SET UNUSED ...

Declares a column or columns to be unused. Those columns are no longer accessible from *SELECT* statements, though they still count toward the maximum number of columns allowed per table (1,000). *SET UNUSED* is the fastest way to render a column unusable within a table, but it is not the best way. Only use *SET UNUSED* as a shortcut until you can actually use *ALTER TABLE...DROP* to drop the column.

COALESCE

Merges the contents of index blocks used to maintain the index-organized table so that the blocks can be reused. *COALESCE* is similar to *SHRINK*, though *COALESCE* compacts the segments less densely than *SHRINK* and does not release unused space.

ALLOCATE EXTENT

Explicitly allocates a new extent for the table using the *SIZE*, *DATAFILE*, and *INSTANCE* parameters. You may mix and match any of these parameters. The size of the extent may be specified in bytes (no suffix), kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T).

DEALLOCATE UNUSED [KEEP int [K | M | G | T]]

Deallocates unused space at the end of the table, *LOB* segment, partition, or subpartition. The deallocated space is then usable by other objects in the database. The *KEEP* keyword indicates how much space you want to have left over after deallocation is complete.

SHRINK SPACE [COMPACT] [CASCADE]

Shrinks the table, index-organized table, index, partition, subpartition, materialized view, or materialized log view, though only segments in tablespaces with automatic segment management may be shrunk. Shrinking a segment moves rows in the table, so make sure *ENABLE ROW MOVEMENT* is also used in the *ALTER TABLE...SHRINK* statement. Oracle compacts the segment, releases the emptied space, and adjusts the high-water mark unless the optional keywords *COMPACT* and/or *CASCADE* are applied. The *COMPACT* keyword only defragments the segment space and compacts the table row for subsequent release; it

does not readjust the high-water mark or empty the space immediately. The *CASCADE* keyword performs the same shrinking operation (with some restrictions and exceptions) on all dependent objects of the table, including secondary indexes on index-organized tables. Used only with *ALTER TABLE*.

UPGRADE [NOT] INCLUDING DATA

Converts the metadata of object tables and relational tables with object columns to the latest version for each referenced type. The *INCLUDING DATA* clause will either convert the data to the latest type format (*INCLUDING DATA*) or leave it unchanged (*NOT INCLUDING DATA*).

MOVE ...

Moves the tablespace, index-organized table, partition, or subpartition to a new location on the filesystem.

[NO]MINIMIZE RECORDS_PER_BLOCK

Tells Oracle to restrict or leave open the number of records allowed per block. The *MINIMIZE* keyword tells Oracle to calculate the maximum number of records per block and set the limit at that number. (Best to do this when a representative amount of data is already in the table.) This clause is incompatible with nested tables and index-organized tables. *NOMINIMIZE* is the default.

PROJECT COLUMN {REFERENCE | ALL}

Determines how the driver for the external data source validates the rows of the external table in subsequent queries. *REFERENCE* processes only those columns in the select item list. *ALL* processes the values in all columns, even those not in the select item list, and validates rows with full and valid column entries. Under *ALL*, rows are rejected when errors occur, even on columns that are not selected. *ALL* returns consistent results, while *REFERENCE* returns varying numbers of rows depending on the columns referenced.

{ENABLE | DISABLE} {TABLE LOCK | ALL TRIGGERS}

Enables or disables table-level locks and all triggers on the table, respectively. *ENABLE TABLE LOCK* is required if you wish to change the structure against an existing table, but it is not required when changing or reading the data of the table.

A global temporary table is available to all user sessions, but the data stored within a global temporary table is visible only to the session that inserted it. The *ON COMMIT* clause, which is allowed only when creating temporary tables, tells Oracle either to truncate the table after each commit against the table (*DELETE ROWS*) or to truncate the table when the session terminates (*PRESERVE ROWS*). For example:

```
CREATE GLOBAL TEMPORARY TABLE shipping_schedule
  (ship_date DATE,
   receipt_date DATE,
   received_by VARCHAR2(30),
   amt NUMBER)
  ON COMMIT PRESERVE ROWS;
```

This *CREATE TABLE* statement creates a global temporary table, *shipping_schedule*, which retains inserted rows across multiple transactions.

The Oracle physical_attributes_clause

The *physical_attributes_clause* (shown in the following code block) defines storage characteristics for an entire local table, or, if the table is partitioned, for a specific partition (discussed later). To declare the physical attributes of a new table or change the attributes on an existing table, simply declare the new values:

```
-- physical_attributes_clause
[ {PCTFREE int | PCTUSED int | INITTRANS int |
  storage_clause} ]
```

where:

PCTFREE int

Defines the percentage of free space reserved for each data block in the table. For example, a value of 10 reserves 10% of the data space for new rows to be inserted.

PCTUSED int

Defines the minimum percentage of space allowed in a block before it can receive new rows. For example, a value of 90 means new rows are inserted in the data block when the space used falls below 90%. The sum of *PCTFREE* and *PCTUSED* cannot exceed 100.

INITTRANS int

Rarely tinkered with; defines the allocation of from 1 to 255 initial transactions to a data block.



In versions prior to 11g the *MAXTRANS* parameter was used to define the maximum allowed number of concurrent transactions on a data block, but this parameter has now been deprecated. Oracle 11g automatically sets *MAXTRANS* to 255, silently overriding any other value that you specify for this parameter (although existing objects retain their established *MAXTRANS* settings).

The Oracle storage_clause and LOBs

The *storage_clause* controls a number of attributes governing the physical storage of data:

```
-- storage_clause
STORAGE ( [ {INITIAL int [K | M | G | T]
  | NEXT int [K | M]
  | MINEXTENTS int
  | MAXEXTENTS {int | UNLIMITED}
  | PCTINCREASE int
  | FREELISTS int
  | FREELIST GROUPS int
  | BUFFER_POOL {KEEP | RECYCLE | DEFAULT}} ] [...] )
```

When delineating the storage clause attributes, enclose them in parentheses and separate them with spaces—for example, *(INITIAL 32M NEXT8M)*. The attributes are as follows:

INITIAL int [K | M | G | T]

Sets the initial extent size of the table in bytes, kilobytes (K), megabytes (M), gigabytes (G), or terabytes (T).

NEXT int [K | M]

Tells how much additional space to allocate after *INITIAL* is filled.

MINEXTENTS int

Tells Oracle to create a minimum number of extents. By default, only one is created, but more can be created when the object is initialized.

MAXEXTENTS int | UNLIMITED

Tells Oracle the maximum number of extents allowed. This value may be set to **UNLIMITED**. (Note that **UNLIMITED** should be used with caution, since a table could grow until it consumes all free space on a disk.)

PCTINCREASE int

Controls the growth rate of the object after the first growth. The initial extent gets allocated as specified, the second extent is the size specified by **NEXT**, the third extent is **NEXT + (NEXT * PCTINCREASE)**, and so on. When **PCTINCREASE** is set to 0, **NEXT** is always used. Otherwise, each added extent of storage space is **PCTINCREASE** larger than the previous extent.

FREELISTS int

Establishes the number of freelists for each group, defaulting to 1.

FREELIST GROUPS int

Sets the number of groups of freelists, defaulting to 1.

BUFFER_POOL {KEEP | RECYCLE | DEFAULT}

Specifies a default buffer pool or cache for any non-cluster table where all object blocks are stored. Index-organized tables may have a separate buffer pool for the index and overflow segments. Partitioned tables inherit the buffer pool from the table definition unless they are specifically assigned a separate buffer pool.

KEEP

Puts object blocks into the **KEEP** buffer pool; that is, directly into memory. This enhances performance by reducing I/O operations on the table. **KEEP** takes precedence over the **NOCACHE** clause.

RECYCLE

Puts object blocks into the **RECYCLE** buffer pool.

DEFAULT

Puts object blocks into the **DEFAULT** buffer pool. When this clause is omitted, **DEFAULT** is the default buffer pool behavior.

For example, the table **books_sales** is defined on the **sales** tablespace as consuming an initial 8 MB of space, to grow by no less than 8 MB when the first extent is full. The table has no less than 1 and no more than 8 extents, limiting its maximum size to 64 MB:

```
CREATE TABLE book_sales
  (qty NUMBER,
   period_end_date DATE,
   period_nbr NUMBER)
TABLESPACE sales
STORAGE (INITIAL 8M NEXT 8M MINEXTENTS 1 MAXEXTENTS 8);
```

An example of a **LOB** table called **large_objects** with special handling for text and image storage might look like this:

```
CREATE TABLE large_objects
  (pretty_picture BLOB,
   interesting_text CLOB)
STORAGE (INITIAL 256M NEXT 256M)
LOB (pretty_picture, interesting_text)
STORE AS (TABLESPACE large_object_segment
          STORAGE (INITIAL 512M NEXT 512M)
          NOCACHE LOGGING);
```

The exact syntax used to define a *LOB*, *CLOB*, or *NCLOB* column is defined by the *lob_parameter_clause*. *LOBs* can appear at many different levels within an Oracle table. For instance, separate *LOB* definitions could exist in a single table in a partition definition, in a subpartition definition, and at the top table-level definition. The syntax of *lob_parameter_clause* follows:

```
{TABLESPACE tablespace_name} [{SECUREFILE | BASICFILE}]
[{{ENABLE | DISABLE} STORAGE IN ROW]
[storage_clause] [CHUNK int] [PCTVERSION int]
[RETENTION [{MAX | MIN int | AUTO | NONE}]]
[{{DEDUPLICATE | KEEP_DUPLICATES}}
[{{NOCOMPRESS | COMPRESS [{HIGH | MEDIUM}]}}]
[{{FREEPOOLS int}]
[{{CACHE | {NOCACHE | CACHE READS} [{LOGGING | NOLOGGING}]}]
```

In the *lob_parameter_clause*, each parameter is identical to those of the wider *CREATE TABLE LOB*-object level. However, the following parameters are unique to *LOBs*:

SECUREFILE | BASICFILE

Specifies use of either the high-performance *LOB* storage (*SECUREFILE*) or the traditional *LOB* storage (*BASICFILE*, the default). When using *SECUREFILE*, you get access to other new features such as *LOB* compression, encryption, and deduplication.

{ENABLE | DISABLE} STORAGE IN ROW

Defines whether the *LOB* value is stored inline with the other columns of the row and the *LOB* locator (*ENABLE*), when it is smaller than approximately 4,000 bytes or less, or outside of the row (*DISABLE*). This setting cannot be changed once it is set.

*CHUNK *int**

Allocates *int* number of bytes for *LOB* manipulation. *int* should be a multiple of the database block size; otherwise, Oracle will round up. *int* should also be less than or equal to the value of *NEXT*, from the *storage_clause*, or an error will be raised. When omitted, the default chunk size is one block. This setting cannot be changed once it is set.

*PCTVERSION *int**

Defines the maximum percentage (*int*) of the overall *LOB* storage dedicated to maintaining old versions of the *LOB*. When omitted, the default is 10%.

*RETENTION [{MAX | MIN *int* | AUTO | NONE}]*

Used in place of *PCTVERSION* on databases in automatic undo mode. *RETENTION* tells Oracle to retain old versions of the *LOB*. When using *SECUREFILE*, you may specify additional options. *MAX* tells Oracle to allow the undo file to grow until the *LOB* segment has reached its maximum size, as defined by the *MAXSIZE* value of the *storage_clause*. *MIN* limits undo to *int* seconds if the database is in flashback mode. *AUTO*, the default, maintains enough undo for consistent reads. *NONE* specifies that the undo is not required.

DEDUPLICATE | KEEP_DUPLICATES

Specifies whether to keep duplicate *LOB* values within an entire *LOB* segment (*KEEP_DUPLICATES*) or to eliminate duplicate copies (*DEDUPLICATE*, the default). Only usable with *SECUREFILE LOBs*.

NOCOMPRESS | COMPRESS [{HIGH | MEDIUM}]

NOCOMPRESS, the default, disables server-side compression of LOBs in the SECUREFILE format. Alternately, you may tell Oracle to compress LOBs using either a *MEDIUM* (the default when a value is omitted) or *HIGH* degree of compression (*HIGH* compression incurs more overhead).

The following example shows our **large_objects** LOB table with added parameters to control inline storage and retention of old LOBs:

```
CREATE TABLE large_objects
  (pretty_picture BLOB,
   interesting_text CLOB)
  STORAGE (INITIAL 256M NEXT 256M)
  LOB (pretty_picture, interesting_text)
  STORE AS (TABLESPACE large_object_segment
            STORAGE (INITIAL 512M NEXT 512M)
            NOCACHE LOGGING
            ENABLE STORAGE IN ROW
            RETENTION);
```

The earlier example added parameter values for *STORAGE IN ROW* and *RETENTION*, but since we did not set one for *CHUNK*, that value is set to the Oracle default for the LOB.

Oracle nested tables

Oracle allows the declaration of a *NESTED TABLE*, in which a table is virtually stored within a column of another table. The *STORE AS* clause enables a proxy name for the table within a table, but the nested table must be created initially as a user-defined datatype. This capability is valuable for sparse arrays of values, but we don't recommend it for day-to-day tasks. This example creates a table called **proposal_types** along with a nested table called **props_nt**, which is stored as **props_nt_table**:

```
CREATE TYPE prop_nested_tbl AS TABLE OF props_nt;
CREATE TABLE proposal_types
  (proposal_category VARCHAR2(50),
   proposals          PROPS_NT)
  NESTED TABLE props_nt STORE AS props_nt_table;
```

Oracle compressed tables

Starting at Oracle 9i Release 2, Oracle allows compression of both keys and entire tables. (Oracle 9i Release 1 allowed only key compression.) Although compression adds a tiny bit of overhead, it significantly reduces the amount of disk space consumed by a table. This is especially useful for databases pushing the envelope in terms of size. Key compression is handled in the *ORGANIZE INDEX* clause, while table compression is handled in the *ORGANIZE HEAP* clause.

Oracle partitioned and subpartitioned tables

Oracle allows tables to be partitioned and subpartitioned. You can also break out LOBs onto their own partition(s). A partitioned table may be broken into distinct parts, possibly placed on separate disk subsystems to improve I/O performance (based on four strategies: range, hash, list, or a composite of the first three), or on a system partition. Partitioning syntax is quite elaborate:

```
{ PARTITION BY RANGE (column[, ...])
  [INTERVAL (expression) [STORE IN (tablespace[, ...])]]
  (PARTITION [partition_name])}
```

```

VALUES LESS THAN ({MAXVALUE | value}{[, ...]})  

[table_partition_description]) |  

PARTITION BY HASH (column[, ...])  

{(PARTITION [partition_name] [partitioning_storage_clause][, ...]) |  

 PARTITIONS hash_partition_qty [STORE IN (tablespace[, ...])] |  

 [OVERFLOW STORE IN (tablespace[, ...])]} |  

PARTITION BY LIST (column[, ...]) (PARTITION [partition_name]  

VALUES ({MAXVALUE | value}{[, ...]})  

[table_partition_description]) |  

PARTITION BY RANGE (column[, ...])  

{subpartition_by_list | subpartition_by_hash}  

(PARTITION [partition_name] VALUES LESS THAN  

 ({MAXVALUE | value}{[, ...]})  

[table_partition_description]) |  

PARTITION BY SYSTEM [int] |  

PARTITION BY REFERENCE (constraint)  

[ (PARTITION [partition_name] [table_partition_description][, ...]) ]  

}

```

The following example code shows the **orders** table partitioned by range:

```

CREATE TABLE orders
(order_number NUMBER,
order_date DATE,
cust_nbr NUMBER,
price NUMBER,
qty NUMBER,
cust_shp_id NUMBER)
PARTITION BY RANGE(order_date)
  (PARTITION pre_yr_2000 VALUES LESS THAN
   TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
  PARTITION pre_yr_2004 VALUES LESS THAN
   TO_DATE('01-JAN-2004', 'DD-MON-YYYY'
  PARTITION post_yr_2004 VALUES LESS THAN
   MAXVALUE));

```

This example creates three partitions on the **orders** table—one for the orders taken before the year 2000 (**pre_yr_2000**), one for the orders taken before the year 2004 (**pre_yr_2004**), and another for the orders taken after the year 2004 (**post_yr_2004**)—all based on the range of dates that appear in the **order_date** column.

The *INTERVAL* clause further facilitates range partitioning on numeric or datetime values by automatically creating new partitions when the current range boundaries are exceeded. The interval *expression* defines a valid number for the range boundary. Use the *STORE IN* subclause to tell Oracle which tablespace(s) will store the interval partition data. You cannot use interval partitioning on index-organized tables, with domain indexes, or at a subpartition level.

The next example creates the **orders** table based on a hash value in the **cust_shp_id** column:

```

CREATE TABLE orders
(order_number NUMBER,
order_date DATE,
cust_nbr NUMBER,
price NUMBER,
qty NUMBER,
cust_shp_id NUMBER)

```

```

PARTITION BY HASH (cust_shp_id)
  (PARTITION shp_id1 TABLESPACE tblspc01,
  PARTITION shp_id2 TABLESPACE tblspc02,
  PARTITION shp_id3 TABLESPACE tblspc03)
  ENABLE ROW MOVEMENT;

```

The big difference in how the records are divided among partitions between the hash partition example and the range partition example is that the range partition code explicitly defines where each record goes, while the hash partition example allows Oracle to decide (by applying a hash algorithm) which partition to place the record in. (Note that we also enabled row movement for the table).

In addition to breaking tables apart into partitions (for easier backup, recovery, or performance reasons), you may further break them apart into subpartitions. The *subpartition_by_list* clause syntax follows:

```

SUBPARTITION BY LIST (column)
  [SUBPARTITION TEMPLATE
  { (SUBPARTITION subpartition_name
      [VALUES {DEFAULT | {val | NULL}[, ...]}]
      [partitioning_storage_clause]) |
    hash_subpartition_qty } ]

```

As an example, we'll recreate the **orders** table once again, this time using a range-hash composite partition. In a range-hash composite partition, the partitions are broken apart by range values, while the subpartitions are broken apart by hashed values. List partitions and subpartitions are broken apart by a short list of specific values. Because you must list out all the values by which the table is partitioned, the partition value is best taken from a small list of values. In this example, we've added a column (**shp_region_id**) that allows four possible regions:

```

CREATE TABLE orders
  (order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20))
PARTITION BY RANGE(order_date)
  SUBPARTITION BY LIST(shp_region)
    SUBPARTITION TEMPLATE(
      (SUBPARTITION shp_region_north
        VALUES ('north','northeast','northwest'),
      SUBPARTITION shp_region_south
        VALUES ('south','southeast','southwest'),
      SUBPARTITION shp_region_central
        VALUES ('midwest'),
      SUBPARTITION shp_region_other
        VALUES ('alaska','hawaii','canada'))
    (PARTITION pre_yr_2000 VALUES LESS THAN
      TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
    PARTITION pre_yr_2004 VALUES LESS THAN
      TO_DATE('01-JAN-2004', 'DD-MON-YYYY')
    PARTITION post_yr_2004 VALUES LESS THAN
      MAXVALUE))
  ENABLE ROW MOVEMENT;

```

This code example sends the records of the table to one of three partitions based on the **order_date**, and further partitions the records into one of four subpartitions based on the region where the order is being shipped and on the value of the **shp_region** column. By using the *SUBPARTITION TEMPLATE* clause, you apply the same set of subpartitions to each partition. You can manually override this behavior by specifying subpartitions for each partition.

You may also subpartition using a hashing algorithm. The *subpartition_by_hash* clause syntax follows:

```
SUBPARTITION BY HASH (column[, ...])
  {SUBPARTITIONS qty [STORE IN (tablespace_name[, ...])] | 
  SUBPARTITION TEMPLATE
    { (SUBPARTITION subpartition_name [VALUES {DEFAULT | {val | NULL}
      [, ...]}] [partitioning_storage_clause] | 
      hash_subpartition_qty )}
```

The *table_partition_description* clause referenced in the partitioning syntax is, in itself, very flexible and supports precise handling of *LOB* and *VARRAY* data:

```
[segment_attr_clause] [[NO] COMPRESS [int]] [OVERFLOW segment_attr_clause]
[partition_level_subpartition_clause]
[ { LOB { (lob_item[, ...]) STORE AS lob_param_clause |
  (lob_item) STORE AS {lob_segnane (log_param_clause) |
  log_segnane | (log_param_clause)} } | 
  VARRAY varray_item [{[[ELEMENT] IS OF [TYPE] (ONLY type_name) |
  [NOT] SUBSTITUTABLE AT ALL LEVELS}] STORE AS LOB { log_segnane |
  log_segnane } (log_param_clause) } | 
  {[[ELEMENT] IS OF [TYPE] (ONLY type_name) |
  [NOT] SUBSTITUTABLE AT ALL LEVELS}] }
```

The *partition_level_subpartition_clause* syntax follows:

```
{SUBPARTITIONS hash_subpartition_qty [STORE IN (tablespace_name[, ...])] | 
  SUBPARTITION subpartition_name [VALUES {DEFAULT | {val | NULL}[, ...] } |
  [partitioning_storage_clause] }
```

The *partition_storage_clause*, like the table-level *storage_clause* defined earlier, defines how elements of a partition (or subpartition) are stored. The syntax follows:

```
[[TABLESPACE tablespace_name] | [OVERFLOW TABLESPACE tablespace_name] |
  VARRAY varray_item STORE AS LOB log_segnane |
  LOB (lob_item) STORE AS { (TABLESPACE tablespace_name) |
  log_segnane [(TABLESPACE tablespace_name)] }]
```

SYSTEM partitioning is simple because it does not require partitioning key columns or range or list boundaries. Instead, **SYSTEM** partitions are equipartitioned subordinate tables, like nested tables or domain index storage tables, whose parent table is partitioned. If you leave off the *int* variable, Oracle will create one partition called **SYS_Pint**. Otherwise, it will create *int* number of partitions, up to a limit of 1,024K – 1. System partitioned tables are similar to other partitioned or subpartitioned tables, but they do not support the *OVERFLOW* clause within the *table_partition_description* clause.

REFERENCE partitioning is allowable only when the table is created. It enables equipartitioning of a table based on a referential-integrity constraint found in an existing partitioned parent table. All maintenance on the subordinate table with **REFERENCE** partitioning occurs automatically, because operations on the parent partition automatically cascade to the subordinate table.

In this final partitioning example, we'll again recreate the `orders` table using a composite range-hash partition, this time with *LOB* (actually, an *NCLOB* column) and storage elements:

```
CREATE TABLE orders
  (order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NCLOB)
PARTITION BY RANGE(order_date)
SUBPARTITION BY HASH(cust_shp_id)
  (PARTITION pre_yr_2000 VALUES LESS THAN
    TO_DATE('01-JAN-2000', 'DD-MON-YYYY') TABLESPACE tlspc01
    LOB (order_desc) STORE AS (TABLESPACE tlspc_a01
      STORAGE (INITIAL 10M NEXT 20M) )
    SUBPARTITIONS subpartn_a,
    PARTITION pre_yr_2004 VALUES LESS THAN
      TO_DATE('01-JAN-2004', 'DD-MON-YYYY') TABLESPACE tlspc02
      LOB (order_desc) STORE AS (TABLESPACE tlspc_a02
        STORAGE (INITIAL 25M NEXT 50M) )
    SUBPARTITIONS subpartn_b TABLESPACE tlspc_x07,
    PARTITION post_yr_2004 VALUES LESS THAN
      MAXVALUE (SUBPARTITION subpartn_1,
        SUBPARTITION subpartn_2,
        SUBPARTITION subpartn_3
        SUBPARTITION subpartn_4) )
  ENABLE ROW MOVEMENT;
```

In this somewhat more complex example, we define the `orders` table with the added *NCLOB* table called `order_desc`. In the `pre_yr_2000` and `pre_yr_2004` partitions, we specify that all of the non-*LOB* data goes to tablespaces `tlspc01` and `tlspc02`, respectively. However, the *NCLOB* values of the `order_desc` column will be stored in the `tlspc_a01` and `tlspc_a02` partitions, respectively, with their own unique storage characteristics. Note that the subpartition `subpartn_b` under the partition `pre_yr_2004` is also stored in its own tablespace, `tlspc_x07`. Finally, the last partition (`post_yr_2004`) and its subpartitions are stored in the default tablespace for the `orders` table, because no partition- or subpartition-level *TABLESPACE* clause overrides the default.

Altering partitioned and subpartition tables

Anything about partitions and subpartitions that is explicitly set by the *CREATE TABLE* statement may be altered after the table is created. Many of the clauses shown here (for example, the *SUBPARTITION TEMPLATE* and *MAPPING TABLE* clauses) are merely repetitions of clauses that were described in the earlier section about creating partitioned tables; consequently, descriptions of these clauses will not be repeated. Altering the partitions and/or subpartitions of an Oracle table is governed by this syntax:

```
ALTER TABLE table_name
  [MODIFY DEFAULT ATTRIBUTES [FOR PARTITION partn_name]
    [physical_attributes_clause] [storage_clause] [PCTTHRESHOLD int]
    [{ADD OVERFLOW ... | OVERFLOW ...}] [[NO]COMPRESS]
```

```

[{:LOB (lob_name) | VARRAY varray_name} [(lob_parameter_clause)]]]
[COMPRESS int | NOCOMPRESS]]
[SET SUBPARTITION TEMPLATE {hash_subpartn_quantity |
(SUBPARTITION subpartn_name [partn_list] [storage_clause])}]
[ { SET INTERVAL (expression) |
SET SET STORE IN (tablespace[, ...]) } ]
[MODIFY PARTITION partn_name
{ [table_partition_description] |
[[REBUILD] UNUSABLE LOCAL INDEXES] |
[ADD [subpartn specification]] |
[COALESCE SUBPARTITION [[NO]PARALLEL] [update_index_clause]] |
[{{ADD | DROP} VALUES (partn_value[, ...])} |
[MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSED ...} ] }
[MODIFY SUBPARTITION subpartn_name {hash_subpartn_attributes |
list_subpartn_attributes} ]
[MOVE {PARTITION | SUBPARTITION} partn_name
[MAPPING TABLE] [table_partition_description] [[NO]PARALLEL]
[update_index_clause]]
[ADD PARTITION [partn_name] [table_partition_description]
[[NO]PARALLEL] [update_index_clause]]
[COALESCE PARTITION [[NO]PARALLEL] [update_index_clause]]
[DROP {PARTITION | SUBPARTITION} partn_name [[NO]PARALLEL]
[update_index_clause]]
[RENAME {PARTITION | SUBPARTITION} old_partn_name TO new_partn_name]
[TRUNCATE {PARTITION | SUBPARTITION} partn_name
[{{DROP | REUSE} STORAGE} [[NO]PARALLEL] [update_index_clause]]
[SPLIT {PARTITION | SUBPARTITION} partn_name {AT | VALUES}
(value[, ...])
[INTO (PARTITION [partn_name1]
[table_partition_description],
PARTITION [partn_name2]
[table_partition_description]) |
[[NO]PARALLEL] [update_index_clause]]
[MERGE {PARTITION | SUBPARTITION} partn_name1, partn_name2
[INTO PARTITION [partn_name] [partn_attributes]] [[NO]PARALLEL]
[update_index_clause]]
[EXCHANGE {PARTITION | SUBPARTITION} partn_name WITH TABLE table_name
[{{INCLUDING | EXCLUDING} INDEXES} [{WITH | WITHOUT} VALIDATION]
[[NO]PARALLEL] [update_index_clause] [EXCEPTIONS INTO table_name]]]

```

where:

MODIFY DEFAULT ATTRIBUTES [FOR PARTITION partn_name]

Modifies a wide variety of attributes for the current partition or a specific partition of *partn_name*. Refer to the earlier section “Oracle partitioned and subpartitioned tables” for all the details on partition attributes.

SET SUBPARTITION TEMPLATE {hash_subpartn_quantity | (SUBPARTITION subpartn_name [partn_list] [storage_clause])}

Sets a new subpartition template for the table.

SET INTERVAL (expression | SET SET STORE IN (tablespace[, ...])

Converts a range-partitioned table to an interval-partitioned table or, using **SET STORE IN**, changes the tablespace storage of an existing interval-partitioned table. You can change an interval-partitioned table back to a range-partitioned table using the syntax **SET INTERVAL ()**.

MODIFY PARTITION *partn_name*

Changes a wide variety of physical and storage characteristics, including the storage properties of LOB and VARRAY columns, of a pre-existing partition or subpartition called *partn_name*. Additional syntax may be appended to the **MODIFY PARTITION *partn_name*** clause:

```
{ [table_partition_description] | [[REBUILD] UNUSABLE LOCAL INDEXES] |
  [ADD [subpartn specification]] |
  [COALESCE SUBPARTITION [[NO]PARALLEL] [update_index_clause]
    { [{UPDATE | INVALIDATE} GLOBAL INDEXES] |
      UPDATE INDEXES [ (index_name
        {index_partn | index_subpartn} ), ... ] } ] |
  [{ADD | DROP} VALUES (partn_value, ...)] |
  [MAPPING TABLE {ALLOCATE EXTENT ... | DEALLOCATE UNUSED ...}] }
```

where:

table_partition_description

Described in the earlier section “Oracle partitioned and subpartitioned tables.” This clause may be used on any partitioned table.

[REBUILD] UNUSABLE LOCAL INDEXES

Marks the local index partition as *UNUSABLE*. Adding the optional **REBUILD** keyword tells Oracle to rebuild the unusable local index partition as part of the operation performed by the **MODIFY PARTITION** statement. This clause may not be used with any other subclause of the **MODIFY PARTITION** statement, nor may it be used on tables with subpartitions. This clause may be used on any partitioned table.

ADD [subpartn specification]

Adds a hash or list subpartition specification, as described in the earlier section “Oracle partitioned and subpartitioned tables,” to an existing range partition. This clause may be used to define range-hash or range-list composite partitions only. Oracle populates the new subpartition with rows from other subpartitions using either the hash function or the list values you specify. We recommend the total number of subpartitions be set to a power of 2 for optimal load balancing. You may add range-list subpartitions only if the table does not already have a **DEFAULT** subpartition. When adding range-list subpartitions, the list value clause is required, but it cannot duplicate values found in any other subpartition of the current partition. The only storage or physical attribute you may couple with this clause for both range-hash and range-list subpartitions is the **TABLESPACE** clause. Adding the clause **DEPENDENT TABLES (table_name (partn_specification, ...)) [, ...]** **[{UPDATE | INVALIDATE} [GLOBAL] INDEXES (index_name (index_partn), ...)]** instructs Oracle to cascade partition maintenance and alteration operations on a table to any reference-partitioned child tables (and/or indexes) that may exist.

COALESCE SUBPARTITION [[NO]PARALLEL] [update_index_clause]

Coalesces the subpartition of a range-hash composite partitioned table. This clause tells Oracle to distribute the contents of the last hash subpartition to one or more remaining subpartitions in the set, and then drop the last hash subpartition. Oracle also drops local index subpartitions corresponding to the subpartition you are coalescing. The *update_index_clause* is described later in this list. Global indexes may be updated or invalidated using the syntax **{UPDATE | INVALIDATE} GLOBAL INDEXES**. In addition, local

indexes, index partitions, or index subpartitions may be updated using the syntax *UPDATE INDEXES* (*index_name* ({*index_partn* | *index_subpartn*})).

{*ADD* | *DROP*} *VALUES* (*partn_value*[, ...])

Adds a new value (or values) or drops existing values on an existing list-partitioned table, respectively. Local and global indexes are not affected by this clause. Values cannot be added to or dropped from a *DEFAULT* list partition.

MAPPING TABLE {*ALLOCATE EXTENT* ... | *DEALLOCATE UNUSED* ...}

Defines a mapping table for a partitioned table that is an IOT. The *ALLOCATE EXTENT* and *DEALLOCATE UNUSED* clauses were described earlier, in the syntax description list for the *CREATE TABLE* statement. This clause may be used on any type of partitioned table, as long as the table is an index-organized table.

MODIFY SUBPARTITION *subpartn_name* {*hash_subpartn_attributes* | *list_subpartn_attributes*}

Modifies a specific hash or list subpartition according to the subpartition attributes described in the earlier section “Oracle partitioned and subpartitioned tables.”

MOVE {*PARTITION* | *SUBPARTITION*} *partn_name* [*MAPPING TABLE*] [*table_partition_description*] [[*NO*]*PARALLEL*] [*update_index_clause*]

Moves a specified partition (or subpartition) of *partn_name* to another partition (or subpartition) described in the *table_partition_description* clause. Moving a partition is I/O-intensive, so the optional *PARALLEL* clause may be used to parallelize the operation. When it’s omitted, *NOPARALLEL* is the default. In addition, you may optionally update or invalidate the local and global index, as described in the *update_index_clause* discussed later in this list.

ADD PARTITION [*partn_name*] [*table_partition_description*] [[*NO*]*PARALLEL*] [*update_index_clause*]

Adds a new partition (or subpartition) of *partn_name* to the table. The *ADD PARTITION* clause supports all aspects of creating a new partition or subpartition, via the *table_partition_description* clause. Adding a partition may be I/O-intensive, so the optional *PARALLEL* clause may be used to parallelize the operation. When it’s omitted, *NOPARALLEL* is the default. In addition, you may optionally update or invalidate local and global indexes on the table using the *update_index_clause*.

update_index_clause

Controls the status assigned to indexes once the partitions and/or subpartitions of a table are altered. By default, Oracle invalidates the entire index(es) of a table, not just those portions of the index on the partition and/or subpartition being altered. You may update or invalidate global indexes on the table or update one or more specific index(es) on the table, respectively, using this syntax:

[{*UPDATE* | *INVALIDATE*} *GLOBAL INDEXES*] |
 UPDATE INDEXES [(*index_name* ({*index_partn*|*index_subpartn*}))[, ...]]

COALESCE PARTITION [[*NO*]*PARALLEL*] [*update_index_clause*]

Takes the contents of the last partition of a set of hash partitions and rehashes the contents to one or more of the other partitions in the set. The last partition is then dropped. Obviously, this clause is only for use with hash partitions. The *update_index_clause* may be applied to update or invalidate the local and/or global indexes of the table being coalesced.

`DROP {PARTITION | SUBPARTITION} partn_name [[NO]PARALLEL] [update_index_clause]`

Drops an existing range or list partition or subpartition of *partn_name* from the table. The data within the partition is also dropped. If you want to keep the data, use the *MERGE PARTITION* clause. If you want to get rid of a hash partition or subpartition, use the *COALESCE PARTITION* clause. Tables with only a single partition are not affected by the *ALTER TABLE...DROP PARTITION* statement; instead, use the *DROP TABLE* statement.

`RENAME {PARTITION | SUBPARTITION} old_partn_name TO new_partn_name`

Renames an existing partition or subpartition of *old_partn_name* to a new name of *new_partn_name*.

`TRUNCATE {PARTITION | SUBPARTITION} partn_name [[DROP | REUSE] STORAGE] [[NO]PARALLEL] [update_index_clause]`

Removes all of the rows of a partition or subpartition of *partn_name*. If you truncate a composite partition, all the rows of the subpartition(s) are also dropped. On IOTs, mapping table partitions and overflow partitions are also truncated. *LOB* data and index segments, if the table has any *LOB* columns, are also truncated. Finally, disable any existing referential integrity constraints on the data, or else delete the rows from the table first, then truncate the partition or subpartition. The optional *DROP* and *REUSE STORAGE* subclauses define whether the space freed by the truncated data is made available for other objects in the tablespace or remains allocated to the original partition or subpartition.

`SPLIT {PARTITION | SUBPARTITION} partn_name {AT | VALUES} (value[, ...]) [INTO (PARTITION [partn_name1] [table_partition_description]), (PARTITION [partn_name2] [table_partition_description])] [[NO]PARALLEL] [update_index_clause]`

Creates from the current partition (or subpartition) identified by *partn_name* two new partitions (or subpartitions) called *partn_name1* and *partn_name2*. These new partitions may have their own complete specification, as defined by the *table_partition_description* clause. When such a specification is omitted, the new partitions inherit all physical characteristics of the current partition. When splitting a *DEFAULT* partition, all of the split values go to *partn_name1*, while all of the default values go to *partn_name2*. For IOTs, Oracle splits any mapping table partition in a manner corresponding to the split. Oracle also splits *LOB* and *OVERFLOW* segments, but you may specify your own *LOB* and *OVERFLOW* storage characteristics, as described in the earlier section on LOBs.

`{AT | VALUES} (value[, ...])`

Splits range partitions (using *AT*) or list partitions (using *VALUES*) according to the *value*(s) you specify. The *AT* (*value*[, ...]) clause defines the new noninclusive upper range for the first of the two new partitions. The new value should be less than the partition boundary of the current partition, but greater than the partition boundary of the next lowest partition (if one exists). The *VALUES* (*value*1[, ...]) clause defines the values to go into the first of the two new list partitions. The first new list partition is built from *value*1, and the second is built from the remaining partition values in the current partition of *partn_name*. The value list must include values that already exist in the current partition, but it cannot contain all of the values of the current partition.

`INTO (PARTITION [partn_name1] [table_partition_description]), (PARTITION [partn_name2] [table_partition_description])`

Defines the two new partitions that result from the split. At a minimum, the two *PARTITION* keywords, in parentheses, are required. Any characteristics not explicitly declared for the new partitions are inherited from the current partition of *partn_name*, including any subpartitioning. There are a few restrictions to note. When subpartitioning range-hash composite partitioned tables, only the *TABLESPACE* value is allowed for the subpartitions. Subpartitioning is not allowed at all when splitting range-list composite partitioned tables. Any indexes on heap-organized tables are invalidated by default when the table is split. You must use the *update_index_clause* to update their status.

`MERGE {PARTITION | SUBPARTITION} partn_name1, partn_name2 [INTO PARTITION [partn_name] [partn_attributes]] [[NO]PARALLEL] [update_index_clause]`

Merges the contents of two or more partitions or subpartitions of a table into a single new partition. Oracle then drops the two original partitions. Merged range partitions must be adjacent and are then bound by the higher boundary of the original two partitions when merged. Merged list partitions need not be adjacent and result in a single new partition with a union of the two sets of partition values. If one of the list partitions was the *DEFAULT* partition, the new partition will be the *DEFAULT*. Merged range-list composite partitions are allowed but may not have a new subpartition template. Oracle creates a subpartition template from the existing one(s) or, if none exist, creates a new *DEFAULT* subpartition. Physical attributes not defined explicitly are inherited from the table-level settings. By default, Oracle makes all local index partitions and global indexes *UNUSABLE* unless you override this behavior using the *update_index_clause*. (The exception to this rule is with IOTs, which, being index-based, will remain *USABLE* throughout the merge operation.) Merge operations are not allowed on hash-partitioned tables; use the *COALESCE PARTITION* clause instead.

`EXCHANGE {PARTITION | SUBPARTITION} partn_name WITH TABLE table_name [{INCLUDING | EXCLUDING} INDEXES] [{WITH | WITHOUT} VALIDATION] [[NO]PARALLEL] [update_index_clause] [EXCEPTIONS INTO table_name]`

Exchanges the data and index segments of a nonpartitioned table with those of a partitioned table, or the data and index segments of a partitioned table of one type with those of a partitioned table of another type. The structure of the tables in the exchange must be identical, including the same primary key. All segment attributes (e.g., tablespaces, logging, and statistics) of the current partitioned table, called *partn_name*, and the table it is being exchanged with, called *table_name*, are exchanged. Tables containing *LOB* columns will also exchange *LOB* data and index segments. Additional syntax details that have not previously been defined elsewhere in this list follow:

`WITH TABLE table_name`

Defines the table that will exchange segments with the current partition or subpartition.

`{INCLUDING | EXCLUDING} INDEXES`

Exchanges local index partitions or subpartitions with the table index (on nonpartitioned tables) or the local index (on hash-partitioned tables), using the *INCLUDING INDEXES* clause. Alternately, marks all index partitions and subpartitions as well as regular indexes and partitioned indexes of the exchanged table with the *UNUSABLE* status, using the *EXCLUDING INDEXES* clause.

{WITH | WITHOUT} VALIDATION

Returns errors when any rows in the current table fail to map into a partition or subpartition of the exchanged table, using the *WITH VALIDATION* clause. Otherwise, the *WITHOUT VALIDATION* clause may be included to skip checking of row mapping between the tables.

EXCEPTIONS INTO *table_name*

Places the ROWIDs of all rows violating a *UNIQUE* constraint (in *DISABLE VALIDATE* state) on the partitioned table. When this clause is omitted, Oracle assumes there is a table in the current schema called *EXCEPTIONS*. The *EXCEPTIONS* table is defined in the *utlexcpt.sql* and *utlexpt1.sql* scripts that ship with Oracle. Refer to the Oracle documentation if you need these scripts.

There are a couple of caveats to remember about altering a partitioned table. First, altering a partition on a table that serves as the source for one or more materialized views requires that the materialized views be refreshed. Second, bitmap join indexes defined on the partitioned table being altered will be marked *UNUSABLE*. Third, several restrictions apply if the partitions (or subpartitions) are ever spread across tablespaces that use different block sizes. Refer to the Oracle documentation when attempting these sorts of alterations to a partitioned table.

In the next few examples, assume we are using the **orders** table, partitioned as shown here:

```
CREATE TABLE orders
  (order_number NUMBER,
  order_date    DATE,
  cust_nbr      NUMBER,
  price         NUMBER,
  qty           NUMBER,
  cust_shp_id   NUMBER)
PARTITION BY RANGE(order_date)
  (PARTITION pre_yr_2000 VALUES LESS THAN
    TO_DATE('01-JAN-2000', 'DD-MON-YYYY'),
  PARTITION pre_yr_2004 VALUES LESS THAN
    TO_DATE('01-JAN-2004', 'DD-MON-YYYY')
  PARTITION post_yr_2004 VALUES LESS THAN
    MAXVALUE) ;
```

The following statement will mark all of the local index partitions as *UNUSABLE* in the **orders** table for the **post_yr_2004** partition:

```
ALTER TABLE orders MODIFY PARTITION post_yr_2004
  UNUSABLE LOCAL INDEXES;
```

However, say we've decided to now split the **orders** table partition **post_yr_2004** into two new partitions, **pre_yr_2008** and **post_yr_2008**. Values that are now less than **MAXVALUE** will be stored in the **post_yr_2008** partition, while values less than '01-JAN-2008' will be stored in **pre_yr_2008**:

```
ALTER TABLE orders SPLIT PARTITION post_yr_2004
  AT (TO_DATE('01-JAN-2008', 'DD-MON-YYYY'))
  INTO (PARTITION pre_yr_2008, PARTITION post_yr_2008);
```

Assuming that the **orders** table contained a *LOB* or a *VARRAY* column, we could further refine the alteration by including additional details for handling these columns, while also updating the global indexes as the operation completes:

```
ALTER TABLE orders SPLIT PARTITION post_yr_2004
  AT (TO_DATE('01-JAN-2008','DD-MON-YYYY'))
  INTO
    (PARTITION pre_yr_2008
      LOB (order_desc) STORE AS (TABLESPACE order_tblspc_a1),
     PARTITION post_yr_2008)
      LOB (order_desc) STORE AS (TABLESPACE order_tblspc_a1) )
  UPDATE GLOBAL INDEXES;
```

Now, assuming the **orders** table has been grown at the upper end, let's merge together the partitions at the lower end:

```
ALTER TABLE orders
  MERGE PARTITIONS pre_yr_2000, pre_yr_2004
  INTO PARTITION yrs_2004_and_earlier;
```

After a few more years have passed, we might want to get rid of the oldest partition, or at least give it a better name:

```
ALTER TABLE orders DROP PARTITION yrs_2004_and_earlier;
ALTER TABLE orders RENAME PARTITION yrs_2004_and_earlier TO pre_yr_2004;
```

Finally, let's truncate a table partition, delete all of its data, and return the empty space for use by other objects in the tablespace:

```
ALTER TABLE orders
  TRUNCATE PARTITION pre_yr_2004
  DROP STORAGE;
```

As these examples illustrate, anything related to partitioning and subpartitioning that can be created with the Oracle *CREATE TABLE* statement can later be changed, augmented, or cut down using the Oracle *ALTER TABLE* statement.

Organized tables: heaps, IOTs, and external tables

Oracle 11g offers powerful means of controlling the physical storage behavior of tables.

The most useful aspect of the *ORGANIZATION HEAP* clause is that you can now compress an entire table within Oracle. This is extremely useful for reducing disk storage costs in database environments with multiterabyte tables. The following example creates the **orders** table in a compressed and logged heap, along with a primary key constraint and storage details:

```
CREATE TABLE orders
  (order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NCLOB,
  CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
  ORGANIZATION HEAP
    COMPRESS LOGGING
    PCTTHRESHOLD 2
    STORAGE
      (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
    OVERFLOW STORAGE
      (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
    ENABLE ROW MOVEMENT;
```

To define the same table using an index-organized table based on the `order_date` column, we would use this syntax:

```
CREATE TABLE orders
  (order_number NUMBER,
  order_date DATE,
  cust_nbr NUMBER,
  price NUMBER,
  qty NUMBER,
  cust_shp_id NUMBER,
  shp_region VARCHAR2(20),
  order_desc NCLOB,
  CONSTRAINT ord_nbr_pk PRIMARY KEY (order_number) )
  ORGANIZATION HEAP
    INCLUDING order_date
  PCTTHRESHOLD 2
  STORAGE
    (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
  OVERFLOW STORAGE
    (INITIAL 4M NEXT 2M PCTINCREASE 0 MINEXTENTS 1 MAXEXTENTS 1)
  ENABLE ROW MOVEMENT;
```

Finally, we'll create an external table that stores our customer shipping information, called `cust_shipping_external`. The code shown in bold is the *opaque_format_spec*:

```
CREATE TABLE cust_shipping_external
  (external_cust_nbr NUMBER(6),
  cust_shp_id NUMBER,
  shipping_company VARCHAR2(25) )
  ORGANIZATION EXTERNAL
    (TYPE oracle_loader
    DEFAULT DIRECTORY dataloader
    ACCESS PARAMETERS
      (RECORDS DELIMITED BY newline
      BADFILE 'upload_shipping.bad'
      DISCARDFILE 'upload_shipping.dis'
      LOGFILE 'upload_shipping.log'
      SKIP 20
      FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY ''
      (client_id INTEGER EXTERNAL(6),
      shp_id CHAR(20),
      shipper CHAR(25) ))
    LOCATION ('upload_shipping.ctl') )
  REJECT LIMIT UNLIMITED;
```

In this example, the external table type is `ORACLE_LOADER` and the default directory is `DATALOADER`. This example illustrates the fact that you define the metadata of the table within Oracle and then describe how that metadata references a data source outside of the Oracle database server itself.

Oracle XMLType and object-type tables

When an Oracle XMLType table is created, Oracle automatically stores the data in a `CLOB` column, unless you create an XML schema-based table. (For details on Oracle's XML support, see Oracle's XMLDB Developer's Guide.) The following code example first creates an XMLType table, `distributors`, with the implicit `CLOB` data storage, then creates a second such table, `suppliers`, with a more sophisticated XML-schema definition:

```
CREATE TABLE distributors OF XMLTYPE;
CREATE TABLE suppliers OF XMLTYPE
XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
ELEMENT "vendors";
```

A key advantage of tables based on XML schemas is that you can create B-tree indexes on them. In the following example, we create an index on **suppliercity**:

```
CREATE INDEX suppliercity-index
ON suppliers
(S."XMLDATA"."ADDRESS"."CITY");
```

You may similarly create tables using a mix of standard and *XMLTYPE* columns. In this case, the *XMLTYPE* column may store its data as a *CLOB*, or it may store its data in an object-relational column of a structure determined by your specification. For example, we'll recreate the **distributors** table (this time with some added storage specifications) and the **suppliers** table with both standard and *XMLTYPE* columns:

```
CREATE TABLE distributors
(distributor_id NUMBER,
distributor_spec XMLTYPE)
XMLTYPE distributor_spec
STORE AS CLOB
(TABLESPACE tblspc_dist
STORAGE (INITIAL 10M NEXT 5M)
CHUNK 4000
NOCACHE
LOGGING);
CREATE TABLE suppliers
(supplier_id NUMBER,
supplier_spec XMLTYPE)
XMLTYPE supplier_spec STORE AS OBJECT RELATIONAL
XMLSCHEMA "http://www.lookatallthisstuff.com/suppliers.xsd"
ELEMENT "vendors"
OBJECT IDENTIFIER IS SYSTEM GENERATED
OIDINDEX vendor_ndx TABLESPACE tblspc_xml_vendors;
```

When creating XML and object tables, you may refer to *inline_ref_constraint* and *table_ref_constraint* clauses. The syntax for an *inline_ref_constraint* clause is:

```
{SCOPE IS scope_table |
WITH ROWID |
[CONSTRAINT constraint_name] REFERENCES object [ (column_name) ]
[ON DELETE {CASCADE | SET NULL}]
[constraint_state]}
```

The only difference between an inline reference constraint and a table reference constraint is that inline reference constraints operate at the column level and table reference constraints operate at the table level. (This is essentially the same behavior and coding rule of standard relational constraints like *PRIMARY KEY* or *FOREIGN KEY*.) The syntax for a *table_ref_constraint* follows:

```
{SCOPE FOR (ref_col | ref_attr) IS scope_table |
REF (ref_col | ref_attr) WITH ROWID |
[CONSTRAINT constraint_name] FOREIGN KEY (ref_col | ref_attr)
REFERENCES object [ (column_name) ]
[ON DELETE {CASCADE | SET NULL}]
[constraint_state]}
```

The *constraint_state* clause contains a number of options that have already been defined earlier in the discussion of the Oracle *CREATE TABLE* statement. However, these options are applied only to the condition of the scope reference:

```
[NOT] DEFERRABLE
INITIALLY {IMMEDIATE | DEFERRED}
{ENABLE | DISABLE}
{VALIDATE | NOVALIDATE}
{RELY | NORELY}
EXCEPTIONS INTO table_name
USING INDEX {index_name | ( create_index_statement ) | index_attributes}
```

Object-type tables are useful for creating tables containing user-defined types. For example, the following code creates the *building_type* type:

```
CREATE TYPE OR REPLACE building_type AS OBJECT
  (building_name VARCHAR2(100),
   building_address VARCHAR2(200));
```

We can then create a table called **offices_object_table** that contains the object and defines some of its characteristics, such as OID information. In addition, we'll create two more tables, based upon *building_type*, that reference the object type as an *inline_ref_constraint* and a *table_ref_constraint*, respectively:

```
CREATE TABLE offices_object_table
  OF building_type (building_name PRIMARY KEY)
  OBJECT IDENTIFIER IS PRIMARY KEY;
CREATE TABLE leased_offices
  (office_nbr NUMBER,
   rent DEC(9,3),
   office_ref REF building_type
     SCOPE IS offices_object_table);
CREATE TABLE owned_offices
  (office_nbr NUMBER,
   payment DEC(9,3),
   office_ref REF building_type
     CONSTRAINT offc_in_bld REFERENCES offices_object_table);
```

In these examples, the *SCOPE IS* clause defines the *inline_ref_constraint*, while the *CONSTRAINT* clause defines the *table_ref_constraint*.

Oracle ALTER TABLE

When using the Oracle command *ALTER TABLE*, you are able to *ADD*, *DROP*, or *MODIFY* every aspect of each element of the table. For example, the syntax diagram shows that the method for adding or modifying an existing column includes its attributes, but you need to explicitly state that the *attributes* include any Oracle-specific extensions. So, while the ANSI standard only lets you modify attributes such as *DEFAULT* or *NOT NULL* (as well as column-level constraints assigned to the column), Oracle also allows you to alter any special characteristics that might exist, such as *LOB*, *VARRAY*, *NESTED TABLE*, index-organized table, *CLUSTER*, or *PARTITION* settings.

For example, the following code adds a new column to a table in Oracle and adds a new, unique constraint to that table:

```
ALTER TABLE titles
  ADD subtitle VARCHAR2(32) NULL
  CONSTRAINT unq_subtitle UNIQUE;
```

When a foreign key constraint is added to a table, the DBMS verifies that all existing data in the table meets that constraint. If not, the *ALTER TABLE* fails.



Any queries that use *SELECT ** return the new columns, even if this was not planned. Precompiled objects, such as stored procedures, can return any new columns if they use the *%ROWTYPE* attribute. Otherwise, a precompiled object may not return any new columns.

Oracle also allows you to perform multiple actions, such as *ADD* or *MODIFY*, on multiple columns by enclosing the actions within parentheses. For example, the following command adds several columns to a table with a single statement:

```
ALTER TABLE titles
  ADD (subtitles VARCHAR2(32) NULL,
       year_of_copyright INT,
       date_of_origin DATE);
```

PostgreSQL

PostgreSQL supports the ANSI standards for *CREATE* and *ALTER TABLE*, with a couple of extensions that enable you to quickly build a new table from existing table definitions. Following is the syntax for *CREATE TABLE*:

```
CREATE [LOCAL] [[TEMP]ORARY] TABLE table_name
  (column_name datatype attributes[, ...]) |
  CONSTRAINT constraint_name [{NULL | NOT NULL}]
  {[UNIQUE] | [PRIMARY KEY (column_name[, ...])] | [CHECK (expression)] | 
  REFERENCES reference_table (reference_column[, ...])
    [MATCH {FULL | PARTIAL | default}]
    [ON {UPDATE | DELETE}]
    {CASCADE | NO ACTION | RESTRICT | SET NULL | SET DEFAULT value}]
    {[NOT] DEFERRABLE} [INITIALLY {DEFERRED | IMMEDIATE}]}, ... | 
  [table_constraint][, ...]
  [WITH[OUT] OIDE]
  [INHERITS (inherited_table[, ...])]
  [ON COMMIT {DELETE | PRESERVE} ROWS]
  [AS select_statement]
```

And the PostgreSQL syntax for *ALTER TABLE* is:

```
ALTER TABLE [ONLY] table_name [*]
  [ADD [COLUMN] column_name datatype attributes [...]]
  | [ALTER [COLUMN] column_name
    {SET DEFAULT value | DROP DEFAULT | SET STATISTICS int}]
    | [RENAME [COLUMN] column_name TO new_column_name]
    | [RENAME TO new_table_name]
    | [ADD table_constraint]
    | [DROP CONSTRAINT constraint_name RESTRICT]
    | [OWNER TO new_owner]
```

The parameters are as follows:

REFERENCES...MATCH...ON {UPDATE | DELETE}...

Checks a value inserted into the column against the values of a column in another table. This clause can also be used as part of a *FOREIGN KEY* declaration. The *MATCH* options are *FULL*, *PARTIAL*, and the default, where *MATCH* has no other keyword. *FULL* match forces all columns of a multicolumn foreign key

either to be NULL or to contain a valid value. The default allows mixed NULLs and values. *PARTIAL* matching is a valid syntax, but is not supported. The *REFERENCES* clause also allows several different behaviors to be declared for *ON DELETE* and/or *ON UPDATE* referential integrity:

NO ACTION

Produces an error when the foreign key is violated (the default).

RESTRICT

Synonym for *NO ACTION*.

CASCADE

Sets the value of the referencing column to the value of the referenced column.

SET NULL

Sets the value of the referencing column to NULL.

SET DEFAULT *value*

Sets the referencing column to its declared default value or NULL, if no default value exists.

[NOT] DEFERRABLE [INITIALLY {DEFERRED | IMMEDIATE}]

The *DEFERRABLE* option of the *REFERENCES* clause tells PostgreSQL to defer evaluation of all constraints until the end of a transaction. *NOT DEFERRABLE* is the default behavior for the *REFERENCES* clause. Similar to the *DEFERRABLE* clause is the *INITIALLY* clause: specifying *INITIALLY DEFERRED* checks constraints at the end of a transaction; *INITIALLY IMMEDIATE* checks constraints after each statement (the default).

FOREIGN KEY

Can be declared only as a table-level constraint, not as a column-level constraint. All options for the *REFERENCES* clause are supported as part of the *FOREIGN KEY* clause. The syntax follows:

[FOREIGN KEY (*column_name*[, ...]) REFERENCES...]

WITH[OUT] OIDS

Specifies whether rows of the new table should have automatically assigned object IDs (*WITH OIDS*) or not (*WITHOUT OIDS*). The default is *WITH OIDS*. *OIDs* are assigned based on a 32-bit counter. Once the counter is surpassed, the *OID* numbers cycle over and uniqueness can no longer be guaranteed. We recommend that you add a unique index to any *OID* columns you specify to avoid an error with duplicate *OIDs* on very large tables. A table with an inherited structure receives the *OID* setting of its parent table.

INHERITS *inherited_table*

Specifies a table or tables from which the table you are creating inherits all columns. The newly created table also inherits functions attached to tables higher in the hierarchy. If any inherited column appears more than once, the statement fails.

ON COMMIT {DELETE | PRESERVE} ROWS

Used only with temporary tables. This clause controls the behavior of the temporary table after records are committed to the table. *ON COMMIT DELETE ROWS* clears the temporary table of all rows after each commit. This is the default if the *ON COMMIT* clause is omitted. *ON COMMIT PRESERVE ROWS* saves the rows in the temporary table after the transaction has committed.

AS *select_statement*

Enables you to create and populate a table with data from a valid *SELECT* statement. The column names and datatypes do not need to be defined, since they are inherited from the query. The *CREATE TABLE...AS* statement has similar functionality to *SELECT...INTO*, but is more readable.

ONLY

Specifies that only the named table is affected by the *ALTER TABLE* statement, not any parent or subtables in the table hierarchy.

OWNER TO *new_owner*

Changes the owner of the table to the user identified by *new_owner*.

A PostgreSQL table cannot have more than 1,600 columns. However, you should limit the number of columns to well below 1,600, for performance reasons. For example:

```
CREATE TABLE distributors
  (name  VARCHAR(40) DEFAULT 'Thomas Nash Distributors',
   dist_id INTEGER DEFAULT NEXTVAL('dist_serial'),
   modtime TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```



Unique to PostgreSQL is the ability to create column-level constraints with multiple columns. Since PostgreSQL also supports standard table-level constraints, the ANSI-standard approach is still the recommended approach.

PostgreSQL's implementation of *ALTER TABLE* allows the addition of extra columns using the *ADD* keyword. Existing columns may have new default values assigned to them using *ALTER COLUMN...SET DEFAULT*, while *ALTER COLUMN...DROP DEFAULT* allows the complete erasure of a column-based default. In addition, new defaults may be added to columns using the *ALTER* clause, but only newly inserted rows will be affected by such new default values. *RENAME* allows new names for existing columns and tables.

SQL Server

SQL Server offers a plethora of options when defining or altering a table, its columns, and its table-level constraints. Its *CREATE TABLE* syntax is:

```
CREATE TABLE table_name
  (column_name datatype { [DEFAULT default_value]
    | [IDENTITY [(seed,increment) [NOT FOR REPLICATION]]]
    | [ROWGUIDCOL] [NULL | NOT NULL]
    | [{PRIMARY KEY | UNIQUE}
      [CLUSTERED | NONCLUSTERED]
      [WITH FILLFACTOR = int] [ON {filegroup | DEFAULT}]]
    | [[FOREIGN KEY]
      REFERENCES reference_table [(reference_column[, ...])]
      [ON {DELETE | UPDATE} {CASCADE | NO ACTION}]
      [NOT FOR REPLICATION]]
    | [CHECK [NOT FOR REPLICATION] (expression)]
    | [COLLATE collation_name]
  | column_name AS computed_column_expression }
  [, ...]
  | [table_constraint][, ...])
  [ON {filegroup | DEFAULT}]
  [TEXTIMAGE_ON {filegroup | DEFAULT}])
```

And the SQL Server version of *ALTER TABLE* is:

```
ALTER TABLE table_name
  [ALTER COLUMN column_name new_datatype attributes {ADD | DROP} ROWGUIDCOL]
  | [ADD [COLUMN] column_name datatype attributes][, ...]
  | [WITH CHECK | WITH NOCHECK] ADD table_constraint][, ...]
  | [DROP { [CONSTRAINT] constraint_name | COLUMN column_name }][, ...]
  | [{CHECK | NOCHECK} CONSTRAINT { ALL | constraint_name[, ...] }]
  | [{ENABLE | DISABLE} TRIGGER { ALL | trigger_name[, ...] }]
```

The parameters are as follows:

DEFAULT default_value

Applies to any column except those with a *TIMESTAMP* datatype or an *IDENTITY* property. The *default_value* must be a constant value such as a character string or a number, a system function such as *GETDATE()*, or *NULL*.

IDENTITY [(seed, increment)]

Creates and populates the column with a monotonically increasing number when applied to an integer column. The *IDENTITY* starts counting at the value of *seed* and increases by the value of *increment*. When either is omitted, the default is 1.

NOT FOR REPLICATION

Specifies that the values of an *IDENTITY* or *FOREIGN KEY* are not replicated to subscribing servers. This helps in situations in which different servers require the same table structures, but not the exact same data.

ROWGUIDCOL

Identifies a column as a globally unique identifier (GUID), which ensures no two values are ever repeated across any number of servers. Only one such column may be identified per table. This clause does not, however, create the unique values itself. They must be inserted using the *NEWID* function.

(PRIMARY KEY | UNIQUE)

Defines a unique or primary key constraint for the table. The primary key declaration differs from the ANSI standard by allowing you to assign the clustered or nonclustered attributes on the primary key index, as well as a starting fillfactor. (Refer to the section “*PRIMARY KEY Constraints*” in Chapter 2 for more information.) The attributes of a unique or primary key include:

CLUSTERED | NONCLUSTERED

Declares that the column or columns of the primary key set the physical sort order of the records in the table (*CLUSTERED*), or that the primary key index maintains pointers to all of the records of the table (*NONCLUSTERED*). *CLUSTERED* is the default when this clause is omitted.

WITH FILLFACTOR = int

Declares that a percentage of space (*int*) should remain free on each data page when the table is created. SQL Server does not maintain *FILLFACTOR* over time, so you should rebuild the index on a regular basis.

ON {filegroup | DEFAULT}

Specifies that the primary key either is located on the pre-existing, named filegroup or is assigned to the *DEFAULT* filegroup.

FOREIGN KEY

Checks values as they are inserted into the table against a column in another table in order to maintain referential integrity. Foreign keys are described in detail in

Chapter 2. A foreign key can only reference columns that are defined as a *PRIMARY KEY* or *UNIQUE* index on the referencing table. A referential action may be specified to take place on the *reference_table* when the record is deleted or updated, according to the following:

ON {DELETE | UPDATE}

Specifies that an action needs to happen in the local table when either (or both) an *UPDATE* or *DELETE* occurs on the referenced table.

CASCADE

Specifies that any *DELETE* or *UPDATE* also takes place on the referring table for any records dependent on the value of the *FOREIGN KEY*.

NO ACTION

Specifies that no action occurs on the referring table when a record on the current table is deleted or updated.

NOT FOR REPLICATION

Specifies that an *IDENTITY* property should not be enforced when the data is replicated from another database. This ensures that data from the published server is not assigned new identity values.

CHECK

Ensures that a value inserted into the specified column of the table is a valid value, based on the *CHECK* expression. For example, the following shows a table with two column-level *CHECK* constraints:

```
CREATE TABLE people
  (people_id      CHAR(4)
    CONSTRAINT pk_dist_id PRIMARY KEY CLUSTERED
    CONSTRAINT ck_dist_id CHECK (dist_id LIKE '
      [A-Z][A-Z][A-Z][A-Z]'),
  people_name    VARCHAR(40) NULL,
  people_addr1   VARCHAR(40) NULL,
  people_addr2   VARCHAR(40) NULL,
  city           VARCHAR(20) NULL,
  state          CHAR(2)    NULL
    CONSTRAINT def_st DEFAULT ("CA")
    CONSTRAINT chk_st REFERENCES states(state_ID),
  zip            CHAR(5)    NULL
    CONSTRAINT ck_dist_zip
    CHECK(zip LIKE '[0-9][0-9][0-9][0-9][0-9]'),
  phone          CHAR(12)   NULL,
  sales_rep      empid     NOT NULL DEFAULT USER)
```

GO

The *CHECK* constraint on *people_id* ensures an all-alphabetic ID, while the one on *zip* ensures an all-numeric value. The *REFERENCES* constraint on *state* performs a lookup on the *states* table. The *REFERENCES* constraint is essentially the same as a *CHECK* constraint, except that it derives its list of acceptable values from the values stored in another column. This example illustrates how column-level constraints are named using the *CONSTRAINT constraint_name* syntax.

COLLATE

Allows programmers to change, on a column-by-column basis, the sort order and character set that is used by the column.

TEXTIMAGE_ON {filegroup | DEFAULT}

Controls the placement of **text**, **ntext**, and **image** columns, allowing you to place *LOB* data on the pre-existing *filegroup* of your choice. When omitted, these columns are stored in the default filegroup with all other tables and database objects.

WITH [NO]CHECK

Tells SQL Server whether the data in the table should be validated against any newly added constraints or keys. When constraints are added using **WITH NOCHECK**, the query optimizer ignores them until they are enabled via **ALTER TABLE table_name CHECK CONSTRAINT ALL**. When constraints are added using **WITH CHECK**, the constraints are checked immediately against all data already in the table.

[NO]CHECK CONSTRAINT

Enables an existing constraint with **CHECK CONSTRAINT** or disables one with **NOCHECK CONSTRAINT**.

{ENABLE | DISABLE} TRIGGER { ALL | trigger_name[,...] }

Enables or disables the specified trigger or triggers, respectively. All triggers on the table may be enabled or disabled by substituting the keyword **ALL** for the table name, as in **ALTER TABLE employee DISABLE TRIGGER ALL**. You may, alternately, disable or enable a single *trigger_name* or more than one trigger by placing each *trigger_name* in a comma-delimited list.

SQL Server allows any column-level constraint to be named by specifying **CONSTRAINT constraint_name...**, and then the text of the constraint. Several constraints may be applied to a single column, as long as they are not mutually exclusive (for example, **PRIMARY KEY** and **NULL**).

SQL Server also allows a local temporary table to be created, but not using ANSI syntax. A local temporary table, which is stored in the **tempdb** database, requires a prefix of a single pound sign (#) to the name of the table. The local temporary table is usable by the person or process that created it and is deleted when the person logs out or the process terminates. A global temporary table, which is usable by all people and processes that are currently logged in/running, can be established by prefixing two pound signs (##) to the name of the table. The global temporary table is deleted when its process terminates or its creator logs out.

SQL Server also allows the creation of tables with columns that contain computed values. Such a column does not actually contain data; instead, it is a virtual column containing an expression using other columns already in the table. For example, a computed column could contain an expression such as **order_cost AS (price * qty)**. Computed columns also can contain constants, functions, variables, noncomputed columns, or any of these combined with each other using operators.

Any of the column-level constraints shown earlier also may be declared at the table level. That is, **PRIMARY KEY** constraints, **FOREIGN KEY** constraints, **CHECK** constraints, and others may be declared after all the columns have been defined in the **CREATE TABLE** statement. This is very useful for constraints that cover more than one column. For example, a column-level **UNIQUE** constraint can be applied only to that column. However, declaring the constraint at the table level allows it to span several columns. Here is an example of both column- and table-level constraints:

```
-- Creating a column-level constraint
CREATE TABLE favorite_books
  (isbn      CHAR(100)  PRIMARY KEY NONCLUSTERED,
```

```

book_name      VARCHAR(40)  UNIQUE,
category       VARCHAR(40)  NULL,
subcategory    VARCHAR(40)  NULL,
pub_date       DATETIME    NOT NULL,
purchase_date  DATETIME    NOT NULL)
GO
-- Creating a table-level constraint
CREATE TABLE favorite_books
(isbn          CHAR(100)   NOT NULL,
book_name     VARCHAR(40)  NOT NULL,
category      VARCHAR(40)  NULL,
subcategory   VARCHAR(40)  NULL,
pub_date      DATETIME    NOT NULL,
purchase_date DATETIME    NOT NULL,
CONSTRAINT pk_book_id PRIMARY KEY NONCLUSTERED (isbn)
    WITH FILLFACTOR=70,
CONSTRAINT unq_book UNIQUE CLUSTERED (book_name, pub_date))
GO

```

These two commands provide nearly the same results, except that the table-level *UNIQUE* constraint has two columns, whereas only one column is included in the column-level *UNIQUE* constraint.

The following example adds a new *CHECK* constraint to a table, but does not check to ensure that the existing values in the table pass the constraint:

```

ALTER TABLE favorite_book WITH NOCHECK
ADD CONSTRAINT extra_check CHECK (ISBN > 1)
GO

```

In this example, we further add a new column with an assigned *DEFAULT* value that is placed in each existing record in the table:

```

ALTER TABLE favorite_book ADD reprint_nbr INT NULL
CONSTRAINT add_reprint_nbr DEFAULT 1 WITH VALUES
GO
-- Now, disable the constraint
ALTER TABLE favorite_book NOCHECK CONSTRAINT add_reprint_nbr
GO

```

See Also

[CREATE SCHEMA](#)
[DROP](#)

CREATE/ALTER TRIGGER Statement

A *trigger* is a special kind of stored procedure that fires automatically (hence the term “trigger”) when a specific data-modification statement is executed against a table. The trigger is directly associated with the table and is considered a dependent object. For example, you might want all of the **part_numbers** in the **sales** table to be updated when a **part_number** is changed in the **products** table, thus ensuring that part numbers are always in sync. You can accomplish this with a trigger.



ALTER TRIGGER is not an ANSI-supported statement.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
CREATE TRIGGER trigger_name
  {BEFORE | AFTER} {DELETE | INSERT | UPDATE [OF column[, ...]}}
  ON table_name
  [REFERENCING {OLD {[ROW] | TABLE} [AS] old_name | NEW
  {[ROW | TABLE] [AS] new_name} } [FOR EACH { ROW | STATEMENT } ]
  [WHEN (conditions)]
  [BEGIN ATOMIC] code_block
  [END]
```

Keywords

CREATE TRIGGER *trigger_name*

Creates a trigger named *trigger_name* and associates it with a specific table.

BEFORE | AFTER

Declares that the trigger logic is fired either *BEFORE* or *AFTER* the data-manipulation operation that invoked the trigger. *BEFORE* triggers perform their operations before the *INSERT*, *UPDATE*, or *DELETE* operation occurs, allowing you to do dramatic things like circumvent the data-manipulation operation altogether. *AFTER* triggers fire after the data-manipulation operation has completed and are useful for after-the-fact operations like recalculating running totals.

DELETE | INSERT | UPDATE [OF *column*[, ...]]

Defines the data-manipulation operation that causes the trigger to fire: *DELETE* statements, *INSERT* statements, or *UPDATE* statements. You may optionally choose which columns will trigger an update trigger using *UPDATE OF *column*[, ...]*. If an update occurs on any columns not in the column list, the trigger will not fire.

ON *table_name*

Declares the pre-existing table on which the trigger is dependent.

REFERENCING {OLD {[ROW] | TABLE} [AS] *old_name* | NEW {ROW | TABLE} [AS] *new_name*}

Enables aliasing for the old or new *ROW* or *TABLE* acted upon by the trigger. Although the syntax shows the options as exclusive, you may have up to four aliasing references: one for the old row, one for the old table, one for the new row, and one for the new table. The alias *OLD* refers to the data contained in the table or row before the data-manipulation operation that fired the trigger, while the alias *NEW* refers to the data that will be contained in the table or row after the data-manipulation operation that fired the trigger. Note that the syntax indicates

that *ROW* is optional, but *TABLE* is not. (That is, *OLD ROW AS* is the same as *OLD AS*, but for *TABLE*, the only valid option is *OLD TABLE AS*.) *INSERT* triggers do not have an *OLD* context, while *DELETE* triggers do not have a *NEW* context. The keyword *AS* is noise and may be omitted. If the *REFERENCING* clause specifies either *OLD ROW* or *NEW ROW*, the *FOR EACH ROW* clause is required.

FOR EACH { ROW | STATEMENT }

Tells the database to apply the trigger for each row in the table that has changed (*ROW*) or for each SQL statement issued against the table (*STATEMENT*). Consider a single *UPDATE* statement that updates the salaries of 100 employees. If you specify *FOR EACH ROW*, the trigger will execute 100 times. If you specify *FOR EACH STATEMENT*, the trigger will execute only once.

WHEN (conditions)

Allows you to define additional criteria for a trigger. For example, you might have a trigger called *DELETE employee* that will fire whenever an employee is deleted. When a trigger fires, if the search conditions contained in the *WHEN* clause evaluate to *TRUE*, the trigger action will fire. Otherwise, the trigger will not fire.

BEGIN ATOMIC | code_block | END

The ANSI standard requires that the *code_block* should contain only one SQL statement or, if it contains multiple SQL statements, they should be enclosed in a *BEGIN* and *END* block.

Rules at a Glance

Triggers, by default, fire once at the *statement level*. That is, a single *INSERT* statement might insert 500 rows into a table, but an insert trigger on that table will fire only one time. However, some vendors allow a trigger to fire for each row of the data-modification operation. A statement that inserts 500 rows into a table that has a row-level insert trigger will cause that trigger to fire 500 times, once for each inserted row.

In addition to being associated with a specific data-modification statement (*INSERT*, *UPDATE*, or *DELETE*) on a given table, triggers are associated with a specific *time of firing*. In general, triggers can fire *BEFORE* the data-modification statement is processed, *AFTER* it is processed, or (when supported by the vendor) *INSTEAD OF* the statement being processed. Triggers that fire before or instead of the data-modification statement do not see the changes that the statement renders, while those that fire afterward can see and act upon the changes rendered.

Triggers make use of two *pseudotables*. (They are pseudotables in the sense that they are not declared with a *CREATE TABLE* statement, but they exist logically on the database.) The pseudotables have different names on the different platforms, but we'll call them **Before** and **After** here. They are structured exactly the same as the table a trigger is associated with, but they contain snapshots of the table's data: the **Before** table contains a snapshot of all the records in the table before the trigger fired, while the **After** table contains a snapshot of how all the records in the table will look after the event that fires the trigger has occurred. You can then use comparison operations to compare the data in the table before and after the event to determine exactly what you want to happen.

Here is an Oracle *BEFORE* trigger that uses the *OLD* and *NEW* pseudotables to compare values. (By way of comparison, SQL Server uses the *DELETED* and *INSERTED* pseudotables in the same way.) This trigger creates an audit record before changing an employee's pay record:

```

CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
WHEN (new.emp_salary <> old.emp_salary)
BEGIN
    INSERT INTO employee_audit
    VALUES ('old', :old.emp_id, :old.emp_salary, :old.emp_ssn);
END;

```

You can also take advantage of capabilities within each database platform to make your triggers more powerful and easier to program. For example, Oracle has a special *IF...THEN* clause for use just in triggers. This *IF...THEN* clause takes the form *IF {DELETING | INSERTING | UPDATING} THEN*. The following example builds an Oracle *DELETE* and *UPDATE* trigger that uses the *IF DELETING THEN* clause:

```

CREATE TRIGGER if_emp_changes
BEFORE DELETE OR UPDATE ON employee
FOR EACH ROW
BEGIN
    IF DELETING THEN
        INSERT INTO employee_audit
        VALUES ('DELETED', :old.emp_id, :old.emp_salary, :old.emp_ssn);
    ELSE
        INSERT INTO employee_audit
        VALUES ('UPDATED', :old.emp_id, :new.emp_salary, :old.emp_ssn);
    END IF;
END;

```

This SQL Server example adds a new table called **contractor** to the database, and all records in the **employee** table that indicate that the employee is a contractor are moved into the **contractor** table. Now we'll specify that all new employees inserted into the **employee** table will go into the **contractor** table instead, using an *INSTEAD OF INSERT* trigger:

```

CREATE TRIGGER if_emp_is_contractor
INSTEAD OF INSERT ON employee
BEGIN
    INSERT INTO contractor
    SELECT * FROM inserted WHERE status = 'CON'
    INSERT INTO employee
    SELECT * FROM inserted WHERE status = 'FTE'
END
GO

```

Adding triggers to a table that already has data in it does not cause the triggers to fire. A trigger will fire only for data-modification statements declared in the trigger definition that occur after the trigger is created.

Once in place, a trigger generally ignores structural changes to tables, such as an added column or an altered datatype on an existing column, unless the modification directly interferes with the operation of the trigger. For example, if you add a new column to a table, existing triggers will ignore the new column (with the exception of *UPDATE* triggers). On the other hand, removing from a table a column that is used by a trigger will cause that trigger to fail every time it executes.

Programming Tips and Gotchas

One of the key programming issues associated with triggers is the inappropriate and uncontrolled use of nested and recursive triggers. A *nested trigger* is a trigger that invokes a data-manipulation operation that causes other triggers to fire. For example, assume we have three tables, **T1**, **T2**, and **T3**. Table **T1** has a *BEFORE INSERT* trigger that inserts a record into table **T2**. Table **T2** also has a *BEFORE INSERT* trigger that inserts a record into table **T3**. Although this is not necessarily a bad thing if your logic is well considered and fully thought out, it does introduce two problems. First, an *INSERT* into table **T1** now requires many more I/O operations and transactions than a simple *INSERT* statement typically does. Second, you can get yourself into hot water if table **T3** performs an *INSERT* operation against table **T1**. In a case like that, you may have a looping trigger process that can consume all available disk space and even shut down the server.

Recursive triggers are triggers that can fire themselves; for example, an *INSERT* trigger that performs an *INSERT* against its own base table. If the procedural logic within the *code_body* is not properly constructed, adding a recursive trigger can cause a looping trigger error. In recognition of this danger, using recursive triggers often requires setting a special configuration flag on the various database platforms.

MySQL

MySQL's implementation of the *CREATE TRIGGER* statement follows:

```
CREATE [DEFINER = {user_name | CURRENT_USER}]
  TRIGGER trigger_name {BEFORE | AFTER} {INSERT | UPDATE | DELETE}
    ON table_name FOR EACH ROW code_body
```

where:

DEFINER = {*user_name* | *CURRENT_USER*}

Specifies the user account to use when checking privileges. You may specify either a pre-existing user or the user who issued the *CREATE TRIGGER* statement (i.e., the *CURRENT_USER*). *CURRENT_USER* is the default when this clause is omitted.

MySQL does not allow triggers on temporary tables. Insert triggers will fire any time data is inserted into a table, not just on *INSERT* statements. Thus, insert triggers on a table will also fire when *LOAD DATA* and *REPLACE* statements are executed. Similarly, a delete trigger will also fire on a *REPLACE* statement. Triggers are not, however, activated by cascading foreign key actions.

MySQL allows only one type of trigger per table. Thus, a table could have a *BEFORE INSERT* and an *AFTER INSERT* trigger, but not two *AFTER INSERT* triggers. To fire multiple SQL statements for each row that the trigger fires upon, be sure to use a *BEGIN...END* block inside of the *code_body*.

MySQL doesn't yet support the *ALTER TRIGGER* statement.

Oracle

Oracle supports the ANSI standard for *CREATE TRIGGER*, with several additions and variations:

```
CREATE [OR REPLACE] TRIGGER trigger_name
  {BEFORE | AFTER | INSTEAD OF}
  { {[object_event]} [database_event] [...] ON {DATABASE | schema.SCHEMA} } |
```

```

{[DELETE] [OR] [INSERT] [OR] [UPDATE [OF column[, ...]]] [...]
  ON {table_name | [NESTED TABLE column_name OF] view_name}
  [REFERENCING {[OLD [AS] old_name] [NEW [AS] new_name]
    [PARENT [AS] parent_name]}}
  [FOR EACH ROW] }
  [FOLLOWS trigger_name]
  [{ENABLE | DISABLE}]
  [WHEN (conditions)] code_block

```

Here is the syntax for *ALTER TRIGGER*, which allows you to rename, enable, or disable a trigger without dropping and recreating it:

```

ALTER TRIGGER trigger_name
{ {ENABLE | DISABLE} | RENAME TO new_name |
  COMPILE [compiler_directives] [DEBUG] [REUSE SETTINGS] }

```

The parameters are:

OR REPLACE

Recreates an existing trigger by assigning a new definition to an existing trigger named *trigger_name*.

object_event

In addition to the standard data-modification events, Oracle allows triggers to fire based on object events. *object_event* operations may be paired with *BEFORE* and *AFTER* keywords. An *object_event* fires the trigger whenever such an event occurs, according to the following keywords:

ALTER

Fires whenever an *ALTER* statement (except *ALTER DATABASE*) is issued.

ANALYZE

Fires whenever Oracle validates the structure of a database object, or collects or deletes statistics on an index.

ASSOCIATE STATISTICS

Fires whenever Oracle associates a statistics type with a database object.

AUDIT

Fires whenever Oracle tracks a SQL statement or operation against a schema object.

COMMENT

Fires whenever an Oracle comment is added in the data dictionary to a database object.

DDL

Fires whenever Oracle encounters any *object_event* in this list.

DISASSOCIATE STATISTICS

Fires whenever Oracle disassociates a statistics type from a database object.

DROP

Fires whenever a *DROP* statement erases a database object from the data dictionary.

GRANT

Fires whenever a user grants privileges or roles to another user or role.

NOAUDIT

Fires whenever the *NOAUDIT* statement causes Oracle to stop tracking SQL statements or operations against schema objects.

RENAME

Fires whenever the *RENAME* statement changes the name of a database object.

REVOKE

Fires whenever a user revokes privileges or roles from another user or role.

TRUNCATE

Fires whenever a *TRUNCATE* statement is issued against a table or cluster.

database_event

In addition to the standard data-modification events, Oracle allows triggers to fire based on database events. *database_event* operations may be paired with *BEFORE* and *AFTER* keywords. The list of allowable *database_event* keywords is:

LOGON

Fires whenever a client application logs on to the database. Valid for *AFTER* triggers only.

LOGOFF

Fires whenever a client application logs off of the database. Valid for *BEFORE* triggers only.

SERVERERROR

Fires whenever a server error message is logged. Valid for *AFTER* triggers only.

SHUTDOWN

Fires whenever an instance of the database is shut down. Valid only for *BEFORE* triggers with the *ON DATABASE* clause.

STARTUP

Fires whenever an instance of the database is opened. Valid only for *AFTER* triggers with the *ON DATABASE* clause.

SUSPEND

Fires whenever a server error causes a transaction to suspend. Valid for *AFTER* triggers only.

ON {DATABASE | *schema.SCHEMA*}

Declares that the trigger fires whenever any database user invokes a triggering event with *ON DATABASE*. The trigger then fires for events occurring anywhere in the entire database. Otherwise, *ON schema.SCHEMA* declares that the trigger fires whenever a user connected as *schema* invokes a triggering event. The trigger then fires for events occurring anywhere in the current schema.

ON [NESTED TABLE *column_name* OF] *view_name*

Declares that the trigger fires only if the data-manipulation operation applies to the column(s) of the view called *view_name*. The *ON NESTED TABLE* clause is compatible only with *INSTEAD OF* triggers.

REFERENCING PARENT [AS] *parent_name*

Defines the alias for the current row of the parent table (i.e., supertable). Otherwise, identical to the ANSI standard.

FOLLOWS *trigger_name*

Specifies that the new trigger is of the same type as another trigger and that it should fire only after the other trigger has fired. The *trigger_name* must already exist. Rather than creating a series of triggers that must fire in a specific order, it is recommended that you instead create a single trigger with logic to handle all of the situations that the multiple triggers handled.

ENABLE

Enables a deactivated trigger when used with *ALTER TRIGGER*, or creates a new trigger in enabled mode (the default). You may alternately use the statement *ALTER TABLE table_name ENABLE ALL TRIGGERS*.

DISABLE

Disables an activated trigger when used with *ALTER TRIGGER*, or creates a new trigger in disabled mode. You may alternately use the statement *ALTER TABLE table_name DISABLE ALL TRIGGERS*.

RENAME TO *new_name*

Renames the trigger to *new_name*, though the state of the trigger remains unchanged when used with *ALTER TRIGGER*.

COMPILE [DEBUG] [REUSE SETTINGS]

Compiles a trigger, whether valid or invalid, and all the objects on which the trigger depends. If any of the objects are invalid, the trigger is invalid. If all of the objects are valid, including the *code_body* of the trigger, the trigger is valid.

DEBUG

Tells the PL/SQL compiler to generate and store extra information for use by the PL/SQL debugger.

REUSE SETTINGS

Tells Oracle to retain all compiler switch settings, which can save significant time during the compile process.

compiler_directives

Defines a special value for the PL/SQL compiler in the format: *directive = 'value'*. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The *directive* is valid only for the unit being compiled.

When referencing values in the *OLD* and *NEW* pseudotables, the values must be prefaced with a colon (:), except in the trigger's *WHEN* clause, where no colons are used. In this example, we'll call a procedure in the *code_body* and use both *:OLD* and *:NEW* values as arguments:

```
CREATE TRIGGER scott.sales_check
BEFORE INSERT OR UPDATE OF ord_id, qty ON scott.sales
FOR EACH ROW
WHEN (new.qty > 10)
CALL check_inventory(:new.ord_id, :new.qty, :old.qty);
```

Multiple trigger types may be combined into a single trigger command *if* they are of the same level (row or statement) and they are on the same table. When triggers are combined in a single statement, the clauses *IF INSERTING THEN*, *IF UPDATING THEN*, and *IF DELETING THEN* may be used in the PL/SQL block to break the code logic into distinct segments. An *ELSE* clause also can be used in this structure.

Following is an example of a *database_event*-style trigger:

```
CREATE TRIGGER track_errors
AFTER SERVERERROR ON DATABASE
BEGIN
IF (IS_SERVERERROR (04030))
THEN INSERT INTO errors ('Memory error');
```

```

    ELSE (IS_SERVERERROR (01403))
        THEN INSERT INTO errors ('Data not found');
    END IF;
END;

```

This example creates a trigger that is *SCHEMA*-wide in scope:

```

CREATE OR REPLACE TRIGGER create_trigger
AFTER CREATE ON scott.SCHEMA
BEGIN
    RAISE_APPLICATION_ERROR (num => -20000, msg =>
        'Scott created an object');
END;

```

DDL triggers

Oracle now has support for DDL triggers, meaning that you can have a trigger fire, for example, when a new table is created or when a view is dropped. DDL triggers may be created on either a *DATABASE* or a *SCHEMA* for events like *CREATE*, *ALTER*, or *DROP*. For example:

```

CREATE TRIGGER audit_object_changes AFTER CREATE ON SCHEMA
    code_body;

```

The full list of DDL trigger events includes the firing of any of these statements: *ALTER*, *ANALYZE*, *ASSOCIATE STATISTICS*, *AUDIT*, *COMMENT*, *CREATE*, *DISASSOCIATE STATISTICS*, *DROP*, *GRANT*, *NOAUDIT*, *RENAME*, *REVOKE*, *TRUNCATE*, and *DDL* (which will fire the trigger when any of the preceding DDL statements are issued). You may also create a DDL trigger that fires on a specific database state, rather than on a DDL statement. The database states you may use include *AFTER STARTUP*, *BEFORE SHUTDOWN*, *AFTER DB_ROLE_CHANGE*, *AFTER LOGON*, *BEFORE LOGOFF*, *AFTER SERVERERROR*, and *AFTER SUSPEND*.

PostgreSQL

PostgreSQL's implementation of *CREATE TRIGGER* offers a subset of the features found in the ANSI standard. On PostgreSQL, a trigger may fire *BEFORE* a data-modification operation is attempted on a record and before any constraints are fired, or it may fire *AFTER* a data-manipulation operation fires (and after constraints have been checked), making all operations involved in the transaction visible to the trigger. Finally, the trigger may fire *INSTEAD OF* the data-modification operation and completely replace the *INSERT*, *UPDATE*, or *DELETE* statement with some other behavior. The *CREATE TRIGGER* syntax is:

```

CREATE TRIGGER trigger_name
{ BEFORE | AFTER }
{ {[DELETE] [OR | ,] [INSERT] [OR | ,] [UPDATE]} [OR ...] }
ON table_name
FOR EACH { ROW | STATEMENT }
EXECUTE PROCEDURE function_name (parameters)

```

PostgreSQL's implementation of *ALTER TRIGGER* merely allows you to rename an existing trigger:

```
ALTER TRIGGER trigger_name ON table_name RENAME TO new_trigger_name
```

The *CREATE TRIGGER* parameters are:

OR

Declares an additional trigger action. The **OR** keyword is a synonym for the comma delimiter.

FOR EACH ROW

Explicitly declares that the trigger acts at the row level. This is also the default. Although PostgreSQL does not reject the **STATEMENT** keyword, it does not enforce statement-level triggers.

EXECUTE PROCEDURE *function_name (parameters)*

Executes a previously defined function (created using **CREATE FUNCTION**) rather than a block of procedural code. (PostgreSQL does not have its own procedural language.)

The following is an example of a PostgreSQL **BEFORE** trigger that checks at the row level to ensure that the specified distributor code exists in the **distributors** table before inserting or updating a row in the **sales** table:

```
CREATE TRIGGER if_dist_exists
BEFORE INSERT OR UPDATE ON sales
FOR EACH ROW
EXECUTE PROCEDURE check_primary_key
  ('dist_id', 'distributors', 'dist_id');
```

BEFORE and **AFTER** triggers are supported by the ANSI standard. **INSTEAD OF** triggers on PostgreSQL completely skip the data-modification operation that triggered them in favor of code that you substitute for the data-modification operation.

SQL Server

SQL Server supports the basics of the ANSI standard with the addition of the **INSTEAD OF** trigger type and a column change check. It does not support the **REFERENCING** or **WHEN** clauses. Its syntax is:

```
{CREATE | ALTER} TRIGGER trigger_name ON table_name
  [WITH [ENCRYPTION] [EXECUTE] AS {CALLER | SELF | 'user_name'}]
  {FOR | AFTER | INSTEAD OF}
  { dml_events | ddl_events }
  [WITH APPEND]
  [NOT FOR REPLICATION]
  AS
    [IF UPDATE(column) [{AND | OR} UPDATE(column)][...]]
    code_block
```

where:

{**CREATE** | **ALTER**} **TRIGGER** *trigger_name*

Creates a new trigger named *trigger_name* or alters an existing trigger of that name by adding or changing trigger properties or the trigger *code_block*. When altering an existing trigger, the permissions and dependencies of the existing trigger are maintained.

ON *table_name*

Declares the table or view on which the trigger is dependent. Views may have **INSTEAD OF** triggers defined on them, as long as they are updatable and do not have the **WITH CHECK** clause on them.

WITH ENCRYPTION

Encrypts the text of the *CREATE TRIGGER* statement as shown in the **syscomments** table. This option is useful to protect important intellectual property. *WITH ENCRYPTION* prevents the trigger from being used in a SQL Server replication scheme.

EXEC[UTE] AS {CALLER | SELF | OWNER | 'user_name'}

Specifies the privileges under which the trigger will execute. *CALLER* indicates the routine will run with the privileges of the user invoking the routine. When this clause is omitted, SQL Server assumes *CALLER*. *SELF* indicates the routine will run with the privileges of the creator of the routine. *OWNER* indicates the routine will run with the privileges of the current owner of the routine. *'user_name'* indicates the routine will run with the privileges of the named, pre-existing user.

FOR | AFTER | INSTEAD OF

Tells SQL Server when the trigger should fire. *FOR* and *AFTER* are synonyms and specify that the trigger should fire only after the triggering data-modification statement (and any cascading actions and constraint checks) have completed successfully. The *INSTEAD OF* trigger is similar to the ANSI *BEFORE* trigger in that the code of the trigger may completely replace the data-modification operation. It specifies that the trigger be executed instead of the data-modification statement that fired the trigger. *INSTEAD OF DELETE* triggers cannot be used when there is a cascading action on the delete. Only *INSTEAD OF* triggers can access *TEXT*, *NTEXT*, or *IMAGE* columns.

dml_events

Specifies an ANSI-standard DML event that causes the trigger to fire: *DELETE*, *INSERT*, and/or *UPDATE*. You may specify one or more DML events in a single statement.

ddl_events

Specifies a DDL event that causes the trigger to fire: *CREATE*, *ALTER*, *DROP*, *GRANT*, *DENY*, *REVOKE*, *BIND*, *UNBIND*, *RENAME*, or *UPDATE STATISTICS*. DDL events may fire *FOR* or *AFTER* the event. You may also specify a trigger that fires *FOR* or *AFTER LOGON*. SQL Server also has a number of shortcuts called “*DDL Event Groups*” that include many DDL events with one term: for example, *DDL_TABLE_EVENTS* includes the *CREATE*, *ALTER*, and *DROP TABLE* DDL events. Refer to the SQL Server documentation for a full listing of DDL Event Groups.

WITH APPEND

Adds an additional trigger of an existing type to a table or view. This clause is supported for backward compatibility with earlier versions of the product and can be used only with *FOR* triggers. This clause cannot be used with *INSTEAD OF* triggers or if *AFTER* is explicitly stated during creation, nor can it be used with CLR triggers.

NOT FOR REPLICATION

Prevents data-manipulation operations invoked through SQL Server’s built-in replication engine from firing the trigger.

IF UPDATE(column) [{AND | OR} UPDATE(column)] [...]

Allows you to choose the specific columns that fire the trigger. A column-specific trigger fires only on *UPDATE* and *INSERT* operations, not on *DELETE* operations. If an *UPDATE* or *INSERT* occurs on any columns not in the *column* list, the trigger will not fire.

SQL Server allows multiple triggers for a given data-manipulation operation on a table or view. Thus, three *UPDATE* triggers are possible on a single table, and multiple *AFTER* triggers are possible on a given table. Their specific order is undefined, though the first and last triggers can be explicitly declared using the **sp_settriggerorder** system stored procedure. Only one *INSTEAD OF* trigger is possible per *INSERT*, *UPDATE*, or *DELETE* statement on a given table.

In SQL Server, any combination of triggers is possible in a single trigger definition statement; simply separate each option with a comma. (When you do so, the same code fires for each statement in the combination definition.)

SQL Server implicitly fires in the *FOR EACH STATEMENT* style of the ANSI standard.

SQL Server instantiates two important pseudotables when a trigger is fired: **deleted** and **inserted**. They are equivalent, respectively, to the **before** and **after** pseudotables described earlier, in the SQL2003 section. These tables are identical in structure to the table on which the triggers are defined, except that they contain the old data before the data-modification statement fired (**deleted**) and the new values of the table after the data-modification statement has fired (**inserted**).

The *AS IF UPDATE(column)* clause tests specifically for *INSERT* or *UPDATE* actions on a given column or columns, in the way the ANSI statement uses the *UPDATE(column)* syntax. Specify multiple columns by adding separate *UPDATE(column)* clauses after the first. Follow the *AS IF UPDATE(column)* clause with a Transact-SQL *BEGIN...END* block to allow the trigger to fire multiple Transact-SQL operations. This clause is functionally equivalent to the *IF...THEN...ELSE* operation.

In addition to intercepting data-modification statements as shown in the ANSI SQL example, SQL Server allows you to perform other sorts of actions when a data-modification operation occurs. In the following example, we've decided that the table **sales_archive_2002** is off-limits and that anyone who attempts to alter data in this table is to be notified of that restriction:

```
CREATE TRIGGER archive_trigger
ON sales_archive_2002
FOR INSERT, UPDATE
AS RAISERROR (50009, 16, 10, 'No changes allowed to this table')
GO
```

SQL Server *does not* allow the following statements within the Transact-SQL *code block* of a trigger: *ALTER*, *CREATE*, *DROP*, *DENY*, *GRANT*, *REVOKE*, *LOAD*, *RESTORE*, *RECONFIGURE*, or *TRUNCATE*. In addition, it does not allow any *DISK* statements or the *UPDATE STATISTICS* command.

SQL Server allows triggers to fire recursively using the *recursive triggers* setting of the **sp_dboption** system stored procedure. Recursive triggers, by their own action, cause themselves to fire again. For example, if an *INSERT* trigger on table **T1** performs an *INSERT* operation on table **T1**, it might perform a recursive operation. Since recursive triggers can be dangerous, this functionality is disabled by default.

Similarly, SQL Server allows *nested triggers* up to 32 levels deep. If any one of the nested triggers performs a *ROLLBACK* operation, no further triggers execute. An example of nested triggers is a trigger on table **T1** firing an operation against table **T2**, which also has a trigger that fires an operation against table **T3**. The triggers cancel if an infinite loop is encountered. Nested triggers are enabled with the nested triggers

setting of the system stored procedure `sp_configure`. If nested triggers are disabled, recursive triggers are disabled as well, regardless of what the recursive triggers setting is in `sp_dboption`.

In the following example, we want to reroute the user activity that occurs on the `people` table—especially *UPDATE* transactions—so that changes to records in the `people` table are instead written to the `people_reroute` table. Our update trigger will record any changes to columns 2, 3, or 4 of the `people` table and write them to the `people_reroute` table. The trigger will also record which user issued the *UPDATE* statement and the time of the transaction:

```

CREATE TABLE people
  (people_id      CHAR(4),
  people_name    VARCHAR(40),
  people_addr   VARCHAR(40),
  city           VARCHAR(20),
  state          CHAR(2),
  zip            CHAR(5),
  phone          CHAR(12),
  sales_rep     empid NOT NULL)
GO
CREATE TABLE people_reroute
  (reroute_log_id      UNIQUEIDENTIFIER DEFAULT NEWID(),
  reroute_log_type    CHAR (3) NOT NULL,
  reroute_people_id   CHAR(4),
  reroute_people_name VARCHAR(40),
  reroute_people_addr VARCHAR(40),
  reroute_city        VARCHAR(20),
  reroute_state       CHAR(2),
  reroute_zip         CHAR(5),
  reroute_phone       CHAR(12),
  reroute_sales_rep  empidNOT NULL,
  reroute_user sysname DEFAULT SUSER_SNAME(),
  reroute_changed datetime DEFAULT GETDATE())
GO
CREATE TRIGGER update_person_data
ON people
FOR update AS
IF (COLUMNS_UPDATED(people_name)
    OR COLUMNS_UPDATED(people_addr)
    OR COLUMNS_UPDATED(city) )
BEGIN
  -- Audit OLD record
  INSERT INTO people_reroute (reroute_log_type, reroute_people_id,
    reroute_people_name, reroute_people_addr, reroute_city)
  SELECT 'old', d.people_id, d.people_name, d.people_addr, d.city
  FROM deleted AS d
  -- Audit NEW record
  INSERT INTO people_reroute (reroute_log_type, reroute_people_id,
    reroute_people_name, reroute_people_addr, reroute_city)
  SELECT 'new', n.people_id, n.people_name, n.people_addr, n.city
  FROM inserted AS n
END
GO

```

Note that SQL Server *CREATE* statements allow *deferred name resolution*, meaning that commands will be processed even if they refer to a database object that does not yet exist in the database.

SQL Server supports the creation of triggers written in Microsoft .NET Framework common language runtime (CLR) methods that can take and return user-supplied parameters. These routines have similar *CREATE* and *ALTER* declarations to regular SQL triggers, but the code bodies are external assemblies. Refer to the SQL Server documentation if you want to learn more about programming routines using the CLR.

See Also

CREATE/ALTER FUNCTION/PROCEDURE
DELETE
DROP
INSERT
UPDATE

CREATE/ALTER TYPE Statement

The *CREATE TYPE* statement allows you to create a *user-defined type* (UDT); that is, a user-defined datatype or “class” in object-oriented terms. UDTs extend SQL capabilities into the realm of object-oriented programming by allowing inheritance and other object-oriented features. You can also create something called *typed tables* with the *CREATE TABLE* statement using a previously created type made with the *CREATE TYPE* statement. Typed tables are based on UDTs and are equivalent to “instantiated classes” from object-oriented programming.

Platform	Command
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
CREATE TYPE type_name
[UNDER supertype_name]
[AS [new_udt_name] datatype [attribute][, ...]
{[REFERENCES ARE [NOT] CHECKED
[ON DELETE
{NO ACTION | CASCADE | RESTRICT | SET NULL | SET DEFAULT}]] | |
[DEFAULT value] |
[COLLATE collation_name]}]
[[NOT] INSTANTIABLE]
[[NOT] FINAL]
[REF IS SYSTEM GENERATED | |
REF USING datatype
[CAST {(SOURCE AS REF) | (REF AS SOURCE)} WITH identifier] | |
REF new_udt_name[, ...] ]
[CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)} WITH identifier]
[method_definition[, ...]]
```

The following syntax alters an existing user-defined datatype:

```
ALTER TYPE type_name {ADD ATTRIBUTE type_definition |  
DROP ATTRIBUTE type_name}
```

Keywords

{CREATE | ALTER} TYPE *type_name*

Creates a new type or alters an existing type with the name *type_name*.

UNDER *supertype_name*

Creates a subtype that is dependent upon a single, pre-existing, named supertype. (A UDT can be a supertype if it is defined as *NOT FINAL*.)

AS [*new_udt_name*] *datatype* [*attribute*][, ...]

Defines attributes of the type as if they were column declarations in a *CREATE TABLE* statement without constraints. You should either define the UDT attribute on an existing datatype, such as *VARCHAR(10)*, or on another, previously created UDT, or even on a user-defined domain. Defining a UDT using a predefined datatype (e.g., *CREATE TYPE my_type AS INT*) creates a *distinct type*, while a UDT defined with an attribute definition is a *structured type*. The allowable attributes for a structured type are:

ON DELETE NO ACTION

Produces an error when the foreign key is violated (the default).

ON DELETE RESTRICT

Synonym for NO ACTION.

ON DELETE CASCADE

Sets the value of the referencing column to the value of the referenced column.

ON DELETE SET NULL

Sets the value of the referencing column to NULL.

ON DELETE SET DEFAULT *value*

Defines a default value for the UDT for when the user does not supply a value. Follows the rules of a *DEFAULT* in the section.

COLLATE *collation_name*

Assigns a collation—that is, a sort order—for the UDT. When omitted, the collation of the database where the UDT was created applies. Follows the rules of a *COLLATION* in the section.

[NOT] INSTANTIABLE

Defines the UDT such that it can be instantiated. *INSTANTIABLE* is required for typed tables, but not for standard UDTs.

[NOT] FINAL

Required for all UDTs. *FINAL* means the UDT may have no subtypes. *NOT FINAL* means the UDT may have subtypes.

REF

Defines either a system-generated or user-generated reference specification—that is, a sort of unique identifier that acts as a pointer that another type may reference. By referencing a pre-existing type using its reference specification, you can have a new type inherit properties of a pre-existing type. There are three ways to tell the DBMS how the typed table's reference column gets its values (i.e., its reference specification):

new_udt_name[, ...]

Declares that the reference specification is provided by another pre-existing UDT called *new_udt_name*.

IS SYSTEM GENERATED

Declares that the reference specification is system-generated (think of an automatically incrementing column). This is the default when the *REF* clause is omitted.

USING datatype [CAST {(SOURCE AS REF) | (REF AS SOURCE)} WITH identifier]

Declares that the user defines the reference specification. You do this by using a predefined datatype and optionally casting the value. You can use *CAST (SOURCE AS REF) WITH identifier* to cast the value with the specified *datatype* to the reference type of the structured type, or use *CAST (REF AS SOURCE) WITH identifier* to cast the value for the structured type to the *datatype*. The *WITH* clause allows you to declare an additional identifier for the cast datatype.

CAST {(SOURCE AS DISTINCT) | (DISTINCT AS SOURCE)} WITH identifier method_definition[, ...]

Defines one or more pre-existing methods for the UDT. A *method* is merely a specialized user-defined function and is created using the *CREATE METHOD* statement (see *CREATE FUNCTION*). The *method_definition* clause is not needed for structured types since their method(s) are implicitly created. The default characteristics of a method are *LANGUAGE SQL*, *PARAMETER STYLE SQL*, *NOT DETERMINISTIC*, *CONTAINS SQL*, and *RETURN NULL ON NULL INPUT*.

ADD ATTRIBUTE type_definition

Adds an additional attribute to an existing UDT, using the format described earlier under the *AS* clause. Available via the *ALTER TYPE* statement.

DROP ATTRIBUTE type_name

Drops an attribute from an existing UDT. Available via the *ALTER TYPE* statement.

Rules at a Glance

You can create user-defined types as a way to further ensure data integrity in your database and to ease the work involved in doing so. An important concept of UDTs is that they allow you to easily create *subtypes*, which are UDTs built upon other UDTs. The UDT that subtypes depend on is called a parent type or *supertype*. Subtypes inherit the characteristics of their supertypes.

Assume, for example, that you want to define a general UDT for phone numbers called *phone_nbr*. You could then easily build new subtypes of *phone_nbr* called *home_phone*, *work_phone*, *cell_phone*, *pager_phone*, etc. Each of the subtypes could inherit the general characteristics of the parent type but also have characteristics of its own.

In this example, we create a general root UDT called *money* and then several subtypes:

```
CREATE TYPE money (phone_number DECIMAL (10,2) )
    NOT FINAL;
CREATE TYPE dollar UNDER money AS DECIMAL(10,2)
    (conversion_rate DECIMAL(10,2) ) NOT FINAL;
CREATE TYPE euro    UNDER money AS DECIMAL(10,2)
    (dollar_conversion_rate DECIMAL(10,2) ) NOT FINAL;
CREATE TYPE pound   UNDER euro
    (euro_conversion_rate DECIMAL(10,2) ) FINAL;
```

Programming Tips and Gotchas

The biggest programming gotcha for user-defined types is that they are seldom used and not well understood by most database developers and database administrators. Consequently, they can be problematic due to simple ignorance. They offer, however, a consistent and labor-saving approach for representing commonly reused conceptual elements in a database, such as an address (e.g., *street1*, *street2*, *city*, *state*, *postal code*).

MySQL

Not supported.

Oracle

Oracle has *CREATE TYPE* and *ALTER TYPE* statements, but they are not ANSI-standard. Instead of a single *CREATE TYPE* statement, Oracle uses *CREATE TYPE BODY* to define the code that makes up the UDT, while *CREATE TYPE* defines the argument specification for the type. The syntax for *CREATE TYPE* is:

```
CREATE [OR REPLACE] TYPE type_name
{ [OID 'object_identifier'] [AUTHID {DEFINER | CURRENT_USER}]
  { AS | IS } OBJECT | [UNDER supertype_name] |
  {OBJECT | TABLE OF datatype | VARRAY ( limit ) OF datatype} }
  EXTERNAL NAME java_ext_name LANGUAGE JAVA USING data_definition
  { [ (attribute datatype[, ...]) [EXTERNAL NAME 'name'] |
    [[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]
    [{ {MEMBER | STATIC}
      {function_based | procedure_based} | constructor_clause |
      map_clause } [...] ]
    [pragma_clause] ] } }
```

Once the type has been declared, you encapsulate all of the UDT logic in the type body declaration. The *type_name* for both *CREATE TYPE* and *CREATE TYPE BODY* should be identical. The syntax for *CREATE TYPE BODY* is shown here:

```
CREATE [OR REPLACE] TYPE BODY type_name
{AS | IS}
{{MEMBER | STATIC}
{function_based | procedure_based | constructor_clause}}{...}
[map_clause]
```

Oracle's implementation of *ALTER TYPE* enables you to drop or add attributes and methods from or to the type:

```
ALTER TYPE type_name
{COMPILE [DEBUG] [{SPECIFICATION | BODY}]
  [compiler_directives] [REUSE SETTINGS] |
  REPLACE [AUTHID {DEFINER | CURRENT_USER}] AS OBJECT
  (attribute datatype[, ...] [element_definition[, ...]]) |
  [[NOT] OVERRIDING] [[NOT] FINAL] [[NOT] INSTANTIABLE]
  { {ADD | DROP} { {MAP | ORDER} MEMBER FUNCTION function_name
    (parameter datatype[, ...]) } |
    { {MEMBER | STATIC} {function_based | procedure_based} |
      constructor_clause | map_clause [pragma_clause] } [...] |
    {ADD | DROP | MODIFY} ATTRIBUTE (attribute [datatype][, ...]) |
    MODIFY {LIMIT int | ELEMENT TYPE datatype} |
    {INVALIDATE |
    CASCADE [{ [NOT] INCLUDING TABLE DATA | CONVERT TO SUBSTITUTABLE }] |
    [FORCE] [EXCEPTIONS INTO table_name]} ]}
```

Parameters for the three statements are as follows:

OR REPLACE

Recreates the UDT if it already exists. Objects that depend on the type are marked as *DISABLED* after you recreate the type.

AUTHID {DEFINER | CURRENT_USER}

Determines what user permissions any member functions or procedures are executed under and how external name references are resolved. (Note that subtypes inherit the permission styles of their supertypes.) This clause can be used only with an *OBJECT* type, not with a *VARRAY* type or a nested table type.

DEFINER

Executes functions or procedures under the privileges of the user who created the UDT. Also specifies that unqualified object names (object names without a schema definition) in SQL statements are resolved to the schema where the member functions or procedures reside.

CURRENT_USER

Executes functions or procedures under the privileges of the user who invoked the UDT. Also specifies that unqualified object names in SQL statements are resolved to the schema of the user who invoked the UDT.

UNDER *supertype_name*

Declares that the UDT is a subtype of another, pre-existing UDT. The supertype UDT must be created with the *AS OBJECT* clause. A subtype will inherit the properties of the supertype, though you should override some of those or add new properties to differentiate it from the supertype.

OID '*object_identifier*'

Declares an equivalent identical object, of the name '*object_identifier*', in more than one database. This clause is most commonly used by those developing Oracle Data Cartridges and is seldom used in standard SQL statement development.

AS OBJECT

Creates the UDT as a root object type (the top-level object in a UDT hierarchy of objects).

AS TABLE OF *datatype*

Creates a named nested table type of a UDT called *datatype*. The datatype cannot be an *NCLOB*, but *CLOB* and *BLOB* are acceptable. If the datatype is an object type, the columns of the nested table must match the name and attributes of the object type.

AS VARRAY (*limit*) OF *datatype*

Creates the UDT as an ordered set of elements, all of the same datatype. The *limit* is an integer of zero or more. The type name must be a built-in datatype, a *REF*, or an object type. The *VARRAY* cannot contain *LOB* or *XMLType* datatypes. *VARRAY* may be substituted with *VARYING ARRAY*.

EXTERNAL NAME *java_ext_name* LANGUAGE JAVA USING *data_definition*

Maps a Java class to a SQL UDT by specifying the name of a public Java external class, *java_ext_name*. Once defined, all objects in a Java class must be Java objects. The *data_definition* may be *SQLData*, *CustomDatum*, or *OraData*, as defined in the “Oracle9i JDBC Developers Guide.” You can map many Java object types to the same class, but there are two restrictions. First, you should not map two or more subtypes of a common datatype to the same class. Second, subtypes must be mapped to an immediate subclass of the class to which their supertype is mapped.

datatype

Declares the attributes and datatypes used by the UDT. Oracle does not allow *ROWID*, *LONG*, *LONG ROW*, or *UROWID*. Nested tables and *VARARRAYS* do not allow attributes of **AnyType**, **AnyData**, or **AnyDataSet**.

EXTERNAL NAME 'name' [NOT] OVERRIDING

Declares that this method overrides a *MEMBER* method defined in the supertype (*OVERRIDING*) or not (*NOT OVERRIDING*, the default). This clause is valid only for *MEMBER* clauses.

MEMBER | STATIC

Describes the way in which subprograms are associated with the UDT as attributes. *MEMBER* has an implicit first argument referenced as *SELF*, as in *object_expression.method()*. *STATIC* has no implicit arguments, as in *type_name.method()*. The following bases are allowed to call subprograms:

function_based

Declares a subprogram that is function-based using the syntax:

```
FUNCTION function_name (parameter datatype[, ...])
  return_clause | java_object_clause
```

This clause allows you to define the PL/SQL function-based UDT body without resorting to the *CREATE TYPE BODY* statement. The *function_name* cannot be the name of any existing attribute, including those inherited from supertypes. The *return_clause* and *java_object_clause* are defined later in this list.

procedure_based

Declares a subprogram that is function-based using the syntax:

```
PROCEDURE procedure_name (parameter datatype[, ...])
  {AS | IS} LANGUAGE {java_call_spec | c_call_spec}
```

This clause allows you to define the PL/SQL procedure-based UDT body without resorting to the *CREATE TYPE BODY* statement. The *procedure_name* cannot be the name of any existing attribute, including those inherited from supertypes. Refer to the entries on *java_call_spec* and *c_call_spec* later in this list for details on those clauses.

constructor_clause

Declares one or more constructor specifications, using the following syntax:

```
[FINAL] [INSTANTIABLE] CONSTRUCTOR FUNCTION datatype
  [ ( [SELF IN OUT datatype,] parameter datatype[, ...] ) ]
  RETURN SELF AS RESULT
  [ {AS | IS} LANGUAGE {java_call_spec | c_call_spec} ]
```

A constructor specification is a function that returns an initialized instance of a UDT. Constructor specifications are always *FINAL*, *INSTANTIABLE*, and *SELF IN OUT*, so these keywords are not required. The *java_call_spec* and *c_call_spec* subclauses may be replaced with a PL/SQL code block in the *CREATE TYPE BODY* statement. (Refer to the entries on *java_call_spec* and *c_call_spec* later in this list for details.)

map_clause

Declares the mapping or ordering of a supertype, using the following syntax:

```
{MAP | ORDER} MEMBER function_based
```

MAP uses more efficient algorithms for object comparison and is best in situations where you're performing extensive sorting or hash joins. *MAP MEMBER* specifies the relative position of a given instance in the ordering of all instances of the UDT. *ORDER MEMBER* specifies the explicit position of an instance and references a *function_based* subprogram that returns a *NUMBER* datatype value. Refer to the entry on *function_based* earlier in this list for details.

return_clause

Declares the datatype return format of the SQL UDT using the syntax:

```
RETURN datatype [ {AS | IS} LANGUAGE {java_call_spec |  
          c_call_spec} ]
```

java_object_clause

Declares the return format of the Java UDT using the syntax:

```
RETURN {datatype | SELF AS RESULT} EXTERNAL [VARIABLE] NAME  
'java_name'
```

If you use the *EXTERNAL* clause, the value of the public Java method must be compatible with the SQL returned value.

pragma_clause

Declares a *pragma restriction* (that is, an Oracle precompiler directive) for the type using the syntax:

```
PRAGMA RESTRICT REFERENCES ( {DEFAULT | method_name},  
                           {RNDS | WNDS | RNPS | WNPS | TRUST}[, ...] )
```

This feature is deprecated and *should be avoided*. It is intended to control how UDTs read and write database tables and variables.

DEFAULT

Applies the pragma to all methods in the type that don't have another pragma in place.

method_name

Identifies the exact method to which to apply the pragma.

RNDS

Reads no database state—no database reads allowed.

WNDS

Writes no database state—no database writes allowed.

RNPS

Reads no package state—no package reads allowed.

WNPS

Writes no package state—no package writes allowed.

TRUST

States that the restrictions of the pragma are assumed but not enforced.

java_call_spec

Identifies the Java implementation of a method using the syntax *JAVA NAME 'string'*. This clause allows you to define the Java UDT body without resorting to the *CREATE TYPE BODY* statement.

c_call_spec

Declares a C language call specification using the syntax:

```
C [NAME name] LIBRARY lib_name [AGENT IN (argument)]  
[WITH CONTEXT] [PARAMETERS (parameter[, ...])]
```

This clause allows you to define the C UDT body without resorting to the *CREATE TYPE BODY* statement.

COMPILE

Compiles the object type specification and body. This is the default when neither a *SPECIFICATION* clause nor a *BODY* clause is defined.

DEBUG

Generates and stores additional codes for the PL/SQL debugger. Do not specify both *DEBUG* and the *compiler_directive PLSQL_DEBUG*.

SPECIFICATION | BODY

Indicates whether to recompile the *SPECIFICATION* of the object type (created by the *CREATE TYPE* statement) or the *BODY* (created by the *CREATE TYPE BODY* statement).

compiler_directives

Defines a special value for the PL/SQL compiler in the format *directive = 'value'*. The directives are: *PLSQL_OPTIMIZE_LEVEL*, *PLSQL_CODE_TYPE*, *PLSQL_DEBUG*, *PLSQL_WARNINGS*, and *NLS_LENGTH_SEMANTICS*. They may each specify a value once in the statement. The directive is valid only for the unit being compiled.

REUSE SETTINGS

Retains the original value for the *compiler_directives*.

REPLACE AS OBJECT

Adds new member subtypes to the specification. This clause is valid only for object types.

[NOT] OVERRIDING

Indicates that the method overrides a *MEMBER* method defined in the supertype. This clause is valid only with *MEMBER* methods and is required for methods that define (or redefine, using *ALTER*) a supertype method. *NOT OVERRIDING* is the default if this clause is omitted.

ADD

Adds a new *MEMBER* function, function- or procedure-based subprogram, or attribute to the UDT.

DROP

Drops an existing *MEMBER* function, function- or procedure-based subprogram, or attribute from the UDT.

MODIFY

Alters the properties of an existing attribute of the UDT.

MODIFY LIMIT *int*

Increases the number of elements in a *VARRAY* collection type up to *int*, as long as *int* is greater than the current number of elements in the *VARRAY*. Not valid for nested tables.

MODIFY ELEMENT TYPE *datatype*

Increases the precision, size, or length of a scalar datatype of a *VARRAY* or nested table. This clause is valid for any non-object collection type. If the collection is a *NUMBER*, you may increase its precision or scale. If the collection is a *RAW*, you may increase its maximum size. If the collection is a *VARCHAR2* or *NVARCHAR2*, you may increase its maximum length.

INVALIDATE

Invalidates all dependent objects without checks.

CASCADE

Cascades the change to all subtypes and tables. By default, the action will be rolled back if any errors are encountered in the dependent types or tables.

[NOT] INCLUDING TABLE DATA

Converts data stored in the UDT columns to the most recent version of the column's type (*INCLUDING TABLE DATA*, the default), or not (*NOT INCLUDING TABLE DATA*). When *NOT*, Oracle checks the metadata but does not check or update the dependent table data.

CONVERT TO SUBSTITUTABLE

Used when changing a type from *FINAL* to *NOT FINAL*. The altered type then can be used in substitutable tables and columns, as well as in subtypes, instances of dependent tables, and columns.

FORCE

Causes the *CASCADE* operation to go forward, ignoring any errors found in dependent subtypes and tables. All errors are logged to a previously created *EXCEPTIONS* table.

EXCEPTIONS INTO *table_name*

Logs all error data to a table previously created using the system package **DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE**.

In this example, we create a Java SQLJ object type called *address_type*:

```
CREATE TYPE address_type AS OBJECT
  EXTERNAL NAME 'scott.address' LANGUAGE JAVA
  USING SQLDATA ( street1 VARCHAR(30) EXTERNAL NAME 'str1',
    street2      VARCHAR(30) EXTERNAL NAME 'str2',
    city         VARCHAR(30) EXTERNAL NAME 'city',
    state        CHAR(2)    EXTERNAL NAME 'st',
    locality_code CHAR(15)   EXTERNAL NAME 'lc',
    STATIC FUNCTION square_feet RETURN NUMBER
      EXTERNAL VARIABLE NAME 'square_feet',
    STATIC FUNCTION create_addr (str VARCHAR,
      City VARCHAR, state VARCHAR, zip NUMBER)
      RETURN address_type
      EXTERNAL NAME 'create (java.lang.String,
        java.lang.String, java.lang.String, int)
        return scott.address',
    MEMBER FUNCTION rtrims RETURN SELF AS RESULT
      EXTERNAL NAME 'rtrim_spaces () return scott.address' )
  NOT FINAL;
```

We could create a UDT using a *VARRAY* type with four elements:

```
CREATE TYPE employee_phone_numbers AS VARRAY(4) OF CHAR(14);
```

In the following example, we alter the *address_type* that we created earlier by adding a *VARRAY* called *phone_varray*:

```
ALTER TYPE address_type
  ADD ATTRIBUTE (phone phone_varray) CASCADE;
```

In this last example, we'll create a supertype and a subtype called *menu_item_type* and *entry_type*, respectively:

```
CREATE OR REPLACE TYPE menu_item_type AS OBJECT
(id INTEGER, title VARCHAR2(500),
 NOT INSTANTIABLE
 MEMBER FUNCTION fresh_today
 RETURN BOOLEAN)
NOT INSTANTIABLE
NOT FINAL;
```

In the preceding example, we created a type specification (but not the type body) that defines items that may appear on a menu at a café. Included with the type specification is a subprogram method called *fresh_today*, a Boolean indicator that tells whether the menu item is made fresh that day. The *NOT FINAL* clause that appears at the end of the code tells Oracle that this type may serve as the supertype (or base type) to other subtypes that we might derive from it. So now, let's create the *entre_type*:

```
CREATE OR REPLACE TYPE entry_type UNDER menu_item_type
(entre_id INTEGER, desc_of_entre VARCHAR2(500),
OVERRIDING MEMBER FUNCTION fresh_today
RETURN BOOLEAN)
NOT FINAL;
```

PostgreSQL

PostgreSQL supports both an *ALTER TYPE* statement and a *CREATE TYPE* statement used to create a new datatype. PostgreSQL's implementation of the *CREATE TYPE* statement is nonstandard:

```
CREATE TYPE type_name
( INPUT = input_function_name,
OUTPUT = output_function_name
[, INTERNALLENGTH = { int | VARIABLE } ]
[, DEFAULT = value ]
[, ELEMENT = array_element_datatype ]
[, DELIMITER = delimiter_character ]
[, PASSEDBYVALUE ]
[, ALIGNMENT = {CHAR | INT2 | INT4 | DOUBLE} ]
[, STORAGE = {PLAIN | EXTERNAL | EXTENDED | MAIN} ]
[, SEND = send_function ]
[, RECEIVE = receive_function ]
[, ANALYZE = analyze_function ] )
```

PostgreSQL allows you to change the schema or owner of an existing type using the *ALTER TYPE* statement:

```
ALTER TYPE type_name [OWNER TO new_owner_name] [SET SCHEMA new_schema_name]
```

The parameters are as follows:

CREATE TYPE type_name

Creates a new user-defined datatype called *type_name*. The name may not exceed 30 characters in length, nor may it begin with an underscore.

INPUT = input_function_name

Declares the name of a previously created function that converts external argument values into a form usable by the type's internal form.

OUTPUT = output_function_name

Declares the name of a previously created function that converts internal output values to a display format.

INTERNALLENGTH = { int | VARIABLE }

Specifies a numeric value, *int*, for the internal length of the new type, if the datatype is fixed-length. The keyword *VARIABLE* (the default) declares that the internal length is variable.

DEFAULT = *value*

Defines a value for *type* when it defaults.

ELEMENT = *array_element_datatype*

Declares that the datatype is an array and that *array_element_datatype* is the datatype of the array elements. For example, an array containing integer values would be *ELEMENT = INT4*. In general, you should allow the *array_element_datatype* value to default. The only time you might want to override the default is when creating a fixed-length UDT composed of an array of multiple identical elements that need to be directly accessible by subscripting.

DELIMITER = *delimiter_character*

Declares a character to be used as a delimiter between output values of an array produced by the type. Only used with the *ELEMENT* clause. The default is a comma.

PASSEDBYVALUE

Specifies that the values of the datatype are passed by value and not by reference. This optional clause cannot be used for types whose value is longer than the *DATUM* datatype (4 bytes on most operating systems, 8 bytes on a few others).

ALIGNMENT = {CHAR | INT2 | INT4 | DOUBLE}

Defines a storage alignment for the type. Four datatypes are allowed, with each equating to a specific boundary: *CHAR* equals a 1-byte boundary, *INT2* equals a 2-byte boundary, *INT4* equals a 4-byte boundary (the requirement for a variable-length UDT on PostgreSQL), and *DOUBLE* equals an 8-byte boundary.

STORAGE = {PLAIN | EXTERNAL | EXTENDED | MAIN}

Defines a storage technique for variable-length UDTs. (*PLAIN* is required for fixed-length UDTs.) Four types are allowed:

PLAIN

Stores the UDT inline, when declared as the datatype of a column in a table, and uncompressed.

EXTERNAL

Stores the UDT outside of the table without trying to compress it first.

EXTENDED

Stores the UDT as a compressed value if it fits inside the table. If it is too long, PostgreSQL will save the UDT outside of the table.

MAIN

Stores the UDT as a compressed value within the table. Bear in mind, however, that there are situations where PostgreSQL cannot save the UDT within the table because it is just too large. The *MAIN* storage parameter puts the highest emphasis on storing UDTs with all other table data.

SEND = *send_function*

Converts the internal representation of the type to the external binary representation. Usually coded in C or another low-level language.

RECEIVE = *receive_function*

Converts the text's external binary representation to the internal representation. Usually coded in C or another low-level language.

ANALYZE = *analyze_function*

Performs type-specific statistical collection for columns of the type.



More details on the *SEND*, *RECEIVE*, and *ANALYZE* functions are available in the PostgreSQL documentation.

When you create a new datatype in PostgreSQL, it is available *only* in the current database. The user who created the datatype is the owner. When you create a new type, the parameters may appear in any order and are largely optional, except for the first two (the input and output functions).

To create a new datatype, you must create at least two functions before defining the type (see the earlier section “CREATE/ALTER FUNCTION/PROCEDURE Statements”). In summary, you must create an *INPUT* function that provides the type with external values that can be used by the operators and functions defined for the type, as well as an *OUTPUT* function that renders a usable external representation of the datatype. There are some additional requirements when creating the input and output functions:

- The input function should either take one argument of type *OPAQUE* or take three arguments of *OPAQUE*, *OID*, and *INT4*. In the latter case, *OPAQUE* is the input text of a C string, *OID* is the element type for array types, and *INT4* (if known) is the typmod of the destination column.
- The output function should either take one argument of type *OPAQUE* or take two arguments of type *OPAQUE* and *OID*. In the latter case, *OPAQUE* is the datatype itself and *OID* is the element type for array types, if needed.

For example, we can create a UDT called *floorplan* and use it to define a column in two tables, one called *house* and one called *condo*:

```
CREATE TYPE floorplan
  (INTERNALLENGTH=12, INPUT=squarefoot_calc_proc,
  OUTPUT=out_floorplan_proc);
CREATE TABLE house
  (house_plan_id      int4,
   size              floorplan,
   descrip          varchar(30) );
CREATE TABLE condo
  (condo_plan_id      INT4,
   size              floorplan,
   descrip          varchar(30)
   location_id      varchar(7) );
```

SQL Server

SQL Server supports the *CREATE TYPE* statement, but not the *ALTER TYPE* statement. New datatypes can also be added to SQL Server using the non-ANSI system stored procedure *sp_addtype*. The syntax for SQL Server’s implementation of *CREATE TYPE* follows:

```
CREATE TYPE type_name
{ FROM base_type [ ( precision [, scale] ) ] [[NOT] NULL] |
  AS TABLE table_definition |
  CLR_definition }
```

where:

FROM *base_type*

Supplies the datatype upon which the datatype alias is based. The datatype may be one of the following: *BIGINT*, *BINARY*, *BIT*, *CHAR*, *DATE*, *DATETIME*, *DATETIME2*, *DATETIMEOFFSET*, *DECIMAL*, *FLOAT*, *IMAGE*, *INT*, *MONEY*, *NCHAR*, *NTEXT*, *NUMERIC*, *NVARCHAR*, *REAL*, *SMALLDATETIME*, *SMALLINT*, *SMALLMONEY*, *SQL_VARIANT*, *TEXT*, *TIME*, *TINYINT*, *UNIQUEIDENTIFIER*, *VARBINARY*, or *VARCHAR*. Where appropriate to the datatype, a precision and scale may be defined.

[NOT] *NULL*

Specifies whether the type can hold a *NULL* value. When omitted, *NULL* is the default.

AS TABLE *table_definition*

Specifies a user-defined table type with columns, datatypes, keys, constraints (such as *CHECK*, *UNIQUE*, and *PRIMARY KEY*), and properties (such as *CLUSTERED* and *NONCLUSTERED*), just like a regular table.

SQL Server supports the creation of types written in Microsoft .NET Framework common language runtime (CLR) methods that can take and return user-supplied parameters. These types have similar *CREATE* and *ALTER* declarations to regular SQL types; however, the code bodies are external assemblies. Refer to the SQL Server documentation if you want to learn more about programming routines using the CLR.

User-defined types created with **sp_addtype** are accessible by the public database role. However, permission to access user-defined types created with **CREATE TYPE** must be granted explicitly, including to **PUBLIC**.

See Also

[CREATE/ALTER FUNCTION/PROCEDURE](#)
[DROP](#)

CREATE/ALTER VIEW Statement

This statement creates a *view* (also known as a *virtual table*). A view acts just like a table but is actually defined as a query. When a view is referenced in a statement, the result set of the query becomes the content of the view for the duration of that statement. Almost any valid *SELECT* statement can define the contents of a view, though some platforms restrict the certain clauses of the *SELECT* statement and certain set operators.

In some cases, views can be updated, causing the view changes to be translated to the underlying data in the base tables. Some database platforms support a *materialized view*; that is, a physically created table that is defined with a query just like a view.



ALTER VIEW is not an ANSI-supported statement.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
CREATE [RECURSIVE] VIEW view_name {[(column[, ...])] |  
[OF udt_name [UNDER supertype_name]  
[REF IS column_name {SYSTEM GENERATED | USER GENERATED | DERIVED}]  
[column_name WITH OPTIONS SCOPE table_name]]}  
AS select_statement [WITH [CASCADED | LOCAL] CHECK OPTION]
```

Keywords

CREATE VIEW *view_name*

Creates a new view using the supplied name.

RECURSIVE

Creates a view that derives values from itself. It must have a *column* clause and may not use the *WITH* clause.

[(*column*[, ...])]

Names all of the columns in the view. The number of columns declared here must match the number of columns generated by the *select_statement*. When omitted, the columns in the view derive their names from the columns in the table. This clause is required when one or more of the columns is derived and does not have a base table column to reference.

OF *udt_name* [UNDER *supertype_name*]

Defines the view on a UDT rather than on the *column* clause. The typed view is created using each attribute of the type as a column in the view. Use the *UNDER* clause to define a view on a subtype.

REF IS *column_name* {SYSTEM GENERATED | USER GENERATED | DERIVED}

Defines the object-ID column for the view.

column_name* WITH OPTIONS SCOPE *table_name

Provides scoping for a reference column in the view. (Since the columns are derived from the type, there is no column list. Therefore, to specify column options, you must use *column_name* WITH OPTIONS....)

AS *select_statement*

Defines the exact *SELECT* statement that provides the data of the view.

WITH [CASCADED | LOCAL] CHECK OPTION

Used *only* on views that allow updates to their base tables. Ensures that only data that may be read by the view may be inserted, updated, or deleted by the view. For example, if a view of **employees** showed salaried employees but not hourly employees, it would be impossible to insert, update, or delete hourly employee records through that view. The *CASCADED* and *LOCAL* options of the *CHECK OPTION* clause are used for nested views. *CASCADED* performs the check option for the current view and all views upon which it is built; *LOCAL* performs the check option only for the current view, even when it is built upon other views.

Rules at a Glance

Views are usually only as effective as the queries upon which they are based. That is why it is important to be sure that the defining *SELECT* statement is speedy and well written. The simplest view is based on the entire contents of a single table:

```
CREATE VIEW employees
AS
SELECT *
FROM employee_tbl;
```

A column list also may be specified after the view name. The optional column list contains aliases serving as names for each element in the result set of the *SELECT* statement. If you use a column list, you must provide a name for every column returned by the *SELECT* statement. If you don't use a column list, the columns of the view will be named whatever the columns in the *SELECT* statement are called. You will sometimes see complex *SELECT* statements within a view that make heavy use of AS clauses for all columns, because that allows the developer of the view to put meaningful names on the columns without including a column list.

The ANSI standard specifies that you must use a column list or an AS clause. However, some vendors allow more flexibility, so follow these rules for when to use an AS clause:

- When the *SELECT* statement contains calculated columns, such as *(salary * 1.04)*
- When the *SELECT* statement contains fully qualified column names, such as **pubs.scott.employee**
- When the *SELECT* statement contains more than one column of the same name (though with separate schema or database prefixes)

For example, the following two view declarations have the same functional result:

```
-- Using a column list
CREATE VIEW title_and_authors
  (title, author_order, author, price, avg_monthly_sales,
  publisher)
AS
SELECT t.title, ta.au_ord, a.au_lname, t.price, (t.ytd_sales / 12),
       t.pub_id
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0;

-- Using the AS clause with each column
CREATE VIEW title_and_authors
AS
SELECT t.title AS title, ta.au_ord AS author_order,
       a.au_lname AS author, t.price AS price,
       (t.ytd_sales / 12) AS avg_monthly_sales, t.pub_id AS publisher
FROM authors AS a
JOIN titleauthor AS ta ON a.au_id = ta.au_id
JOIN titles AS t ON t.title_id = ta.title_id
WHERE t.advance > 0
```

Alternatively, you can change the titles of columns using the column list. In this case, we'll change **avg_monthly_sales** to **avg_sales**. Note that the code overrides the default column names provided by the AS clauses (in bold):

```

CREATE VIEW title_and_authors
  (title, author_order, author, price, avg_sales, publisher)
AS
  SELECT t.title AS title, ta.au_ord AS author_order,
         a.au_lname AS author, t.price AS price,
         (t.ytd_sales / 12) AS avg_monthly_sales, t.pub_id AS publisher
    FROM authors AS a
   JOIN titleauthor AS ta ON a.au_id = ta.au_id
   JOIN titles AS t ON t.title_id = ta.title_id
  WHERE t.advance > 0;

```

An ANSI-standard view can update the base table(s) it is based upon if it meets the following conditions:

- The view does not have *UNION*, *EXCEPT*, or *INTERSECT* operators.
- The defining *SELECT* statement does not contain *GROUP BY* or *HAVING* clauses.
- The defining *SELECT* statement does not contain any reference to non-ANSI pseudocolumns such as *ROWNUM* or *ROWGUIDCOL*.
- The defining *SELECT* statement does not contain the *DISTINCT* clause.
- The view is not materialized.

This example shows a view named **california_authors** that allows data modifications to apply only to authors within the state of California:

```

CREATE VIEW california_authors
AS
  SELECT au_lname, au_fname, city, state
    FROM authors
   WHERE state = 'CA'
  WITH LOCAL CHECK OPTION

```

The view shown in this example would accept *INSERT*, *DELETE*, and *UPDATE* statements against the base table but guarantee that all inserted, updated, or deleted records contain a **state** of 'CA' using the *WITH...CHECK* clause.

The most important rule to remember when updating a base table through a view is that all columns in a table that are defined as *NOT NULL* must receive a not-NULL value when receiving a new or changed value. You can do this explicitly by directly inserting or updating a not-NULL value into the column, or by relying on a default value. In addition, views do not lift constraints on the base table. Thus, the values being inserted into or updated in the base table must meet all the constraints originally placed on the table through unique indexes, primary keys, *CHECK* constraints, etc.

Programming Tips and Gotchas

Views also can be built upon other views, but this is inadvisable and usually considered bad practice. Depending on the platform, such a view may take longer to compile, but may offer the same performance as a transaction against the base table(s). On other platforms, where each view is dynamically created as it is invoked, nested views may take a long time to return a result set because each level of nesting means that another query must be processed before a result set is returned to the user. In this worst-case scenario, a three-level nested view must make three correlated query calls before it can return results to the user.

Although materialized views are defined like views, they take up space more like tables. Ensure that you have enough space available for the creation of materialized views.

MySQL

MySQL supports both a *CREATE VIEW* and an *ALTER VIEW* statement. MySQL doesn't support SQL3 recursive views, UDT and supertyped views, or views using *REF*. The syntax for both statements follows:

```
{ALTER | CREATE [OR REPLACE]}
[ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
[DEFINER = {user_name | CURRENT_USER}]
[SQL SECURITY {DEFINER | INVOKER}]
VIEW view_name [(column[, ...])]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

where:

ALTER | CREATE [OR REPLACE]

Alters an existing view or creates (or replaces) a view.

ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}

Specifies how MySQL should process the view. *MERGE* tells MySQL to merge the query plans of the query referencing the view and the underlying view itself to achieve optimal performance. *TEMPTABLE* tells MySQL to first retrieve the results of the view into a temporary table, then act upon the query that called the view against the temporary table. *UNDEFINED* tells MySQL to choose the best algorithm to process the view. When this clause is omitted, *UNDEFINED* is the default.

DEFINER = {user_name | CURRENT_USER}

Specifies the user account to use when checking privileges. You may specify either a pre-existing user, or the user who issued the *CREATE TRIGGER* statement (i.e., the *CURRENT_USER*). *CURRENT_USER* is the default when this clause is omitted.

SQL SECURITY {DEFINER | INVOKER}

Specifies the security context under which the view runs: either that of the user that created the view (*DEFINER*, the default when this clause is omitted), or the user running the view (*INVOKER*).

In MySQL, tables and views may not have the same names, because they share the same namespace.

Oracle

Oracle supports extensions to the ANSI standard to create object-oriented views, XMLType views, and views that support *LOB* and object types:

```
CREATE [OR REPLACE] [[NO] FORCE] VIEW view_name
  [[(column[, ...]) [constraint_clause] ] |
   [OF type_name {UNDER parent_view |
     WITH OBJECT IDENTIFIER {DEFAULT | (attribute[, ...])}
     [(constraint_clause)]} ] |
   [OF XMLTYPE [ [XMLSCHEMA xml_schema_url] ELEMENT
     {element | xml_schema_url # element} ]
     WITH OBJECT IDENTIFIER {DEFAULT | (attribute[, ...])} ] ]]
```

```
AS
  (select statement)
  [WITH [READ ONLY | CHECK OPTION [CONSTRAINT constraint_name]]]
```

Oracle's implementation of the *ALTER VIEW* statement supports added capabilities, such as adding, dropping, or modifying constraints associated with the view. In addition, the *ALTER VIEW* statement will explicitly recompile a view that is invalid, enabling you to locate recompilation errors before runtime. This recompiling feature enables you to determine whether a change in a base table negatively impacts any dependent views:

```
ALTER VIEW view_name
  {ADD constraint_clause |
  MODIFY CONSTRAINT constraint_clause [NO]RELY] |
  DROP {PRIMARY KEY | CONSTRAINT constraint_clause | UNIQUE (column[, ...])} |
  COMPILE
```

The parameters are:

OR REPLACE

Replaces any existing view of the same *view_name* with the new view.

[NO] FORCE

The *FORCE* clause creates the view regardless of whether the base tables exist or the user creating the view has privileges to read from or write to the base tables, views, or functions defined in the view. The *FORCE* clause also creates the view regardless of any errors that occur during view creation. The *NO FORCE* clause creates the view only if the base tables and proper privileges are in place.

constraint_clause

Allows you to specify constraints on views for *CREATE VIEW* (see the section on *CREATE TABLE* for details). Using *ALTER VIEW*, this clause also allows you to affect a named, pre-existing constraint. You can define the constraint at the view level, similar to a table-level view, or at the column or attribute level. Note that although Oracle allows you to define constraints on a view, it doesn't yet enforce them. Oracle supports constraints on a view in *DISABLE* and *NOVALIDATE* modes.

OF *type_name*

Declares that the view is an object view of type *type_name*. The columns of the view correspond directly to the attributes returned by *type_name*, where *type_name* is a previously declared type (see the section on *CREATE TYPE*). You do not specify column names for object and XMLType views.

UNDER *parent_view*

Specifies a subview based on a pre-existing *parent_view*. The subview must be in the same schema as the parent view, the *type_name* must be an immediate subtype of the *parent_view*, and only one subview is allowed.

WITH OBJECT IDENTIFIER {DEFAULT | (*attribute*[, ...])}

Defines the root object view as well as any *attributes* of the object type used to identify each row of the object view. The attributes usually correspond to the primary key columns of the base table and must uniquely identify each row of the view. This clause is incompatible with subviews and dereferenced or pinned *REF* keys. The *DEFAULT* keyword uses the implicit object identifier of the base object table or view.

OF XMLTYPE [[XMLSCHEMA *xml_schema_url*] ELEMENT {*element* | XMLSCHEMA *url* # *element*}] WITH OBJECT IDENTIFIER {DEFAULT | (*attribute*[, ...])}

Specifies that the view will return XMLType instances. Specifying the optional *xml_schema_url* as a pre-registered XMLSchema and element name further constrains the returned XML as an element in that XMLSchema. The *WITH OBJECT IDENTIFIER* clause specifies the identifier that uniquely identifies each row of the XMLType view. One or more attributes may use non-aggregate functions like *EXTRACTVALUE* to obtain the identifiers from the resultant XMLType.

WITH READ ONLY

Ensures that the view is used only to retrieve data, not to modify it.

WITH CHECK OPTION [*CONSTRAINT constraint_name*]

Forces the view to accept only inserted and updated data that can be returned by the view's *SELECT* statement. Alternately, you can specify a single *CHECK OPTION constraint_name* that exists on the base table that you want to enforce. If the constraint is not named, Oracle names the constraint **SYS_Cn**, where *n* is an integer.

ADD *constraint_clause*

Adds a new constraint to the view. Oracle supports constraints only in *DISABLE* and *NOVALIDATE* modes.

MODIFY CONSTRAINT *constraint_clause* [NO]RELY

Changes the *RELY* or *NORELY* setting of an existing view constraint. (*RELY* and *NORELY* are explained in the section on *CREATE TABLE*.)

DROP {PRIMARY KEY | CONSTRAINT *constraint_clause* | UNIQUE (*column*[, ...])}

Drops an existing constraint on a view.

COMPILE

Recompiles the view.

Any dblinks in the view's *SELECT* statement must be declared using the *CREATE DATABASE LINK...* *CONNECT TO* statement. Any view containing flashback queries will have its *AS OF* clause evaluated at each invocation of the view, not when the view is compiled.

In this example, we create a view that has an added constraint:

```
CREATE VIEW california_authors (last_name, first_name,
    author_ID UNIQUE RELY DISABLE NOVALIDATE,
    CONSTRAINT id_pk PRIMARY KEY (au_id) RELY DISABLE NOVALIDATE)
AS
    SELECT au_lname, au_fname, au_id
    FROM authors
    WHERE state = 'CA';
```

We might also wish to create an object view on an Oracle database and schema. This example creates the type and the object view:

```
CREATE TYPE inventory_type AS OBJECT
( title_id NUM(6),
  warehouse wrhs_typ,
  qty NUM(8) );
CREATE VIEW inventories OF inventory_type
WITH OBJECT IDENTIFIER (title_id)
AS
    SELECT i.title_id, wrhs_typ(w.wrhs_id, w.wrhs_name,
        w.location_id), i.qty
```

```
FROM inventories i
JOIN warehouses w ON i.wrhs_id = w.wrhs_id;
```

We could recompile the `inventory_type` view like this:

```
ALTER VIEW inventory_type COMPILE;
```

An updatable view in Oracle cannot contain any of the following:

- The `DISTINCT` clause
- `UNION`, `INTERSECT`, or `MINUS` clauses
- Joins that cause inserted or updated data to affect more than one table
- Aggregate or analytic functions
- `GROUP BY`, `ORDER BY`, `CONNECT BY`, or `START WITH` clauses
- Subqueries or collection expressions in the `SELECT` item list (subqueries are acceptable in the `SELECT` statement's `WHERE` clause)
- Update pseudocolumns or expressions

There are some restrictions on how subviews and materialized views can be defined in Oracle:

- The subview must use aliases for `ROWID`, `ROWNUM`, or `LEVEL` pseudocolumns.
- The subview cannot query `CURRVAL` or `NEXTVAL` pseudocolumns.
- The subview cannot contain the `SAMPLE` clause.
- The subview evaluates all columns of a `SELECT * FROM...` statement at compile time. Thus, any new columns added to the base table will not be retrieved by the subview until the view is recompiled.

Note that while older versions of Oracle supported partitioned views, this feature has been deprecated. You should use explicitly declared partitions instead.

PostgreSQL

PostgreSQL supports a no-frills subset of the ANSI standard for `CREATE VIEW`:

```
CREATE [OR REPLACE] [TEMP[ORARY]] VIEW view_name [ (column[, ...]) ]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

PostgreSQL does not support `ALTER VIEW`. However, you can use `CREATE OR REPLACE view_name` to substitute the definition of an old view with the definition of a new view. In addition, PostgreSQL allows you to create temporary views, which is an extension to the SQL3 standard.

PostgreSQL's `CREATE VIEW` does not support some of the more complex options that other vendors' versions of this statement do. However, it does allow views to be built on tables and other defined class objects. PostgreSQL views are built only upon other tables, not upon other views. They are always read-only and cannot be used to perform data modifications on the underlying base tables.

SQL Server

SQL Server supports some extensions to the ANSI standard but does not offer object views or subviews:

```
CREATE VIEW view_name [(column[, ...])]
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA}[, ...]]
AS select_statement
[WITH CHECK OPTION]
```

SQL Server's implementation of *ALTER VIEW* allows you to change an existing view without affecting the permissions or dependent objects of the view:

```
ALTER VIEW view_name [(column[, ...])]
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA}[, ...]]
AS select_statement
[WITH CHECK OPTION]
```

The parameters are as follows:

ENCRYPTION

Encrypts the text of the view in the **syscomments** table. This option is usually invoked by software vendors who want to protect their intellectual capital.

SCHEMABINDING

Binds the view to a specific schema, meaning that all objects in the view must be referenced by their full names (both owner and object name). The view and any tables referenced in the view's *SELECT* statement must be fully qualified, as in **pubs.scott.employee**. Views created with *SCHEMABINDING* (and tables referenced by these views) must have the schema binding dropped (via *ALTER VIEW*) before they may be dropped or altered.

VIEW_METADATA

Specifies that SQL Server return metadata about the view (rather than the base table) to calls made from DBLIB or OLEDB APIs. Views created or altered with *VIEW_METADATA* enable their columns to be updated by *INSERT* and *UPDATE INSTEAD OF* triggers.

WITH CHECK OPTION

Forces the view to accept only inserted and updated data that can be returned by the view's *SELECT* statement.

The *SELECT* clause of a SQL Server view cannot:

- Have *COMPUTE*, *COMPUTE BY*, *INTO*, or *ORDER BY* clauses (*ORDER BY* is allowed if you use *SELECT TOP*)
- Reference a temporary table
- Reference a table variable
- Reference more than 1,024 columns, including those referenced by subqueries

Here, we define a SQL Server view with both *ENCRYPTION* and *CHECK OPTION* clauses:

```
CREATE VIEW california_authors (last_name, first_name, author_id)
WITH ENCRYPTION
AS
    SELECT au_lname, au_fname, au_id
    FROM authors
    WHERE state = 'CA'
WITH CHECK OPTION
GO
```

SQL Server allows multiple *SELECT* statements in a view, as long as they are linked with *UNION* or *UNION ALL* clauses. SQL Server also allows functions and hints in a view's *SELECT* statement. A SQL Server view is updatable if all of the conditions in the following list are true:

- The *SELECT* statement has no aggregate functions.
- The *SELECT* statement does not contain *TOP*, *GROUP BY*, *DISTINCT*, or *UNION* clauses.

- The *SELECT* statement has no derived columns (see *SUBQUERY*).
- The *FROM* clause of the *SELECT* statement references at least one table.

SQL Server allows indexes to be created on views (see *CREATE INDEX*). By creating a unique, clustered index on a view, you cause SQL Server to store a physical copy of the view on the database. Changes to the base table are automatically updated in the indexed view. Indexed views consume extra disk space but provide a boost in performance. These views must be built using the *SCHEMABINDING* clause.

SQL Server also allows the creation of local and distributed partitioned views. A *local partitioned view* is a partitioned view where all views are present on the same SQL server. A *distributed partitioned view* is a partitioned view where one or more views are located on remote servers.

Partitioned views must very clearly derive their data from different sources, with each distinct data source joined to the next with a *UNION ALL* statement. Furthermore, all columns of the partitioned views should be selected and identical. (The idea is that you have split the data out logically by means of a frontend application; SQL Server then recombines the data through the partitioned view.) This example shows how the data in the view comes from three separate SQL servers:

```
CREATE VIEW customers
AS
--Select from a local table on server New_York
SELECT *
FROM sales_archive.dbo.customers_A
UNION ALL
SELECT *
FROM houston.sales_archive.dbo.customers_K
UNION ALL
SELECT *
FROM los_angeles.sales_archive.dbo.customers_S
```

Note that each remote server (*New_York*, *houston*, and *los_angeles*) has to be defined as a remote server on all of the SQL servers using the distributed partitioned view.

Partitioned views can greatly boost performance because they can split I/O and user loads across many machines. However, they are difficult to plan, create, and maintain. Be sure to read the vendor documentation for complete details about all the permutations available with partitioned views.

When altering an existing view, SQL Server acquires and holds an exclusive schema lock on the view until the alteration is finished. *ALTER VIEW* drops any indexes that might be associated with a view; you must manually recreate them using *CREATE INDEX*.

See Also

[CREATE/ALTER TABLE](#)
[DROP](#)
[SELECT](#)
[SUBQUERY](#)

DECLARE CURSOR Command

The *DECLARE* command is one of four commands used in cursor processing, along with *FETCH*, *OPEN*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than in a complete set. The *DECLARE CURSOR* command specifies the exact records to be retrieved and manipulated (one row at a time) from a specific table or view.

In other words, cursors are especially important for relational databases because databases are set-based, while most client-centric programming languages are row-based. This is important for two reasons. First, cursors allow programmers to program using methodologies supported by their favorite row-based programming languages. Second, cursors run counter to the default behavior of some relational database platforms, which operate on sets of records, and on those specific platforms cursor operations may be noticeably slower than standard set-based operations.

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

```
DECLARE cursor_name [ {SENSITIVE | INSENSITIVE | ASENSITIVE} ]
  [[NO] SCROLL] CURSOR [{WITH | WITHOUT} HOLD]
  [{WITH | WITHOUT} RETURN]
  FOR select_statement
  [FOR {READ ONLY | UPDATE [OF column[, ...]]}]
```

Keywords

DECLARE cursor_name

Gives the cursor a unique name in the context in which it is defined (for example, in the database or schema where it is created). No other cursors may share the same name.

SENSITIVE | INSENSITIVE | ASENSITIVE

Defines the manner in which the cursor interacts with the source table and the result set retrieved by the cursor query, where:

SENSITIVE

Tells the database to operate directly against the result set, so that the cursor will see changes to its result set as it moves through the records.

INSENSITIVE

Tells the database to create a temporary but separate copy of the result set, so that all changes made against the result set by other operations are invisible to the cursor during the cursor operation.

ASENSITIVE

Allows the database implementation to determine whether to make a copy of the result set. *ASENSITIVE* is the SQL2003 default.

[NO] SCROLL

SCROLL tells the database not to enforce processing one row at a time, using *FETCH NEXT*, but that all forms of the *FETCH* clause are allowed against the result set. *NO SCROLL* enforces processing one row at a time.

{WITH | WITHOUT} HOLD

WITH HOLD tells the cursor to remain open when a transaction encounters the *COMMIT* statement. Conversely, *WITHOUT HOLD* tells the cursor to close when the transaction encounters a *COMMIT* statement. (A cursor that uses *WITH HOLD* can only be and is always closed with a *ROLLBACK* statement or a *CLOSE CURSOR* statement.)

{WITH | WITHOUT} RETURN

Used only in a stored procedure. The *WITH RETURN* clause tells the database to return the result set, if it is still open, when the stored procedure terminates. With the *WITHOUT RETURN* clause, which is the default, all open cursors are implicitly closed when the stored procedure terminates.

FOR *select_statement*

Defines the underlying *SELECT* statement that determines the rows in the result set of the cursor. As with a regular *SELECT* statement, the results of the query may be sorted according to an *ORDER BY* clause.

FOR {READ ONLY | UPDATE [OF *column*[, ...]}}

Specifies that the cursor is not updatable in any way, using *FOR READ ONLY*. This is the default when the cursor is defined with the *SCROLL* or *INSENSITIVE* properties, and when the query contains an *ORDER BY* clause or is against a non-updatable table. Alternately, you can specify *FOR UPDATE OF *column1*, *column2*[, ...]*, defining the columns where you want to execute *UPDATE* statements, or omit the column list to include all columns in the cursor.

Rules at a Glance

The *DECLARE CURSOR* command enables the retrieval and manipulation of records from a table one row at a time. This provides row-by-row processing, rather than the traditional set processing offered by SQL.

At the highest level, these steps are followed when working with a cursor:

1. Create a cursor using *DECLARE*.
2. Open the cursor using *OPEN*.
3. Operate against the cursor using *FETCH*.
4. Dismiss the cursor using *CLOSE*.

The *DECLARE CURSOR* command works by specifying a *SELECT* statement. Each row returned by the *SELECT* statement may be individually retrieved and manipulated. In this Oracle example, the cursor is declared in the declaration block, along with some other variables. The cursor is then opened, fetched against, and then closed in the *BEGIN...END* block that follows:

```
DECLARE CURSOR title_price_cursor IS
  SELECT title, price
  FROM titles
  WHERE price IS NOT NULL;
  title_price_val title_price_cursor%ROWTYPE;
  new_price NUMBER(10,2);
```

DECLARE CURSOR Command

```
BEGIN
  OPEN title_price_cursor;
  FETCH title_price_cursor INTO title_price_val;
  new_price := "title_price_val.price" * 1.25
  INSERT INTO new_title_price
  VALUES (title_price_val.title, new_price)
  CLOSE title_price_cursor;
END;
```

Because this example uses PL/SQL, much of the code is beyond the scope of this book. However, the *DECLARE* block clearly shows that the cursor is declared. In the PL/SQL execution block, the cursor is initialized with the *OPEN* command, values are retrieved with the *FETCH* command, and the cursor finally is terminated with the *CLOSE* command.

The *SELECT* statement is the heart of your cursor, so it is a good idea to test it thoroughly before embedding it in the *DECLARE CURSOR* statement. The *SELECT* statement may operate against a base table or a view. For that matter, read-only cursors may operate against non-updatable views. The *SELECT* statement can have subclauses such as *ORDER BY*, *GROUP BY*, and *HAVING* if it is not updating the base table. If the cursor is *FOR UPDATE*, however, it is a good idea to exclude these clauses from the *SELECT* statement.

Local cursors are sometimes used as output parameters of stored procedures. Thus, a stored procedure could define and populate a cursor and pass that cursor to a calling batch or stored procedure.

In this next example from Microsoft SQL Server, a cursor from the **publishers** table is declared and opened. The cursor takes the first record from **publishers** that matches the *SELECT* statement and inserts it into another table; it then moves to the next record and then the next, until all records are processed. Finally, the cursor is closed and deallocated (*DEALLOCATE* is used only with Microsoft SQL Server):

```
DECLARE @publisher_name VARCHAR(20)
DECLARE pub_cursor CURSOR
FOR SELECT pub_name FROM publishers
  WHERE country <> 'USA'
OPEN pub_cursor
FETCH NEXT FROM pub_cursor INTO @publisher_name
WHILE @@FETCH_STATUS = 0
BEGIN
  INSERT INTO foreign_publishers VALUES(@publisher_name)
END
CLOSE pub_cursor
DEALLOCATE pub_cursor
```

In this example, you see how the cursor moves through a set of records. (The example was intended only to demonstrate this concept, since there is actually a better way to accomplish the same task; namely, an *INSERT...SELECT* statement.)

Programming Tips and Gotchas

Most platforms do not support dynamically executed cursors. Rather, cursors are embedded within an application program, stored procedure, user-defined function, etc. In other words, the various statements for creating and using a cursor are usually used only in the context of a stored procedure or other database-programming object, not in a straight SQL script.

For ease of programming and migration, don't use the *SCROLL* clause or the various sensitivity keywords. Some platforms support only forward-scrolling cursors, so you can avoid headaches later by just sticking to the simplest form of cursor.

Cursors behave differently on the various database platforms when crossing a transaction boundary—for example, when a set of transactions is rolled back in the midst of a cursor operation. Be sure to familiarize yourself with exactly how each database platform behaves in these circumstances.

MySQL does not support server-side cursors in the ANSI SQL style, but it does support extensive C programming extensions that provide the same functionality.

MySQL

MySQL supports a subset of the SQL3 standard syntax:

```
DECLARE cursor_name
FOR select_statement
```

MySQL only allows cursors within stored procedures, functions, and triggers. MySQL cursors are always read-only, non-updatable, non-scrollable (the cursor only moves in one direction and cannot skip rows), and asensitive (meaning that the server will choose whether to make a copy of the result table according to what is most expedient). MySQL will not allow you to create more than one cursor in a block with the same name. Cursors must be declared after variables and conditions, but before handlers.

Oracle

Oracle has a rather interesting implementation of cursors. In reality, all Oracle data-modification statements (*INSERT*, *UPDATE*, *DELETE*, and *SELECT*) implicitly open a cursor. For example, a C program accessing Oracle would not need to issue a *DECLARE CURSOR* statement to retrieve data on a record-by-record basis, because that is the implicit behavior of Oracle. Because of this behavior, you'll only use *DECLARE CURSOR* in PL/SQL constructs such as stored procedures, not in a script that is purely SQL.



Since cursors can only be used in stored procedures and user-defined functions in Oracle, they are documented in the Oracle PL/SQL reference material, not in the SQL reference material.

Oracle utilizes a variation of the *DECLARE CURSOR* statement that supports parameterized inputs, as the following syntax demonstrates:

```
DECLARE CURSOR cursor_name [ (parameter datatype[, ...]) ]
IS select_statement
[FOR UPDATE [OF column_name[, ...]]]
```

where:

[*parameter datatype*[, ...])] IS *select_statement*

Defines the parameter name and datatype of each input parameter as well as the *select_statement* used to retrieve the cursor result set. Serves the same purpose as the ANSI SQL2003 *FOR select_statement* clause.

FOR UPDATE [OF *column_name*]

Defines the cursor or specific columns of the cursor as updatable. Otherwise, the cursor is assumed to be read-only.

In Oracle, variables are not allowed in the *WHERE* clause of the *SELECT* statement unless they are first declared as variables. The parameters are not assigned in the *DECLARE* statement; instead, they are assigned values in the *OPEN* command. This is important since any system function will return an identical value for all rows in the cursor result set.

PostgreSQL

PostgreSQL does not support the *WITH* clauses. It does allow you to return result sets in binary rather than text format. Although the compiler will not cause errors with many of the ANSI keywords, PostgreSQL's implementation of *DECLARE CURSOR* is more limited than it might first appear:

```
DECLARE cursor_name [BINARY] [INSENSITIVE] [[NO] SCROLL] CURSOR
[ {WITH | WITHOUT} HOLD]
FOR select_statement
[FOR {READ ONLY | UPDATE [OF column_name[, ...]]}]
```

where:

BINARY

Forces the cursor to retrieve binary-formatted data rather than text-formatted data.

INSENSITIVE

Indicates that data retrieved by the cursor is unaffected by updates from other processes or cursors. This is PostgreSQL's default behavior, so omitting this keyword has no effect.

[NO] SCROLL

Allows multiple rows to be retrieved by a single *FETCH* operation in either a forward or backward direction. Be aware that *SCROLL* can slow down processing. *NO SCROLL* specifies that the cursor reads only in a forward direction and that it does not skip any records.

{WITH | WITHOUT} HOLD

WITH HOLD specifies that the cursor can continue to be used after the transaction that created it has completed a *COMMIT*. *WITHOUT HOLD*, the default, specifies that the cursor cannot be used outside of the transaction that created it.

FOR {READ-ONLY | UPDATE [OF *column_name*[, ...]]}

Indicates that a cursor is opened using read-only mode or update mode, respectively. However, PostgreSQL only supports read-only cursors. Consequently, the *FOR READ-ONLY* clause has no effect, while the *FOR UPDATE [OF *column_name*[, ...]]* clause produces an informational error message.

PostgreSQL closes any existing cursor when a newly declared cursor is created with the same name. Binary cursors tend to be faster because PostgreSQL stores data as binary on the backend. However, user applications are only text-aware. Therefore, you'll have to build in binary handling for any *BINARY* cursors.

PostgreSQL allows cursors only within a transaction. You should enclose a cursor within a *BEGIN* and *COMMIT* or *ROLLBACK* transaction block.

PostgreSQL does not support an explicit *OPEN* cursor statement. Cursors are considered open when they are declared. So, to declare and open a cursor in PostgreSQL, you could use code like this:

```
DECLARE pub_cursor CURSOR
FOR SELECT pub_name FROM publishers
WHERE country <> 'USA';
```

SQL Server

SQL Server supports the ANSI standard, as well as a good many extensions that provide flexibility in how a cursor scrolls through a result set and manipulates data, as shown in the following syntax:

```
DECLARE cursor_name CURSOR
  [LOCAL | GLOBAL] [INSENSITIVE | FORWARD_ONLY | SCROLL]
  [STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
  [READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
  [TYPE_WARNING]
  FOR select_statement
  [FOR {READ ONLY | UPDATE [OF column_name[, ...] ]} ]
```

The parameters are as follows:

LOCAL | GLOBAL

Scopes the cursor for either the local Transact-SQL batch or makes the cursor available to all Transact-SQL batches issued by the current session via *OPEN* and *FETCH* statements, respectively. A global cursor name can be referenced by any stored procedure, function, or Transact-SQL batch executed in the current session. A global cursor is implicitly deallocated at disconnect, but a local cursor must be manually deallocated. The *LOCAL* and *GLOBAL* keywords are optional. The default behavior, if neither is specified, is defined by the *default to local* database property.

INSENSITIVE | FORWARD_ONLY | SCROLL

Determines how the cursor will move through the result set, with three options:

INSENSITIVE

Creates the result set as a table in **TEMPDB**. Changes to the base table are not reflected in the cursor result set. Not compatible with SQL Server extensions to the *DECLARE CURSOR* command such as *LOCAL*, *GLOBAL*, *STATIC*, *KEYSET*, *DYNAMIC*, *FAST_FORWARD*, etc. Can only be used in a SQL92-style *DECLARE CURSOR* statement, such as *DECLARE sample CURSOR INSENSITIVE FOR select_statement FOR UPDATE*.

FORWARD_ONLY

Indicates that the cursor must scroll from the first to the last record in the result set and that *FETCH NEXT* is the only supported form of the *FETCH* statement. The cursor is assumed to be *DYNAMIC* unless *STATIC* or *KEYSET* is used. *FAST_FORWARD* is mutually exclusive of *FORWARD_ONLY*.

SCROLL

Enables all *FETCH* options (*ABSOLUTE*, *FIRST*, *LAST*, *NEXT*, *PRIOR*, and *RELATIVE*). Otherwise, only *FETCH NEXT* is available. If *DYNAMIC*, *STATIC*, or *KEYSET* is used, the cursor defaults to *SCROLL* behavior. *FAST_FORWARD* is mutually exclusive of *SCROLL*.

STATIC | KEYSET | DYNAMIC | FAST_FORWARD

Determines how records are manipulated in the result set. These settings are incompatible with the *FOR READ ONLY* and *FOR UPDATE* clauses. There are four options:

STATIC

Makes a temporary copy of the result set data and stores it in **tempdb**. Modifications made to the source table or view do not show up when processing the cursor. *STATIC* cursors cannot modify data in the source table or view.

KEYSET

Makes a temporary copy of the result set with membership and fixed row order (also known as a keyset) in **tempdb**. The keyset correlates the data in the cursor result set to the base table or view, so that the cursor can see changes made to the underlying data. Rows that have been deleted or updated show an `@@FETCH_STATUS` of `-2` (unless the update was done using `UPDATE... WHERE CURRENT OF`, in which case they are fully visible), while rows inserted by other users are not visible at all.

DYNAMIC

Determines the records in the cursor result set as each `FETCH` operation is executed. Thus, **DYNAMIC** cursors see all changes made to the base table or view, even those by other users. Because the result set may be in constant flux, **DYNAMIC** cursors do not support `FETCH ABSOLUTE`.

FAST_FORWARD

Creates a `FORWARD_ONLY`, `READ_ONLY` cursor that quickly reads through the entire cursor result set at one time.

READ_ONLY | SCROLL_LOCKS | OPTIMISTIC

Determines concurrency and positional update behavior for the cursor. These settings are incompatible with `FOR READ ONLY` and `FOR UPDATE`. The allowable parameters are:

READ_ONLY

Prevents updates to the cursor and disallows the cursor from being referenced in `UPDATE` or `DELETE` statements that contain `WHERE CURRENT OF`.

SCROLL_LOCKS

Ensures that positional updates and deletes succeed by locking the cursor result set rows as they are read into the cursor. The locks are held until the cursor is closed and deallocated. **SCROLL_LOCKS** is mutually exclusive of **FAST_FORWARD**.

OPTIMISTIC

Ensures that positional updates and deletes succeed unless the row being updated or deleted in the cursor result set has changed since it was read into the cursor. SQL Server accomplishes this by comparing a timestamp, or a checksum value when no timestamp exists, on the columns. It does not lock the rows in the cursor result set. **OPTIMISTIC** is mutually exclusive of **FAST_FORWARD**.

TYPE_WARNING

Warns the user when the cursor is implicitly converted from one type to another (for example, from **SCROLL** to **FORWARD_ONLY**).

FOR {READ ONLY | UPDATE [OF *column_name*[, ...]}}

FOR READ ONLY identifies the cursor as read-only, using the ANSI-standard syntax. This clause is not compatible with the other type identifiers discussed previously; it should only be used with **INSENSITIVE**, **FORWARD_ONLY**, and **SCROLL**. **FOR UPDATE** allows updates to columns in the cursor using `UPDATE` and `DELETE` statements with the `WHERE CURRENT OF` clause. If **FOR UPDATE** is used without a column list, all columns in the cursor are updatable. Otherwise, only those columns listed are updatable.



Microsoft SQL Server allows two basic forms of syntax for the *DECLARE CURSOR* statement. *These syntax forms are not compatible!* The basic forms of syntax are SQL92-compatible and Transact-SQL extensions. You *cannot* mix together keywords of the two forms.

The SQL92 compatibility syntax for *DECLARE CURSOR* is:

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name[, ...] ] } ]
```

while the Transact-SQL extensions syntax for *DECLARE CURSOR* is:

```
DECLARE cursor_name CURSOR
[ LOCAL | GLOBAL ] [ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name[, ...] ] ]
```

The SQL92 compatibility syntax is supported to enable your code to be more portable. The Transact-SQL extensions syntax enables you to define cursors using the same cursor types that are available in popular database APIs such as ODBC, OLE-DB, and ADO, but it may only be used in SQL Server stored procedures, user-defined functions, triggers, and ad hoc queries.

If you do define a Transact-SQL extension cursor but do not define concurrency behavior using the OPTIMISTIC, READ_ONLY, or SCROLL_LOCKS keywords, then:

- The cursor defaults to *READ-ONLY* if was defined as *FAST_FORWARD* or *STATIC*, or the user has insufficient privileges to update the base table or view.
- The cursor defaults to *OPTIMISTIC* if the cursor was defined as *DYNAMIC* or *KEYSET*.

Note that variables may be used in the *select_statement* of a SQL Server cursor, but the variables are evaluated when the cursor is declared. Thus, a cursor containing a column based on the system function *GETDATE()* will have the same date and time for every record in the cursor result set.

In the following SQL Server example, we use a *KEYSET* cursor to change blank spaces to dashes in the **phone** column of the **authors** table:

```
SET NOCOUNT ON
DECLARE author_name_cursor CURSOR LOCAL KEYSET TYPE_WARNING
    FOR SELECT au_fname FROM pubs.dbo.authors
DECLARE @name varchar(40)
OPEN author_name_cursor
FETCH NEXT FROM author_name_cursor INTO @name
WHILE (@@fetch_status <> -1)
BEGIN
    -- @@fetch_status checks for 'record not found' conditions and errors
    IF (@@fetch_status <> -2)
    BEGIN
        PRINT 'updating record for ' + @name
        UPDATE pubs.dbo.authors
```

DELETE Statement

```
SET phone = replace(phone, ' ', '-')
WHERE CURRENT OF author_name_cursor
END
FETCH NEXT FROM author_name_cursor INTO @name
END
CLOSE author_name_cursor
DEALLOCATE author_name_cursor
GO
```

See Also

CLOSE CURSOR
FETCH
OPEN

DELETE Statement

The *DELETE* statement erases records from a specified table or tables. *DELETE* statements acting against tables are sometimes called *search deletes*. The *DELETE* statement may also be used in conjunction with a cursor. *DELETE* statements acting upon the rows of a cursor are sometimes called *positional deletes*.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
DELETE FROM { table_name | ONLY (table_name) }
[ { WHERE search_condition | WHERE CURRENT OF cursor_name } ]
```

Keywords

FROM { table_name | ONLY (table_name) }

Identifies the table (called *table_name*) from which rows will be deleted. The *table_name* assumes the current schema if one is not specified. You may alternately specify a single table view name. *FROM* is mandatory, except in the *DELETE...WHERE CURRENT OF* statement. When not using the *ONLY* clause, do not enclose the *table_name* in parentheses. *ONLY* restricts cascading of the deleted records to any subtables of the target table or view. This clause affects only typed (object-oriented) tables and views. If used with a non-typed table or view, it is ignored and does not cause an error.

WHERE search_condition

Defines search criteria for the *DELETE* statement, using one or more *search_condition* clauses to ensure that only the target rows are deleted. Any legal *WHERE* clause is acceptable. Typically, these criteria are evaluated against each row of the table before the deletion occurs.

WHERE CURRENT OF cursor_name

Restricts the *DELETE* statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

The *DELETE* statement erases rows from a table or view. Space released by erased rows will be returned to the database where the table is located, though this may not happen immediately.

A simple *DELETE* statement that erases all records in a given table has no *WHERE* clause, as in the following:

```
DELETE FROM sales;
```

You can use any valid *WHERE* clause to filter records that you do not want to delete. All three of the following examples show valid *DELETE* statements, and since all are search deletes, they all include the *FROM* clause:

```
DELETE FROM sales
WHERE qty IS NULL;
DELETE FROM suppliers
WHERE supplierid = 17
      OR companyname = 'Tokyo Traders';
DELETE FROM distributors
WHERE postalcode IN
      (SELECT territorydescription FROM territories);
```

Note that in the positional delete, the *FROM* clause is not required.

In some cases, you may wish to delete a specific row that is being processed by a declared and open cursor:

```
DELETE titles WHERE CURRENT OF title_cursor;
```

This query assumes that you have declared and opened a cursor named *title_cursor*; whichever row the cursor is on will be deleted when the command is executed.

Programming Tips and Gotchas

It is rare to issue a *DELETE* statement without a *WHERE* clause, because this results in *all* rows being deleted from the affected table. You should first issue a *SELECT* statement with the same *WHERE* clause you intend to use in the *DELETE* statement. That way, you can be sure exactly which records will be deleted.

If it becomes necessary to remove all the rows in a table, you should consider using the non-ANSI though very common *TRUNCATE TABLE* statement. In those databases that support the command, *TRUNCATE TABLE* is usually a faster method to physically remove all rows. *TRUNCATE TABLE* is faster than *DELETE* because the deletion of individual records is not logged. The reduction of logging overhead saves considerable time when erasing a large number of records, but on some platforms this makes rollback of a *TRUNCATE* statement impossible. Furthermore, on some database platforms all foreign keys on the table must be dropped before issuing a *TRUNCATE* statement.

MySQL

MySQL allows a number of extensions to the ANSI standard, but it does not support the *WHERE CURRENT OF* clause. The syntax is shown here:

```
DELETE [LOW_PRIORITY] [QUICK] [table_name[.*][, ...]]
{FROM table_name[.*][, ...] | [USING table_name[.*][, ...]]}
[WHERE search_condition]
[ORDER BY clause]
[LIMIT nbr_of_rows]
```

The parameters are:

LOW PRIORITY

Delays the execution of *DELETE* until no other clients are reading from the table.

QUICK

Prevents the storage engine from merging index leaves during the *delete* operation.

DELETE *table_name*[, ...]

Enables you to delete from more than one table at a time. Tables listed before the *FROM* clause, assuming one or more tables appear in the *FROM* clause, will be the target of the delete operation. That is, if more than one table appears before the *FROM* clause, all matching records in all of the tables will be deleted.

FROM *table_name*[.*]

Specifies the table or tables from which records will be deleted. (The *.** clause is an option to improve compatibility with MS-Access.) If tables are listed before the *FROM* clause, the table or tables in the *FROM* clause are assumed to be used to support a join or lookup operation.

USING *table_name*[.*][, ...]

Substitutes the table or tables before the *FROM* clause with those after the *FROM* clause.

ORDER BY *clause*

Specifies the order in which rows will be deleted. This is useful only in conjunction with *LIMIT*.

LIMIT *nbr_of_row*

MySQL also can place an arbitrary cap on the number of records deleted before control is passed back to the client using the *LIMIT nbr_of_rows* clause.

MySQL allows deletion from more than one table at a time. For example, the following two *DELETE* statements are functionally equivalent:

```
DELETE orders FROM customers, orders
WHERE customers.customerid = orders.customerid
    AND orders.orderdate BETWEEN '19950101' AND '19951231'
DELETE FROM orders USING customers, orders
WHERE customers.customerid = orders.customerid
    AND orders.orderdate BETWEEN '19950101' AND '19951231'
```

In the preceding examples, we delete all **orders** made by **customers** during the year 1995. Note that you cannot use *ORDER BY* or *LIMIT* clauses in a multitable delete like those just shown.

You can also delete records in orderly batches using the *ORDER BY* clause in conjunction with the *LIMIT* clause:

```
DELETE FROM sales
WHERE customerid = 'TORTU'
ORDER BY customerid
LIMIT 5
```

MySQL exhibits several behaviors that speed up delete operations. For example, it normally returns the number of records deleted when it completes the operation, but it will return zero as the number of deleted records when you delete all records from a table because it is faster to do so than to count the actual number of rows deleted. In *AUTOCOMMIT* mode, MySQL will even substitute a *TRUNCATE* statement for a *DELETE* statement without a *WHERE* clause because *TRUNCATE* is faster.

Note that the speed of a MySQL delete operation is directly related to the number of indexes on the table and the available index cache. You can speed up delete operations by executing the command against tables with few or no indexes or by increasing the size of the index cache.

Oracle

Oracle allows you to delete rows from tables, views, materialized views, nested subqueries, and partitioned views and tables, as follows:

```
DELETE [FROM]
  {table_name | ONLY (table_name)} [alias]
  [{PARTITION (partition_name) |
  SUBPARTITION (subpartition_name)}] |
  (subquery [WITH {READ ONLY |
  CHECK OPTION [CONSTRAINT constraint_name]}]) |
  TABLE (collection_expression) [ (+) ]}
[hint]
[WHERE search_condition]
[RETURNING expression[, ...] INTO variable[, ...]]
[LOG ERRORS [INTO [schema.]table_name] [(simple_expression)]]
[REJECT LIMIT {UNLIMITED |int }]]
```

The parameters are:

table_name [alias]

Specifies the table, view, materialized view, or partitioned table or view from which the records will be deleted. You may optionally prepend a schema identifier to the *table_name* or append a database link. Otherwise, Oracle will assume the user's default schema and the local database server. You may also optionally apply an *alias* to the *table_name*. The alias is required if the target table references an object type attribute or method.

PARTITION partition_name

Applies the delete operation to the named partition, rather than to the entire table. You are not required to name a partition when deleting from a partitioned table, but it can, in many cases, help reduce the complexity of the *WHERE* clause.

SUBPARTITION subpartition_name

Applies the operation to a named subpartition, rather than to the entire table.

(subquery [WITH {READ ONLY | CHECK OPTION [CONSTRAINT constraint_name]}])

Specifies that the target for deletion is a nested subquery, not a table, view, or other database object. The parameters of this clause are:

subquery

Describes the *SELECT* statement that makes up the subquery. The subquery can be any standard subquery, though it may not contain an *ORDER BY* clause.

WITH READ ONLY

Specifies that the *subquery* cannot be updated.

WITH CHECK OPTION [CONSTRAINT constraint_name]

Tells Oracle to abort any changes to the deleted table that would not appear in the result set of the subquery. *[CONSTRAINT constraint_name]* tells Oracle to further restrict changes based upon a specific constraint identified by *constraint_name*.

TABLE (collection_expression) [(+)]

Informs Oracle that the *collection_expression* should be treated like a table even though it may, in fact, be a subquery, a function, or some other collection constructor. In any case, the value returned by the *collection_expression* must be a nested table or VARRAY.

hint

Instructs the database to use specific optimizer instructions other than those it might choose for itself; for example, to use or ignore a specific index. Refer to the vendor documentation for a full discussion of hints.

RETURNING expression

Retrieves the rows affected by the command (*DELETE* normally only shows the number of rows deleted). The *RETURNING* clause can be used when the target is a table, a materialized view, or a view with a single base table. When used for single-row deletes, the *RETURNING* clause stores values from the row deleted by the statement, defined by *expression*, into PL/SQL variables and bind variables. When used for a multirow delete, the *RETURNING* clause stores the values of the deleted rows, defined by *expression*, into bind arrays.

INTO variable

Specifies the variables into which the values returned as a result of the *RETURNING* clause are stored. There must be a corresponding *variable* for every *expression* in the *RETURNING* clause.

LOG ERRORS [INTO [schema.]table_name] [(simple_expression)] [REJECT LIMIT {UNLIMITED | int}]

Captures DML errors and log column values of affected rows, saving them in an error-logging table. Constraint violations will always cause the statement to fail and roll back, regardless of whether you specify a *LOG ERRORS* clause. The *LOG ERRORS* clause also cannot track errors in the error-logging table for columns with *LONG*, *LOG*, or object-type columns, though the table that is the target of the DML operation may have columns of these datatypes. The parameters of this clause are:

INTO [schema.]table_name

Specifies the error-logging table. When omitted, the default is **ERR\$_xxx**, where *xxx* is the first 25 characters of the name of the table from which the records are being deleted.

(simple_expression)

Specifies a value that tags each error in the logging table so that you can differentiate errors from many DML statements.

REJECT LIMIT {UNLIMITED | int}

Specifies an upper limit for the number of errors to be logged (*int*) before terminating the *DELETE* and rolling back any changes. The default is 0. The *UNLIMITED* keyword allows error logging with no upper limit.

When you execute a *DELETE* statement, Oracle releases space from the target table (or the base table of a target view) back to the table or index that owned the data.

When deleting from a view, the view cannot contain a set operator, the *DISTINCT* keyword, joins, an aggregate function, an analytic function, a subquery in the *SELECT* list, a collection expression in the *SELECT* list, a *GROUP BY* clause, an *ORDER BY* clause, a *CONNECT BY* clause, or a *START WITH* clause.

Here's an example where we delete records from a remote server:

```
DELETE FROM scott.sales@chicago;
```

In the following query, we delete from a derived table that is a collection expression:

```
DELETE TABLE(SELECT contactname FROM customers
  c WHERE c.customerid = 'BOTTM') s
 WHERE s.region IS NULL OR s.country = 'MEXICO';
```

Here's an example where we delete from a partition:

```
DELETE FROM sales PARTITION (sales_q3_1997)
 WHERE qty > 10000;
```

Finally, in the next example, we use the *RETURNING* clause to look at the values that are deleted:

```
DELETE FROM employee
 WHERE job_id = 13
   AND hire_date + TO_YMINTERVAL('01-06') <= SYSDATE;
 RETURNING job_lvl
 INTO :into1;
```

This example deletes records from the **employee** table and returns the **job_lvl** values into the predefined **:into1** variable.

PostgreSQL

PostgreSQL uses the *DELETE* command to remove rows and any defined subclasses from the table. Its implementation is otherwise identical to the ANSI standard. The syntax follows:

```
DELETE [FROM] [ONLY] [schema.]table_name
 [ USING usinglist ]
 [ WHERE search_condition | WHERE CURRENT OF cursor_name ]
 [ RETURNING { * | expression [AS alias][, ...] } ]
```

When deleting rows from only the table specified, use the optional *ONLY* clause. Otherwise, PostgreSQL will also delete records from any explicitly defined subtable. PostgreSQL supports two other important subclauses:

USING *usinglist*

Specifies a list of table expressions, thereby allowing columns from other tables to appear in the *WHERE* clause. This has the effect of specifying multiple tables in the *FROM* clause of a *SELECT* statement.

RETURNING { * | expression [AS alias][, ...] }

Specifies an expression to be returned by the *DELETE* statement after each row is deleted. The *expression* can return all columns (using the *** wildcard) or any columns you specify that are in *table_name* or in the *usinglist*.

To delete all records from the **titles** table:

```
DELETE titles
```

To delete all records in the **authors** table where the last name starts with "Mc":

```
DELETE FROM authors
 WHERE au_lname LIKE 'Mc%'
```

To delete all **titles** with an old ID number:

```
DELETE titles WHERE title_id >= 40
```

To delete all **titles** that have no sales:

```
DELETE titles WHERE ytd_sales IS NULL
```

To delete all records in one table based on the results of a subquery against another table (in this case, erasing from the **titleauthor** table the records that have a match concerning “computers” in the **titles** table):

```
DELETE FROM titleauthor
WHERE title_id IN
  (SELECT title_id
  FROM titles
  WHERE title LIKE '%computers%')
DISCONNECT
```

To delete all records in a table and return the full details of the deleted rows:

```
DELETE FROM titles WHERE ytd_sales IS NULL RETURNING *;
```

SQL Server

Microsoft SQL Server allows records to be deleted both from tables and from views that describe a single table. SQL Server also allows a second *FROM* clause to allow *JOIN* constructs, as in the following example:

```
[WITH cte_expression[, ...]]
DELETE [TOP ( number ) [PERCENT]] [FROM] table_name [[AS] alias]
[WITH ( hint [...] )]
[OUTPUT expression INTO {@table_variable | output_table} [ (column_list[, ..])
.]) ]
[FROM table_source[, ...]]
[ [{INNER | CROSS | [LEFT | RIGHT | FULL] OUTER}]
  JOIN joined_table ON condition[, ...] ]
[WHERE search_condition | WHERE CURRENT OF [GLOBAL] cursor_name]
[OPTION ( hint[, ...n] )]
```

The syntax elements are as follows:

WITH *cte_expression*

Defines the temporary named result set of a common table expression, derived from a *SELECT* statement, for the *DELETE* statement.

DELETE *table_name*

Allows the deletion of records from the named table or view filtered by the *WHERE* clause. You can delete records from a view, provided the view is based on one table and contains no aggregate functions and no derived columns. If you omit the server name, database name, or schema name when naming the table or view, SQL Server assumes the current context. An *OPENDATASOURCE* or *OPENQUERY* function, as described in the section on the *SELECT* statement, may be referenced instead of a table or view.

TOP (*number*) [PERCENT]

Indicates that the statement should delete only the specified *number* of rows. If *PERCENT* is specified, only the first *number* percent of the rows are retrieved. If *number* is an expression, such as a variable, it must be enclosed in parentheses. The expression should be of the *FLOAT* datatype with a range of 0 to 100 when using *PERCENT*. When not using *PERCENT*, the *number* should be of the *BIGINT* datatype.

WITH (hint)

Instructs the database to use specific optimizer instructions other than those it might choose for itself; for example, to use or ignore a specific index. The **WITH (hint)** clause specifies one or more table hints that are allowed for the target table. Refer to the vendor documentation for a full discussion of hints.

OUTPUT expression INTO {@table_variable | output_table} [(column_list[, ...])]

Retrieves the rows affected by the command, whereas **DELETE** normally only shows the number of rows deleted, placing the rows you specify in *expression* into either a given *table_variable* or *output_table*. If the *column_list* is omitted for the *output_table*, the *output_table* must have the same number of columns as the number of columns in the **OUTPUT** expression. The *output_table* cannot have triggers, participate in a foreign key, or have any **CHECK** constraints.

FROM table_source

Names an additional **FROM** clause that correlates records from the table in the first **FROM** clause using a **JOIN** rather than forcing you to use a correlated subquery. One or more tables may be listed in the second **FROM** clause.

[{INNER | CROSS | [LEFT | RIGHT | FULL] OUTER}] JOIN joined_table ON condition[, ...]

Specifies one or more **JOIN** clauses, in conjunction with the second **FROM** clause. You may use any of the join types that SQL Server supports. Refer to the section “The **JOIN** clause” within the discussion of the **SELECT** statement, later in this chapter, for more information.

GLOBAL cursor_name

Specifies that the delete operation should occur on the current row of an open global cursor. This clause is otherwise the same as the standard for **WHERE CURRENT OF**.

OPTION (hint[, ...])

Replaces elements of the default query plan with your own. Because the optimizer usually picks the best query plan for any query, we strongly discourage you from placing optimizer hints into your queries.

A significant extension to SQL Server’s implementation of the **DELETE** statement is the addition of a second **FROM** clause. The second **FROM** allows the use of the **JOIN** statement and makes it quite easy to delete rows from the table specified in the first **FROM** by correlating rows of a table declared in the second **FROM**. For example, you could use a rather complex subquery to erase all the **sales** records of computer books with this command:

```
DELETE sales
WHERE title_id IN
  (SELECT title_id
  FROM titles
  WHERE type = 'computer')
```

But SQL Server allows a more elegant construction using a second **FROM** clause and a **JOIN** clause:

```
DELETE s
FROM sales AS s
INNER JOIN titles AS t ON s.title_id = t.title_id
  AND type = 'computer'
```

The following example deletes all rows with an `order_date` of 2003 or earlier, in batches of 2,500:

```
WHILE 1 = 1
BEGIN
    DELETE TOP (2500)
    FROM sales_history WHERE order_date <= '20030101'
    IF @@rowcount < 2500 BREAK
END
```

The `TOP` clause should be used in favor of the old `SET ROWCOUNT` statement because the `TOP` clause is open to many more query-optimization algorithms. (Prior to SQL Server 2005, the `SET ROWCOUNT` statement was often used with data-modification transactions to process large numbers of rows in batches, thus preventing the transaction log from filling up and ensuring that row locks would not escalate to a full table lock.)

Common table expressions may be used with `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements as well as the `CREATE VIEW` statement. These expressions are a means of naming and defining a temporary result set from a `SELECT` statement, even allowing recursive behaviors. When defining a common table expression, you may not use `COMPUTE`, `COMPUTE BY`, `FOR XML`, `FOR BROWSE`, `INTO`, `OPTION`, or `ORDER BY` clauses. Multiple `SELECT` statements are allowed in a common table expression only if they are combined with set operators such as `UNION`, `UNION ALL`, `EXCEPT`, or `INTERSECT`. The following is a simple `DELETE` statement using a common table expression:

```
WITH direct_reports (Manager_ID, DirectReports) AS
    ( SELECT manager_ID, COUNT(*)
        FROM hr.employee AS e
        WHERE manager_id IS NOT NULL
        GROUP BY manager_id )
    DELETE FROM direct_reports
    WHERE DirectReports <= 1;
```

The `OUTPUT` clause allows you to see all of the rows that are being deleted:

```
DELETE TOP 10 error_log WITH (READPAST)
OUTPUT deleted.*
WHERE error_log_id = '28-OCT-2008';
```

See Also

`INSERT`
`SELECT`
`TRUNCATE TABLE`
`UPDATE`

DISCONNECT Statement

The `DISCONNECT` statement terminates one or more connections created between the current SQL process and the database server.

Platform	Command
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
DISCONNECT {CURRENT | ALL | connection_name | DEFAULT}
```

Keywords

CURRENT

Closes the currently active user connection.

ALL

Closes all open connections for the current user.

Rules at a Glance

DISCONNECT is used to disconnect a named SQL session (*connection_name*), the *CURRENT* connection, the *DEFAULT* connection, or *ALL* connections held by the user. For example, we can disconnect a single session called *new_york*:

```
DISCONNECT new_york
```

or disconnect all currently open sessions for the current user process:

```
DISCONNECT ALL
```

Programming Tips and Gotchas

DISCONNECT is not universally supported across platforms. Do not build cross-platform applications based on *DISCONNECT* unless you've made provisions to disconnect SQL sessions using each platform's preferred disconnection methodology.

MySQL

Not supported.

Oracle

Oracle allows *DISCONNECT* *only* in its ad hoc query tool, SQL*Plus, using this syntax:

```
DISC[ONNECT]
```

In this usage, the command ends the current session with the database server but otherwise allows work in SQL*Plus to continue. For example, a programmer can continue to edit the buffer, save run files, and so on, but must establish a new connection to issue any SQL commands. Exiting SQL*Plus and returning to the filesystem requires the *EXIT* or *QUIT* command. For example, to end the current connection with an Oracle server:

```
DISCONNECT;
```

You may achieve a similar effect to the SQL3 command by using the Oracle statement *ALTER SYSTEM DISCONNECT SESSION*.

PostgreSQL

Not supported. Instead, every programming interface supports a disconnect operation; for example, *SPI_FINISH* is available under the Server Programming Interface, and *PG_DISCONNECT* is available under the PL/TCL programming package.

SQL Server

Microsoft SQL Server supports the ANSI syntax for *DISCONNECT* in Embedded-SQL (ESQL) only, not within its ad hoc querying tool, SQL Server Management Studio. To disconnect cleanly from Microsoft SQL Server in an ESQL program, use the *DISCONNECT ALL* statement.

See Also

[CONNECT](#)

DROP Statements

All of the database objects created with *CREATE* statements may be destroyed using complementary *DROP* statements. On some platforms, a *ROLLBACK* statement after a *DROP* statement will recover the dropped object. However, on other database platforms the *DROP* statement is irreversible and permanent, so it is advisable to use the command with care.

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

Currently, the SQL2003 standard supports the ability to drop a lot of object types that are largely unsupported by most vendors. The ANSI SQL2003 syntax follows this format:

`DROP [object_type] object_name {RESTRICT | CASCADE}`

Keywords

`DROP [object_type] object_name`

Irreversibly and permanently destroys the object of type *object_type* called *object_name*. The *object_name* does not need a schema identifier, but if none is provided the current schema is assumed. ANSI SQL2003 supports a long list of object types, each created with its own corresponding *CREATE* statement. *CREATE* statements covered in this book with corresponding *DROP* statements include:

- *DOMAIN*
- *FUNCTION*
- *METHOD*

- PROCEDURE
- ROLE
- SCHEMA
- TABLE
- TRIGGER
- TYPE
- VIEW

RESTRICT | CASCADE

Prevents the *DROP* from taking place if any dependent objects exist (*RESTRICT*), or causes all dependent objects to also be dropped (*CASCADE*). This clause is not allowed with some forms of *DROP*, such as *DROP TRIGGER*, but is mandatory for others, such as *DROP SCHEMA*. To further explain, *DROP SCHEMA RESTRICT* will only drop an empty schema. Otherwise (i.e., if the schema contains objects), the operation will be prevented. In contrast, *DROP SCHEMA CASCADE* will drop a schema and all objects contained therein.

Rules at a Glance

For rules about the creation or modification of each of the object types, refer to the sections on the corresponding *CREATE/ALTER* statements.

The *DROP* statement destroys a pre-existing object. The object is permanently destroyed, and all users who had permission to access the object immediately lose the ability to access it.

The object may be qualified—that is, you may fully specify the schema where the dropped object is located. For example:

```
DROP TABLE scott.sales_2008 CASCADE;
```

This statement will drop not only the table `scott.sales_2004`, but also any views, triggers, or constraints built on it. On the other hand, a *DROP* statement may include an unqualified object name, in which case the current schema context is assumed. For example:

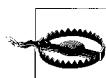
```
DROP TRIGGER before_ins_emp;
DROP ROLE sales_mgr;
```

Although not required by the SQL2003 standard, most implementations cause the *DROP* command to fail if the database object is in use by another user.

Programming Tips and Gotchas

DROP will only work when it is issued against a pre-existing object of the appropriate type and when the user has appropriate permissions (usually the *DROP TABLE* permission—refer to the section on the *GRANT* statement for more information). The SQL standard requires only that the owner of an object be able to drop it, but most database platforms allow variations on that requirement. For example, the database superuser/superadmin can usually drop any object on a database server.

With some vendors, the *DROP* command fails if the database object has extended properties. For example, Microsoft SQL Server will not drop a table that is replicated unless you first remove the table from replication.



It is important to be aware that most vendors do not notify you if the *DROP* command creates a dependency problem. Thus, if a table that is used by a few views and stored procedures elsewhere in the database is dropped, no warning is issued; those other objects simply fail when they are accessed. To prevent this problem, you may wish to use the *RESTRICT* syntax where it is available, or check for dependencies before invoking the *DROP* statement.

You may have noticed that the ANSI standard does not support certain common *DROP* commands, such as *DROP DATABASE* and *DROP INDEX*, even though every vendor covered in this book (and just about every one in the market) supports these commands. The exact syntax for each of these commands is covered in the platform-specific sections that follow.

MySQL

MySQL supports a very limited number of objects via the *CREATE/ALTER* statement. Consequently, the syntax for *DROP* is also limited:

```
DROP { {DATABASE | SCHEMA} | FUNCTION | INDEX [ONLINE | OFFLINE] |
       PROCEDURE | [TEMPORARY] TABLE | TRIGGER | VIEW }
      [IF EXISTS] object_name[, ...]
      [RESTRICT |CASCADE]
```

The supported SQL3 syntax elements are:

{DATABASE | SCHEMA} database_name

Drops the named database, including all the objects it contains (such as tables and indexes). *DROP SCHEMA* is a synonym for *DROP DATABASE* on MySQL. The *DROP DATABASE* command removes all database and table files from the filesystem, as well as two-digit subdirectories. MySQL will return a message showing how many files were erased from the database directory. (Files of these extensions are erased: *.BAK*, *.DAT*, *.FRM*, *.HSH*, *.ISD*, *.ISM*, *.MRG*, *.MYD*, *.MYI*, *.DM*, and *.FM*.) If the database is linked, both the link and the database are erased. You may drop only one database at a time. *RESTRICT* and *CASCADE* are not valid on *DROP DATABASE*.

FUNCTION routine_name

Drops the named routine from a MySQL v5.1 or greater database. You can use the *IF EXISTS* clause with a *DROP FUNCTION* statement.

INDEX [ONLINE | OFFLINE] index_name ON table_name

Drops the named index from the named table, in MySQL v3.22 or later. This command doesn't do anything in versions earlier than 3.22, but in 3.22 it actually executes an *ALTER TABLE...DROP INDEX* command behind the scenes. Beginning with version 5.1.22-ndb-6.2.5, you can drop indexes online using the eponymous keyword. Dropping an index online means that the specified table is not copied, while dropping the index offline creates a copy of the table before performing the action. You cannot use *RESTRICT*, *CASCADE*, or *IF EXISTS* with a *DROP INDEX* statement.

PROCEDURE routine_name

Drops the named routine from a MySQL v5.1 or greater database. You can use the *IF EXISTS* clause with a *DROP PROCEDURE* statement.

[TEMPORARY] TABLE *table_name*[, ...]

Drops one or more named tables, with table names separated from each other by commas. MySQL erases each table's definition and deletes the three table files (.FRM, .MYD, and .MYI) from the filesystem. Issuing this command causes MySQL to commit all active transactions. The *TEMPORARY* keyword drops only temporary tables without committing running transactions or checking access rights.

TRIGGER *schema_name.jtrigger_name*

Drops a named trigger for a MySQL v5.0.2 or greater database. You can use the *IF EXISTS* clause with a *DROP TRIGGER* statement to ensure that you only drop a trigger that actually exists within the database.

VIEW *view_name*

Drops a named view for the MySQL database. You can use the *IF EXISTS* clause with a *DROP VIEW* statement.

IF EXISTS

Prevents an error message when you attempt to drop an object that does not exist. Usable in MySQL v3.22 or later.

RESTRICT | CASCADE

Noise words. These keywords do not generate an error, nor do they have any other effect.

MySQL supports *only* the ability to drop a database, a table (or tables), or an index from a table. Although the *DROP* statement will not fail with the *RESTRICT* and *CASCADE* optional keywords, they have no effect. You can use the *IF EXISTS* clause to prevent MySQL from returning an error message if you try to delete an object that doesn't exist.

Other objects that MySQL allows you to drop using similar syntax include:

```
DROP { EVENT | FOREIGN KEY | LOGFILE GROUP | PREPARE | PRIMARY KEY |
       SERVER | TABLESPACE | USER }
       object_name
```

These variations of the *DROP* statement are beyond the scope of this book. Check the MySQL documentation for more details.

Oracle

Oracle supports most of the ANSI keywords for the *DROP* statements, as well as many additional keywords corresponding to objects uniquely supported by Oracle. Oracle supports the *DROP* statement for the following SQL3 objects:

```
DROP { DATABASE | FUNCTION | INDEX | PROCEDURE | ROLE | TABLE |
       TRIGGER | TYPE [BODY] | VIEW }
       object_name
```

The rules for Oracle *DROP* statements are less consistent than the ANSI standard's rules, so the full syntax of each *DROP* variant is shown in the following list:

DATABASE *database_name*

Drops the named database from the Oracle server.

FUNCTION *function_name*

Drops the named function, as long as it is not a component of a package. (If you want to drop a function from a package, use the *CREATE PACKAGE...OR REPLACE* statement to redefine the package without that function.) Any local objects that depend on or call the function are invalidated, and any statistical types associated with the function are disassociated.

INDEX *index_name* [FORCE]

Drops a named index or domain index from the database. Dropping an index invalidates all objects that depend on the parent table, including views, packages, functions, and stored procedures. Dropping an index also invalidates cursors and execution plans that use the index and will force a hard parse of the affected SQL statements when they are next executed.

Non-IOT indexes are secondary objects and can be dropped and recreated without any loss of user data. IOTs, because they combine both table and index data in the same structure, cannot be dropped and recreated in this manner. IOTs should be dropped using the *DROP TABLE* syntax.

When you drop a partitioned index, all partitions are dropped. When you drop a composite partitioned index, all index partitions and subpartitions are dropped. When you drop a domain index, any statistics associated with the domain index are removed and any statistic types are disassociated. The optional keyword *FORCE* applies only when dropping domain indexes. *FORCE* allows you to drop a domain index marked *IN PROGRESS*, or to drop a domain index when its *indextype* routine invocation returns an error. For example:

```
DROP INDEX ndx_sales_salesperson_quota;
```

PROCEDURE *procedure_name*

Drops the named stored procedure. Any dependent objects are invalidated when you drop a stored procedure, and attempts to access them before you recreate the stored procedure will fail with an error. If you recreate the stored procedure and then access a dependent object, the dependent object will be recompiled.

ROLE *role_name*

Drops the named role, removes it from the database, and revokes it from all users and roles to whom it has been granted. No new sessions can use the role, but sessions that are currently running under the role are not affected. For example, the following statement drops the *sales_mgr* role:

```
DROP ROLE sales_mgr;
```

TABLE *table_name* [CASCADE CONSTRAINTS] [PURGE]

Drops the named table, erases all of its data, drops all indexes and triggers built from the table (even those in other schemas), and invalidates all permissions and all dependent objects (views, stored procedures, etc.). On partitioned tables, Oracle drops all partitions (and subpartitions). On index-organized tables, Oracle drops all dependent mapping tables. Statistic types associated with a dropped table are disassociated. Materialized view logs built on a table are also dropped when the table is dropped.

The *DROP TABLE* statement is effective for standard tables, index-organized tables, and object tables. The table being dropped is only moved to the recycling bin, unless you add the optional keyword *PURGE*, which tells Oracle to immediate free all space consumed by the table. (Oracle also supports a non-ANSI SQL command called *PURGE* that lets you remove tables from the recycling bin outside of the *DROP TABLE* statement.) *DROP TABLE* erases only the metadata of an external table. You must use an external operating system command to drop the file associated with an external table and reclaim its space.

Use the optional *CASCADE CONSTRAINTS* clause to drop all referential integrity constraints elsewhere in the database that depend on the primary or unique key of the dropped table. You cannot drop a table with dependent referential integrity constraints without using the *CASCADE CONSTRAINTS* clause.

The following example drops the **job_desc** table in the **emp** schema, then drops the **job** table and all referential integrity constraints that depend on the primary key and unique key of the **job** table:

```
DROP TABLE emp.job_desc;
DROP TABLE job CASCADE CONSTRAINTS;
```

TRIGGER *trigger_name*

Drops the named trigger from the database.

TYPE [*BODY*] *type_name* [{**FORCE** | **VALIDATE**}]

Drops the specification and body of the named object type, nested table type, or **VARRAY**, as long as they have no type or table dependencies. You must use the optional **FORCE** keyword to drop a supertype, a type with an associated statistic type, or a type with any sort of dependencies. All subtypes and statistic types are then invalidated. Oracle will also drop any public synonyms associated with a dropped type. The optional *BODY* keyword tells Oracle to drop only the body of the type while keeping its specification intact. *BODY* cannot be used in conjunction with the **FORCE** or **VALIDATE** keywords. Use the optional **VALIDATE** keyword when dropping subtypes to check for stored instances of the named type in any of its supertypes. Oracle performs the drop only if no stored instances are found. For example:

```
DROP TYPE salesperson_type;
```

VIEW *view_name* [CASCADE CONSTRAINTS]

Drops the named view and marks as invalid any views, subviews, synonyms, or materialized views that refer to the dropped view. Use the optional clause **CASCADE CONSTRAINTS** to drop all referential integrity constraints that depend on the view. Otherwise, the *DROP* statement will fail if dependent referential integrity constraints exist. For example, the following statement drops the **active_employees** view in the **hr** schema:

```
DROP VIEW hr.active_employees;
```

In the *DROP* syntax, *object_name* can be replaced with *[schema_name.]object_name*. If you omit the schema name, the default schema of the user session is assumed. Thus, the following *DROP* statement drops the specified view from the **sales_archive** schema:

```
DROP VIEW sales_archive.sales_1994;
```

However, if your personal schema is **scott**, the following command is assumed to be against **scott.sales_1994**:

```
DROP VIEW sales_1994;
```

Oracle also supports the *DROP* statement for a large number of objects that aren't part of SQL3, including:

```
DROP { CLUSTER | CONTEXT | DATABASE LINK | DIMENSION | DIRECTORY | DISKGROUP |
FLASHBACK ARCHIVE | INDEXTYPE | JAVA | LIBRARY | MATERIALIZED VIEW |
MATERIALIZED VIEW LOG | OPERATOR | OUTLINE | PACKAGE | PROFILE | RESTORE
POINT |
ROLLBACK SEGMENT | SEQUENCE | SYNONYM | TABLESPACE | TYPE BODY | USER }
object_name
```

These variations are beyond the scope of this book. Refer to the Oracle documentation if you want to drop an object of one of these types (although the basic syntax is the same for almost all variations of the *DROP* statement).

PostgreSQL

PostgreSQL does not support the *RESTRICT* or *CASCADE* optional keywords supported by the ANSI standard. It does support a wide variety of *DROP* variants, as follows:

```
DROP { DATABASE | DOMAIN | FUNCTION | INDEX | ROLE |
       SCHEMA | TABLE | TRIGGER | TYPE | VIEW }
      [IF EXISTS]
      object_name
      [ CASCADE | RESTRICT ]
```

Following is the full SQL3-supported syntax for each variant:

DATABASE *database_name*

Drops the named database and erases the operating system directory containing all of the database's data. This command can only be executed by the database owner, while that user is connected to a database other than the target database. For example, we can drop the **sales_archive** database:

```
DROP DATABASE sales_archive;
```

DOMAIN *domain_name*[, ...] [*CASCADE* | *RESTRICT*]

Drops one or more named domains owned by the session user. *CASCADE* automatically drops objects that depend on the domain, while *RESTRICT* prevents the action from occurring if any objects depend on the domain. When omitted, *RESTRICT* is the default behavior.

FUNCTION *function_name* ([*datatype1*, ...]) [*CASCADE* | *RESTRICT*]

Drops the named user-defined function. Since PostgreSQL allows multiple functions of the same name, distinguished only by the various input parameters they require, you must specify one or more datatypes to uniquely identify the user-defined function you wish to drop. PostgreSQL does not perform any kind of dependency checks on other objects that might reference a dropped user-defined function. (They will fail when invoked against an object that no longer exists.) For example:

```
DROP FUNCTION median_distribution (int, int, int, int);
```

INDEX *index_name*[, ...] [*CASCADE* | *RESTRICT*]

Drops one or more named indexes that you own. For example:

```
DROP INDEX ndx_titles, ndx_authors;
```

ROLE *rule_name*[, ...]

Drops one or more named database roles. On PostgreSQL, a role cannot be dropped when it is referenced in any database. That means you'll need to drop or reassign ownership of any objects owned by the role, using *REASSIGN OWNED* and *DROP OWNED* statements, before dropping it, and then revoke any privileges the role has been granted.

SCHEMA *schema_name*[, ...] [*CASCADE* | *RESTRICT*]

Drops one or more named schemas from the current database. A schema can only be dropped by a superuser or the owner of the schema (even when the owner does not explicitly own all of the objects in the schema).

TABLE *table_name*[, ...] [*CASCADE* | *RESTRICT*]

Drops one or more existing tables from the database, as well as any indexes or triggers specified for the tables. For example:

```
DROP TABLE authors, titles;
```

TRIGGER *trigger_name* ON *table_name* [CASCADE | RESTRICT]

Drops the named trigger from the database. You must specify the *table_name* because PostgreSQL requires that trigger names be unique only on the tables to which they are attached. This means it is possible to have many triggers called, say, **insert_trigger** or **delete_trigger**, each on a different table. For example:

```
DROP TRIGGER insert_trigger ON authors;
```

TYPE *type_name*[, ...] [CASCADE | RESTRICT]

Drops one or more pre-existing user-defined types from the database. PostgreSQL does not check to see what impact the **DROP TYPE** command might have on any dependent objects, such as functions, aggregates, or tables; you must check the dependent objects manually. (Do *not* remove any of the built-in types that ship with PostgreSQL!) Note that PostgreSQL's implementation of types differs from the ANSI standard. Refer to the section on the **CREATE/ALTER TYPE** statement for more information.

VIEW *view_name*[, ...] [CASCADE | RESTRICT]

Drops one or more pre-existing views from the database.

CASCADE | RESTRICT

CASCADE automatically drops objects that depend on the object being dropped, while **RESTRICT** prevents the action from occurring if any objects depend on the object being dropped. When omitted, **RESTRICT** is the default behavior. Since this subclause is not allowed with all forms of **DROP** in PostgreSQL, it is shown with the statements where it is allowed.

IF EXISTS

Suspends the creation of an error message of the object to be dropped does not exist. This subclause is usable for all variations of the **DROP** statement.

Note that PostgreSQL drop operations do not allow you to specify the target database where the operation will take place (except for **DROP DATABASE**). Therefore, you should execute any drop operation from the database where the object you want to drop is located.

PostgreSQL supports variations of the **DROP** statement for several objects that are extensions to the SQL3 standard, as shown here:

```
DROP { AGGREGATE | CAST | CONVERSION | GROUP | LANGUAGE | OPERATOR [CLASS] |  
RULE | SEQUENCE | TABLESPACE | USER } object_name
```

These variations are beyond the scope of this book. Refer to the PostgreSQL documentation if you want to drop an object of one of these types (although the basic syntax is the same for almost all variations of the **DROP** statement).

SQL Server

SQL Server supports several SQL3 variants of the **DROP** statement:

```
DROP { DATABASE | FUNCTION | INDEX | PROCEDURE | ROLE |  
SCHEMA | TABLE | TRIGGER | TYPE | VIEW } object_name
```

Following is the full syntax for each variant:

DATABASE *database_name*[, ...]

Drops the named database(s) and erases all disk files used by the database(s). This command may only be issued from the **master** database. Replicated databases must be removed from their replication schemes before they can be dropped, as must log shipping databases. You cannot drop a database while it is in use, nor can you drop system databases (**master**, **model**, **msdb**, or **tempdb**).

For example, we can drop the **northwind** and **pubs** databases with one command:

```
DROP DATABASE northwind, pubs
GO
```

FUNCTION [schema.]function_name[, ...]

Drops one or more user-defined functions from the current database.

INDEX index_name ON table_or_view_name[, ...] [WITH { MAXDOP = int | ONLINE = {ON | OFF} | MOVE TO location [FILESTREAM_ON location] }]

Drops one or more indexes from tables or indexed views in the current database and returns the freed space to the database. This statement should not be used to drop a **PRIMARY KEY** or **UNIQUE** constraint. Instead, drop these constraints using the **ALTER TABLE...DROP CONSTRAINT** statement. When dropping a clustered index from a table, all non-clustered indexes are rebuilt. When dropping a clustered index from a view, all non-clustered indexes are dropped. The **WITH** subclause may only be used when dropping a clustered index. **MAXDOP** specifies the maximum degrees of parallelism that SQL Server may use to drop the clustered index. Values for **MAXDOP** may be 1 (suppresses parallelism), 0 (the default, using all or fewer processors on the system), or a value greater than 1 (restricts parallelism to the value of *int*). **ONLINE** specifies that queries or updates may continue on the underlying tables (with **ON**), or that table locks are applied and the table is unavailable for the duration of the process (with **OFF**). **MOVE TO** specifies a pre-existing filegroup or partition, or the default location for data within the database to which the clustered index will be moved. The clustered index is moved to the new location in the form of a heap.

PROC[EDURE] procedure_name[, ...]

Drops one or more stored procedures from the current database. SQL Server allows multiple versions of a single procedure via version numbers, but these versions cannot be dropped individually; you must drop all versions of a stored procedure at once. System procedures (those with an **sp_** prefix) are dropped from the **master** database if they are not found in the current user database. For example:

```
DROP PROCEDURE calc_sales_quota
GO
```

ROLE rule_name[, ...]

Drops one or more roles from the current database. The role must not own any objects, or else the statement will fail. You must first drop owned objects or change their ownership before dropping a role that owns any objects.

SCHEMA schema_name

Drops a schema that does not own any objects. To drop a schema that owns objects, first drop the dependent objects or assign them to a different schema.

TABLE [database_name.][schema_name.]table_name[, ...]

Drops a named table and all data, permissions, indexes, triggers, and constraints specific to that table. (The **table_name** may be a fully qualified table name like **pubs.dbo.sales** or a simple table name like **sales**, if the current database and owner are correct.) Views, functions, and stored procedures that reference the table are not dropped or marked as invalid, but will return an error when their procedural code encounters the missing table. Be sure to drop these yourself! You cannot drop a table referenced by a **FOREIGN KEY** constraint without first dropping the constraint. Similarly, you cannot drop a table used in replication without

first removing it from the replication scheme. Any user-defined rules or defaults are unbound when the table is dropped. They must be rebound if the table is recreated.

TRIGGER *trigger_name*[, ...] [ON {DATABASE | ALL SERVER}]

Drops one or more triggers from the current database. The subclause **[ON {DATABASE | ALL SERVER}]** is available when dropping DDL triggers, while the subclause **[ON ALL SERVER]** is also available to *LOGON* event triggers. **ON DATABASE** indicates the scope of the *DDL* trigger applied to the current database and is required if the subclause was used when the trigger was created. **ON ALL SERVER** indicates the scope of the *DDL* or *LOGON* trigger applied to the current server and is required if the subclause was used when the trigger was created.

TYPE [*schema_name*.]*type_name*[, ...]

Drops one or more user-defined types from the current database.

VIEW [*schema_name*.]*view_name*[, ...]

Drops one or more views from the database, including indexed views, and returns all space to the database.

SQL Server also has a large number of objects that extend the ANSI standard and that are removed using the more-or-less standardized syntax of the *DROP* statement. These variations of the syntax include:

```
DROP { AGGREGATE | APPLICATION ROLE | ASSEMBLY | ASYMMETRIC KEY | BROKER
  PRIORITY |
  CERTIFICATE | CONTRACT | CREDENTIAL | CRYPTOGRAPHIC PROVIDER | DATABASE
  AUDIT
  SPECIFICATION | DATABASE ENCRYPTION KEY | DEFAULT | ENDPOINT | EVENT
  NOTIFICATION | EVENT SESSION | FULLTEXT CATALOG | FULLTEXT INDEX | FULLTEXT
  STOPLIST | LOGIN | MASTER KEY | MESSAGE TYPE | PARTITION FUNCTION |
  PARTITION
  SCHEME | QUEUE | REMOTE SERVICE BINDING | RESOURCE POOL | ROUTE | SERVER
  AUDIT |
  SERVER AUDIT SPECIFICATION | SERVICE | SIGNATURE | STATISTICS | SYMMETRIC
  KEY |
  SYNONYM | USER | WORKLOAD GROUP | XML SCHEMA COLLECTION }
object_name
```

These variations are beyond the scope of this book. Refer to the SQL Server documentation to drop an object of one of these types (although the basic syntax is the same for almost all variations of the *DROP* statement).

See Also

[CALL](#)
[CONSTRAINTS](#)
[CREATE/ALTER FUNCTION/PROCEDURE/METHOD](#)
[CREATE SCHEMA](#)
[CREATE/ALTER TABLE](#)
[CREATE/ALTER VIEW](#)
[DELETE](#)
[DROP](#)
[GRANT](#)
[INSERT](#)
[RETURN](#)

EXCEPT Set Operator

SELECT
SUBQUERY
UPDATE

EXCEPT Set Operator

The *EXCEPT* set operator retrieves the result sets of two or more queries, including all the records retrieved by the first query that are not also found in subsequent queries. Whereas *JOIN* clauses are used to return the rows of two or more queries that are in common, *EXCEPT* is used to filter out the records that are present in only one of multiple, but similar, tables.

EXCEPT is in a class of keywords called *set operators*. Other set operators include *INTERSECT* and *UNION*. (*MINUS* is Oracle's equivalent to the *EXCEPT* keyword; *EXCEPT* is the ANSI standard.) All set operators are used to simultaneously manipulate the result sets of two or more queries, hence the term “set operators.”

Platform	Command
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

There are technically no limits to the number of queries that you may combine with the *EXCEPT* operator. The general syntax is:

```
{SELECT statement1 | VALUES (expr1[, ...])}
EXCEPT [ALL | DISTINCT]
[CORRESPONDING [BY (column1, column2, ...)]]
{SELECT statement2 | VALUES (expr2[, ...])}
EXCEPT [ALL | DISTINCT]
[CORRESPONDING [BY (column1, column2, ...)]]
...
...
```

Keywords

VALUES (expr1[, ...])

Generates a derived result set with explicitly declared values as *expr1*, *expr2*, etc. It is essentially a *SELECT* statement result set without the *SELECT...FROM* syntax. This is known as a *row constructor*, since the rows of the result set are manually constructed. According to the ANSI standard, multiple hand-coded rows in a row constructor must be enclosed in parentheses and separated by commas.

EXCEPT

Determines which rows will be excluded from the single result set.

ALL | DISTINCT

ALL considers duplicate rows from all result sets. *DISTINCT* drops duplicate rows from all result sets prior to the *EXCEPT* comparison. Any columns containing a *NULL* value are considered duplicates. (If neither *ALL* nor *DISTINCT* is used, *DISTINCT* behavior is the default.)

CORRESPONDING [BY (column1, column2, ...)]

Specifies that only columns with the listed names are returned, even if one or both queries use the asterisk shortcut.

Rules at a Glance

There is only one significant rule to remember when using *EXCEPT*: the number and order of the columns should be the same in all queries, and the datatypes should be of the same category.

The datatypes do not have to be identical, but they must be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest datatype size of each column in each ordinal position. For example, a query retrieving rows from *VARCHAR(10)* and *VARCHAR(15)* columns will use the *VARCHAR(15)* datatype and size.

Programming Tips and Gotchas

None of the platforms supports the *CORRESPONDING [BY (column1, column2, ...)]* clause.



On platforms that do not support *EXCEPT*, you might substitute a *NOT IN* subquery. However, *NOT IN* subqueries have different NULL handling and, on some database platforms, produce different result sets.

According to the ANSI standard, the *UNION* and *EXCEPT* set operators evaluate with equal precedence. However, the *INTERSECT* set operator evaluates before the other set operators. We recommend that you explicitly control the precedence of the set operators using parentheses as a general best practice.

According to the ANSI standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. To avoid column and table ambiguity, be sure to alias each column for each table with the same respective alias. For example:

```
SELECT au_lname AS 'lastname', au_fname AS 'firstname'
  FROM authors
EXCEPT
  SELECT emp_lname AS 'lastname', emp_fname AS 'firstname'
  FROM employees
  ORDER BY lastname, firstname
```

Also, while each of your column lists may list columns with correspondingly compatible datatypes, there may be some variation in behavior across the DBMS platforms with regard to the length of the columns. For example, if the *au_lname* column in the previous example's first query is markedly longer than the *emp_lname* column in the second query, the platforms may apply different rules as to which length is used for the final result. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Note that you can use *NOT IN* or *NOT EXISTS* operations in conjunction with a correlated subquery as alternatives. The following queries are examples of how you can achieve *EXCEPT* functionality using *NOT EXISTS* and *NOT IN*:

```
SELECT DISTINCT a.city
  FROM pubs..authors AS a
```

```

WHERE NOT EXISTS
  (SELECT *
    FROM pubs..publishers AS p
    WHERE a.city = p.city)
SELECT DISTINCT a.city
  FROM pubs..authors AS a
 WHERE a.city NOT IN
  (SELECT p.city
    FROM pubs..publishers AS p
    WHERE p.city IS NOT NULL)

```

In general, *NOT EXISTS* is faster than *NOT IN*. In addition, there is a subtle issue with NULLs that differentiates the *IN* and *NOT IN* operators and the *EXISTS* and *NOT EXISTS* set operators. To get around this different handling of NULLs, simply add the *IS NOT NULL* clause to the *WHERE* clause, as shown in the preceding example.

Each DBMS may apply its own rules as to which column name is used if the names vary across column lists. In general, the column names of the first query are used.

MySQL

EXCEPT is not supported in MySQL. However, you can use the *NOT IN* or *NOT EXISTS* operations as alternatives, as detailed in the previous section.

Oracle

Oracle does not support the *EXCEPT* set operator. However, it has an alternative set operator, *MINUS*, with identical functionality:

```

<SELECT statement1>
MINUS
<SELECT statement2>
MINUS
...

```

MINUS DISTINCT and *MINUS ALL* are not supported. *MINUS* is the functional equivalent of *MINUS DISTINCT*. Oracle does not support *MINUS* on queries under the following circumstances:

- Queries containing columns whose datatypes are *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY*
- Queries containing a *FOR UPDATE* clause
- Queries containing *TABLE* collection expressions

If the first query in a set operation contains any expressions in the select item list, you must include *AS* clauses to associate aliases with those expressions. Also, only the last query in the set operation may contain an *ORDER BY* clause.

For example, you could generate a list of all store IDs that do not have any records in the **sales** table as follows:

```

SELECT stor_id FROM stores
MINUS
SELECT stor_id FROM sales

```

The *MINUS* command is functionally similar to a *NOT IN* query. This query retrieves the same results:

```
SELECT stor_id FROM stores
WHERE stor_id NOT IN
  (SELECT stor_id FROM sales)
```

PostgreSQL

PostgreSQL supports the *EXCEPT* and *EXCEPT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
EXCEPT [ALL]
<SELECT statement2>
EXCEPT [ALL]
...
...
```

PostgreSQL does not support *EXCEPT* or *EXCEPT ALL* on queries with a *FOR UPDATE* clause. *EXCEPT DISTINCT* is not supported, but *EXCEPT* is the functional equivalent. PostgreSQL also does not support the *CORRESPONDING* clause.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *EXCEPT* or *EXCEPT ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the right-most occurrence of *ORDER BY* or *LIMIT* will be applied to the entire set operation.

PostgreSQL evaluates *SELECT* statements in a multi-*EXCEPT* statement from top to bottom, unless you use parentheses to change the evaluation hierarchy of the statements.

Normally, duplicate rows are eliminated from the two result sets, unless you add the *ALL* keyword. For example, you could find all titles in the **authors** table that have no records in the **sales** table using this query:

```
SELECT title_id
  FROM authors
EXCEPT ALL
SELECT title_id
  FROM sales;
```

SQL Server

EXCEPT is supported, though the SQL3 subclauses *CORRESPONDING*, *ALL*, and *DISTINCT* are not. For comparison purposes, SQL Server considers NULL values equal when evaluating an *EXCEPT* result set. If using the *SELECT...INTO* statement, only the first query may contain the *INTO* clause. *ORDER BY* is only allowed at the end of the statement and is not allowed with each individual query. Conversely, *GROUP BY* and *HAVING* clauses can only be used within individual queries and may not be used to affect the final result set. The *FOR BROWSE* clause may not be used with statements that include *EXCEPT*.

See Also

[INTERSECT](#)
[SELECT](#)
[UNION](#)

EXISTS Operator

The *EXISTS* operator tests a subquery for the existence of rows.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
SELECT ...
WHERE [NOT] EXISTS (subquery)
```

The parameters and keywords are as follows:

WHERE [NOT] EXISTS

Tests the subquery for the existence of one or more rows. If even one row satisfies the subquery clause, it returns a Boolean *TRUE* value. The optional *NOT* keyword returns a Boolean *TRUE* value when the subquery returns no matching rows.

subquery

Retrieves a result set based on a fully formed subquery.

Rules at a Glance

The *EXISTS* operator checks a subquery for the existence of one or more records against the records in the parent query.

For example, if we want to see whether there any jobs where no employee is filling the position:

```
SELECT *
FROM jobs
WHERE NOT EXISTS
  (SELECT * FROM employee
   WHERE jobs.job_id = employee.job_id)
```

This example tests for the absence of records in the subquery using the optional *NOT* keyword. The next example looks for specific records in the subquery to retrieve the main result set:

```
SELECT au_lname
FROM authors
WHERE EXISTS
  (SELECT *
   FROM publishers
   WHERE authors.city = publishers.city)
```

This query returns the last names of authors who live in the same city as their publishers. Note that the asterisk in the subquery is acceptable, since the subquery only needs to return a single record to provide a Boolean *TRUE* value. Columns are irrelevant in these cases. The first example selects only a single column; the key point is whether a *row* exists.

Programming Tips and Gotchas

EXISTS, in many queries, does the same thing as *ANY* (in fact, it is syntactically equivalent to the *ANY* operator). *EXISTS* is usually most effective with correlated subqueries.

The *EXISTS* subquery usually searches for only one of two things. Your first option is to use the asterisk wildcard (e.g., *SELECT * FROM...*) so that you are not retrieving any specific column or value. In this case, the asterisk means “any column.” The second option is to select only a single column in the subquery (e.g., *SELECT au_id FROM...*). Some individual database platforms also allow a subquery against more than one column (e.g., *SELECT au_id, au_lname FROM...*). However, this feature is rare and should be avoided in code that needs to be transportable across platforms.

Platform Differences

All of the platforms support *EXISTS* in the manner described above.

See Also

ALL/ANY/SOME

SELECT

WHERE

FETCH Statement

The *FETCH* statement is one of four commands used in cursor processing, along with *DECLARE*, *OPEN*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than as a complete set. *FETCH* positions a cursor on a specific row and retrieves that row from the result set.

Cursors are especially important in relational databases because they are set-based, while most client-centric programming languages are row-based. Cursors allow you to perform operations a single row at a time, to better fit what a client program can do, rather than operating on a whole set of records at once.

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported, with limitations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
FETCH [ { NEXT | PRIOR | FIRST | LAST |
          { ABSOLUTE int | RELATIVE int } }
        FROM ] cursor_name
        [INTO variable1[, ...]]
```

Keywords

NEXT

Tells the cursor to return the record immediately following the current row, and increments the current row to the row returned. *FETCH NEXT* is the default behavior for *FETCH*. It retrieves the first record if *FETCH* is performed as the first fetch against a cursor.

FETCH Statement

PRIOR

Tells the cursor to return the record immediately preceding the current row and decrements the current row to the row returned. *FETCH PRIOR* does not retrieve a record if it is performed as the first fetch against the cursor.

FIRST

Tells the cursor to return the first record in the cursor result set, making it the current row.

LAST

Tells the cursor to return the last record in the cursor result set, making it the current row.

ABSOLUTE *int*

Tells the cursor to return the *int* record from the cursor record set counting from the top (if *int* is a positive integer), or the *int* record counting from the bottom (if *int* is a negative integer), making the returned record the new current record of the cursor. If *int* is 0, no rows are returned. If the value of *int* moves the cursor past the end of the cursor result set, the cursor is positioned after the last row (for a positive *int*) or before the first row (for a negative *int*).

RELATIVE *int*

Tells the cursor to return the record *int* rows after the current record (if *int* is positive) or *int* rows before the current record (if *int* is negative), making the returned record the new current row of the cursor. If *int* is 0, the current row is returned. If the value of *int* moves the cursor past the end of the cursor result set, the cursor is positioned after the last row (for a positive *int*) or before the first row (for a negative *int*).

[FROM] *cursor_name*

Gives the name of the (open) cursor from which you want to retrieve rows. The cursor must be previously created and instantiated using the *DECLARE* and *OPEN* clauses. *FROM* is optional, but encouraged.

INTO *variable1*[, ...]

Stores data from each column in the open cursor into a local variable. Each column in the cursor must have a corresponding variable of a matching datatype in the *INTO* clause. Each column value is directly related to the variables in ordinal positions.

Rules at a Glance

At the highest level, a cursor must be:

1. Created using *DECLARE*
2. Opened using *OPEN*
3. Operated against using *FETCH*
4. Dismissed using *CLOSE*

By following these steps, you create a result set similar to that of a *SELECT* statement, except that you can operate against each individual row within the result set separately.



Each database platform has its own rules about how variables are used. For example, SQL Server requires an at sign (@) as a prefix, PostgreSQL and Oracle have no prefix, and so forth. The SQL standard says that a colon (:) prefix is necessary for languages that are embedded, like C or COBOL, but no prefix is needed for procedural SQL.

A cursor rests either directly on a row, before the first row, or after the last row. When a cursor is resting directly in a row, that row is known as the *current row*. You can use cursors to position an *UPDATE* or *DELETE* statement in the current row using the *WHERE CURRENT OF* clause.

It is important to remember that a cursor result set does not wrap around. Thus, if you have a result set containing 10 records and you tell the cursor to move forward 12 records, it will not wrap back around to the beginning and continue its count from there. Instead, the default cursor behavior is to stop after the last record in the result set when scrolling forward, and to stop before the first record when scrolling back. For example, on SQL Server:

```
FETCH RELATIVE 12 FROM employee_cursor
INTO @emp_last_name, @emp_first_name, @emp_id
```

When fetching values from the database into variables, make sure that the datatypes and number of columns and variables match. Otherwise, you'll get an error. For example, this will fail since the **employee_cursor** contains three columns while the fetch operation has only two variables:

```
FETCH PRIOR FROM employee_cursor
INTO @emp_last_name, @emp_id
```

Programming Tips and Gotchas

The most common errors encountered with the *FETCH* statement are mismatches between the number, order, or datatypes of the variables and the values in the cursor. So, before you write your *FETCH* statements, make sure you know exactly what values are in the cursor and what their datatypes are.

Typically, the database will lock at least the current row and possibly all rows held by the cursor. According to the ANSI standard, cursor locks are not held through *ROLLBACK* or *COMMIT* operations, although this behavior varies from platform to platform.

Although the *FETCH* statement is detailed in isolation here, it should always be managed as a group with the *DECLARE*, *OPEN*, and *CLOSE* statements. For example, every time you open a cursor, the server consumes memory. If you forget to close your cursors, you could create memory-management problems. So, you need to make sure that every declared and opened cursor is eventually closed.

Cursors are also often used in stored procedures or batches of procedural code. The reason for this is because sometimes you need to perform actions on individual rows rather than on entire sets of data at a time. But because cursors operate on individual rows and not sets of data, they are often much slower than other means of accessing your data. It's important to analyze your approach. Many challenges, such as a convoluted *DELETE* operation or a very complex *UPDATE*, can be solved by using clever *WHERE* and *JOIN* clauses instead of cursors.

MySQL

MySQL supports the basics of the SQL3 standard *FETCH* statement:

```
FETCH cursor_name INTO variable_name1[, ...]
```

MySQL will fetch the next row (if one exists) using the specific open cursor and advance the cursor pointer one increment. When no more rows are available, MySQL returns a *SQLSTATE* value of '02000' and a *NO DATA* condition event. You can detect and treat this occurrence using a handler for the *SQLSTATE* value or for a *NOT FOUND* condition.

Oracle

Oracle cursors are implicitly forward-only cursors that always scroll forward one record at a time. An Oracle cursor, when compared to the ANSI standard, is essentially a *FETCH NEXT 1* cursor. Oracle cursors must either insert the retrieved values into matching variables, or use the *BULK COLLECT* clause to insert all of the records of the result set into an array. Oracle does not support keywords like *PRIOR*, *ABSOLUTE*, and *RELATIVE*. However, Oracle does support both forward-only and scrollable cursors in the database via the Oracle Call Interface (OCI). OCI also supports features like *PRIOR*, *ABSOLUTE*, and *RELATIVE* for read-only cursors whose result sets are based on read-consistent snapshots.

Oracle's *FETCH* syntax is:

```
FETCH cursor_name
{ INTO variable_name1[, ...] | BULK COLLECT INTO
  collection_name[, ...] [LIMIT int] }
```

where:

BULK COLLECT INTO collection_name

Retrieves the entire rowset, or a specified number of rows (see *LIMIT*), into a client-side array or collection variable named *collection_name*.

LIMIT int

Limits the number of records fetched from the database when using the *BULK* statement. The *int* value is a nonzero integer (or a variable representing an integer value).

Oracle supports dynamic SQL-based cursors whose text can be built at runtime. Oracle also supports *cursor_names* that may be any allowable variable, parameter, or host-array type. In fact, you may also use user-defined or *%ROWTYPE* record variables for the *INTO* clause. This allows you to construct flexible, dynamic cursors in a template-like structure.

For example:

```
DECLARE
  TYPE namelist IS TABLE OF employee.lname%TYPE;
  names namelist;
  CURSOR employee_cursor IS SELECT lname FROM employee;
BEGIN
  OPEN employee_cursor;
  FETCH employee_cursor BULK COLLECT INTO names;
  ...
  CLOSE employee_cursor;
END;
```

Oracle *FETCH* often is paired with a PL/SQL *FOR* loop (or other kind of loop) to cycle through all the rows in the cursor. You should use the cursor attributes *%FOUND* or *%NOTFOUND* to detect the end of the rowset. For example:

```

DECLARE
    TYPE employee_cursor IS REF CURSOR RETURN employee%ROWTYPE;
    employee_cursor  EmpCurTyp;
    employee_rec     employee%ROWTYPE;
BEGIN
    LOOP
        FETCH employee_cursor INTO employee_rec;
        EXIT WHEN employee_cursor%NOTFOUND;
        ...
    END LOOP;
    CLOSE employee_cursor;
END;

```

This example uses a standard PL/SQL loop with an *EXIT* clause to end the loop when there are no more rows in the cursor to process.

PostgreSQL

PostgreSQL supports both forward- and backward-scrolling cursors with a superset of modes compared to the SQL3 standard. The syntax for *FETCH* in PostgreSQL is:

```

FETCH { FORWARD [ {ALL | int} ] | BACKWARD [ {ALL | int} ] |
        ABSOLUTE int | RELATIVE int | int | ALL | NEXT | PRIOR | FIRST | LAST }
{ IN | FROM } cursor_name
[ INTO :variable1[, ...] ]

```

where:

FORWARD [{ALL | int}]

Tells PostgreSQL to fetch the next row (same as *NEXT*), if there are no other keywords. This is the default if no other mode is defined. *FORWARD ALL* fetches all remaining rows in the cursor and positions the cursor after the last remaining row, while *FORWARD int* returns all rows up to *int* rows forward (or the current row if *int* is 0) and places the cursor after the last row fetched.

BACKWARD [{ALL | int}]

Fetches the prior row, if no other keywords are specified. *BACKWARD ALL* fetches all prior rows in the cursor scanning backward and positions the cursor before the first row, while *BACKWARD int* returns all rows up to *int* rows back (or the current row if *int* is 0) and places the cursor before the last row fetched.

ABSOLUTE int

Fetches the row occurring at position *int*.

RELATIVE int

Fetches the *int* next rows or *int* prior rows (if negative).

int

A signed integer that indicates how many records to scroll forward (if positive) or backward (if negative).

ALL

Retrieves all records remaining in the cursor.

NEXT

Retrieves the next single record.

PRIOR

Retrieves the previous single record.

IN | FROM *cursor_name*

Defines the previously declared and opened cursor from which data will be retrieved.

INTO *variable*

Assigns a cursor value to a specific variable. As with ANSI cursors, the values retrieved by the cursor and the variables must match in number, datatype, and order.

PostgreSQL cursors must be used with transactions explicitly declared using *BEGIN* and must be closed with *COMMIT* or *ROLLBACK*.

The following PostgreSQL statement retrieves five records from the **employee** table and displays them:

```
FETCH FORWARD 5 IN employee_cursor;
```

PostgreSQL also supports a separate command, *MOVE*, to move to a specific cursor position. It differs from *FETCH* only by not returning values into variables:

```
MOVE { [ FORWARD | BACKWARD | ABSOLUTE | RELATIVE ] }
      { [int | ALL | NEXT | PRIOR ] }
{ IN | FROM } cursor_name
```

For example, the following code declares a cursor, skips forward five records, and then returns the values in the sixth record:

```
BEGIN WORK;
DECLARE employee_cursor CURSOR FOR SELECT * FROM employee;
MOVE FORWARD 5 IN employee_cursor;
FETCH 1 IN employee_cursor;
CLOSE employee_cursor;
COMMIT WORK;
```

The preceding code will return a single row from the **employee_cursor** result set.

SQL Server

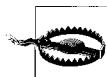
SQL Server supports a variation of *FETCH* that is very close to the ANSI standard:

```
FETCH [ { NEXT | PRIOR | FIRST | LAST |
          { ABSOLUTE int | RELATIVE int } } ]
      [FROM] [GLOBAL] cursor_name
      [INTO @variable1[, ...]]
```

The differences between SQL Server's implementation and the ANSI standard are very small. First, SQL Server allows you to use variables in place of the *int* and *cursor_name* values. Second, SQL Server allows the declaration and use of *GLOBAL* cursors that can be accessed by any user or session, not just the one that created it.

There are some rules that apply to how you use the *FETCH* command based upon how you issued the *DECLARE CURSOR* command:

- When you declare a *SCROLL* SQL-92 cursor, all *FETCH* options are supported. In all other SQL-92 cursors, *NEXT* is the only option supported. (There is also an alternate Transact-SQL-style *DECLARE CURSOR* statement.)
- *DYNAMIC SCROLL* cursors support all *FETCH* options except *ABSOLUTE*.
- *FORWARD_ONLY* and *FAST_FORWARD* cursors support only *FETCH NEXT*.
- *KEYSET* and *STATIC* cursors support all *FETCH* options.



SQL Server also requires a *DEALLOCATE* statement, in addition to *CLOSE*, to release memory consumed by a cursor.

Here's a full example that goes from declaring and opening a cursor, to initiating several fetches, and then finally closing and deallocating the cursor:

```

DECLARE @vc_lname VARCHAR(30), @vc_fname VARCHAR(30), @i_emp_id CHAR(5)
DECLARE employee_cursor SCROLL CURSOR FOR
    SELECT lname, fname, emp_id
    FROM employee
    WHERE hire_date <= 'FEB-14-2004'
OPEN employee_cursor
-- Fetch the last row in the cursor.
FETCH LAST FROM employee_cursor
-- Fetch the row immediately prior to the current row in the cursor.
FETCH PRIOR FROM employee_cursor
-- Fetch the fifth row in the cursor.
FETCH ABSOLUTE 5 FROM employee_cursor
-- Fetch the row that is two rows after the current row.
FETCH RELATIVE 2 FROM employee_cursor
-- Fetch values eight rows prior to the current row into variables.
FETCH RELATIVE -8 FROM employee_cursor
INTO @vc_lname, @vc_fname, @i_emp_id
CLOSE employee_cursor
DEALLOCATE employee_cursor
GO

```

Remember that in SQL Server you must not only *CLOSE* the cursor, but also *DEALLOCATE* it. In some rare cases, you might wish to reopen a closed cursor. You can reuse any cursor that you have closed but not deallocated. The cursor is permanently destroyed only when it is deallocated.

See Also

[CLOSE](#)
[DECLARE CURSOR](#)
[OPEN](#)

GRANT Statement

The *GRANT* statement assigns privileges to users and roles, allowing them to access and use database objects. In addition, most database platforms use the *GRANT* statement to authorize users and roles to create database objects and execute stored procedures, functions, and so on. In other words, on most platforms it can be used to assign both object and usage privileges.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 syntax for *GRANT* allows for only the assignment of object privileges and roles to a specific user:

```
GRANT { {object privilege[, ...] | role[, ...]} }
[ON database_object_name]
[TO grantee[, ...]]
[WITH HIERARCHY OPTION] [WITH GRANT OPTION] [WITH ADMIN OPTION]
[FROM {CURRENT_USER | CURRENT_ROLE}]
```

Keywords

GRANT object privilege[, ...]

Grants privileges to issue one or more SQL statements. Multiple privileges may be granted at one time, separated by commas. You should not combine *ALL PRIVILEGES* with other privileges, but the remainder may be combined in any order. Valid privileges are:

ALL PRIVILEGES

Indicates every privilege the grantor has available to grant; grants all privileges currently assigned to the grantor to the named user(s) and/or for the specified database objects. This is generally not a recommended approach, since it can encourage sloppy permissioning.

EXECUTE

Grants the privilege to execute any ANSI SQL routine. An ANSI *routine* is any stored procedure, user-defined function, or user-defined method.

SELECT | INSERT | UPDATE | DELETE

Grants the specified privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns of a given table on which the privilege will be granted (except for the *DELETE* statement).

REFERENCES

Grants the privilege to use the column(s) in any constraint or assertion, not just a foreign key. The *REFERENCES* clause also allows you to define column-specific details (by including a parenthetical, comma-delimited list) for statements such as *INSERT*, *SELECT*, and so on. Finally, *REFERENCES* grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

TRIGGER

Grants the privilege to create a trigger to operate on a specific table and its columns.

UNDER

Grants the privilege to create subtypes or typed tables.

USAGE

Grants privileges to use a domain, user-defined type, character set, collation, sequence, or translation.

GRANT role[, ...]

Grants a specific, predefined role to the *grantee*; this role must be available to the user or role identified in the *FROM* clause. For example, the database administrator might want to create a role called *Reporter* that has read-only access to

several tables. Subsequently granting this role to a user would bestow upon that user all of the read-only permissions granted to the *Reporter* role. That is, users can have roles assigned to them and receive all of the permissions already assigned to those roles.

ON *database_object_name*

Grants privileges on the specified, predefined database object identified by *database_object_name*, which may be one of the following:

{[TABLE] *object_name* | DOMAIN *object_name* |
COLLATION *object_name* | CHARACTER SET *object_name* |
TRANSLATION *object_name* | SPECIFIC ROUTINE *routine_name*}

TO *grantee*

Assigns the privilege(s) to the named user(s) or role(s), designated as *grantee*. You can assign privileges to multiple users and/or roles, as long as commas separate them. Alternately, privileges may be granted to *PUBLIC*, meaning that all users (including those that will be created in the future) have the specified privileges.

WITH HIERARCHY OPTION

Bestows the *WITH HIERARCHY OPTION* privilege, enabling the grantee to *SELECT* not only from the named table, but also from all of its subtables. This applies to privilege grants only.

WITH GRANT OPTION

Enables the grantee to further grant the privileges to other users. This applies to privilege grants only.

WITH ADMIN OPTION

Bestows the ability to assign a role. The ability to assign role responsibilities is separate from the ability to grant privileges to other users or roles.

FROM {*CURRENT_USER* | *CURRENT_ROLE*}

Names the user who is granting the specified privilege(s), either the *CURRENT_USER* or the *CURRENT_ROLE*. This clause is optional and assumes the current user context.

Rules at a Glance

You may grant many privileges on a single object with one statement, but do not mix the *ALL PRIVILEGES* privilege with the individual SQL statement keywords in a single *GRANT* statement. You can grant a specific privilege on a specific database object to a single user using the syntax:

`GRANT privilege_name ON object_name TO grantee_name`

For example:

`GRANT SELECT ON employee TO Dylan;`

You can grant a specific privilege on a specific object to all users via the *PUBLIC* user. When a grant is issued to *PUBLIC*, that means everyone has permission to the object without the need for a grant specifically to the object. For example:

`GRANT SELECT ON employee TO PUBLIC;`

When granting privileges to multiple grantees, simply place a comma between each:

`GRANT SELECT ON employee TO Dylan, Matt, PUBLIC`

The previous example shows that you may grant privileges to multiple users, in this case *Dylan* and *Matt*, and to the *PUBLIC* user in a single *GRANT* statement. When granting privileges on a table, you may extend or restrict the privileges to the column level by providing a list of columns enclosed in parentheses after the table name. For example:

```
GRANT SELECT ON employee(emp_id, emp_fname, emp_lname, job_id)
  TO Dylan, Matt
```

This example shows that *Dylan* and *Matt* now have select privileges on several columns of the *employee* table.

Programming Tips and Gotchas

Depending on the specific database implementation, views may or may not have access privileges independent from their base tables.

Note that all of the optional *WITH* clauses control extensions to the base privilege(s) granted in the statement. Thus, the following command grants *Dylan* the privilege to *SELECT* records from the *employee* table:

```
GRANT SELECT ON employee TO Dylan;
```

but the following command grants the privilege to *SELECT* records from the *employee* table *and* to grant that *SELECT* privilege to other users:

```
GRANT SELECT ON employee TO Dylan
  WITH GRANT OPTION;
```

Similarly, you can use the *REVOKE* statement to revoke only the *WITH GRANT OPTION*, the individual *SELECT* privilege, or both.

Most of the platforms isolate privileges at the user and role level. Thus, an individual user who is a member of two roles may be granted an individual permission four times—once for herself, once for *PUBLIC*, and once via each of the roles of which she is a member. In this situation, you must be very careful: to remove the privilege completely, you'll have to remove the user from the *PUBLIC* role and then revoke the user's privileges directly in both of the other roles.

MySQL

MySQL provides additional access privileges, primarily relating to object manipulation within a database. *GRANT* in MySQL is available from v3.22.11 and on. The syntax is as follows:

```
GRANT [ { ALL [PRIVILEGES] |
  {SELECT | INSERT | UPDATE} [ (column_name[, ...]) ] | DELETE |
  REFERENCES [ (column_name[, ...]) ] } |
  [{ALTER | CREATE | DROP} [dm1_option]] | [EVENT] | [EXECUTE] |
  [FILE] | [INDEX] | [LOCK TABLES] | [PROCESS] | [RELOAD] |
  [REPLICATION {CLIENT | SLAVE}] | [SHOW DATABASES] | [SHOW VIEW] |
  [SHUTDOWN] |
  [SUPER] | [TRIGGER] | [USAGE]}[, ...]
  ON [ {TABLE | FUNCTION | PROCEDURE} ]
  { [database_name.]table_name | * | *.* | database_name.* }
  TO grantee_name [IDENTIFIED BY [PASSWORD] 'password'][, ...]
  [REQUIRE security_options]
  [WITH with_option[...]]]
```

where:

ALL [PRIVILEGES]

Grants a user all applicable privileges that apply at the current level on one or more objects in the database, with the exception of *WITH GRANT OPTION*. For example, *GRANT ALL ON *.** grants all privileges globally, except *WITH GRANT OPTION*.

SELECT | INSERT | UPDATE | DELETE | REFERENCES

Grants the ability to read, write, modify, and erase data from tables (and possibly from specific columns of a table), respectively. Column-level permissions are allowed on *SELECT*, *INSERT*, and *UPDATE*, but not *DELETE*. Granting column-level permissions can slow down MySQL somewhat, since more permission checking is required. The *REFERENCES* subclause is unimplemented.

{ALTER | CREATE | DROP} [dml_option]

Grants the ability to alter, drop, or create tables and other database objects. When using the *CREATE* syntax, you are not required to specify an *ON* clause to identify a specific table. Specify *ALTER*, *CREATE*, or *DROP*, and an object name; MySQL assumes the object name is for a table. You can further refine this subclause with these *dml_options*:

{CREATE | ALTER | DROP} ROUTINE

Grants the privilege to create, alter, or drop procedures and functions.

CREATE TEMPORARY TABLE

Grants the use of the *CREATE TEMPORARY TABLE* statement, for use with *GRANT CREATE* only.

CREATE USER

Grants the privilege to create, drop, and rename users as well as to use the statement *REVOKE ALL PRIVILEGES*.

CREATE VIEW

Grants the use of the *CREATE VIEW* statement.

EVENT

Grants the privilege to create events for the event scheduler.

EXECUTE

Grants the privilege to run procedures and functions.

FILE

Grants the ability to load data from, or write data to, files using the *SELECT INTO* and *LOAD DATA* commands.

INDEX

Grants the ability to create or drop indexes.

LOCK TABLES

Grants the use of the MySQL command *LOCK TABLES* on tables where the user has *SELECT* privileges.

PROCESS

Grants the ability to view running processes using *SHOW PROCESSLIST*.

RELOAD

Grants the ability to invoke the *FLUSH* and *RESET* commands.

REPLICATION {CLIENT | SLAVE}

Grants a user the privilege to see metadata about replication slaves and masters (*CLIENT*), or grants a replication slave the right to read binlogs from a replication master (*SLAVE*).

SHOW DATABASES

Grants the user the privilege to execute the MySQL command `SHOW DATABASES`.

SHOW VIEW

Grants the use of the `SHOW CREATE VIEW` command.

SHUTDOWN

Grants usage of the `MYSQLADMIN SHUTDOWN` command to kill the server process.

SUPER

Grants the user the privilege of one connection even if the `MAX_CONNECTIONS` threshold is reached. Users with *SUPER* privileges can also execute important MySQL commands such as `CHANGE MASTER`, `KILL`, `MYSQLADMIN DEBUG`, `PURGE [MASTER] LOGS`, and `SET GLOBAL`.

TRIGGER

Grants the ability to create and drop triggers on a specific table, not on a specific trigger. (Before version 5.1.6, the *SUPER* privilege was required to create or drop triggers.)

USAGE

Creates a “no privilege” user account (i.e., a user that has no privileges).

ON {[database_name.]table_name | * | *.* | database_name.*]}

Grants privileges on the specified *table_name*, on all tables within the current database using an asterisk (*), on all tables in all databases with **.**, or on all tables in the specified *database_name* with *database_name.**.

TO *grantee_name* [IDENTIFIED BY [PASSWORD] 'password'][, ...]

Names the user or users who will gain the specified privilege(s). The `PASSWORD` keyword is entirely optional. You may enter multiple users separated by commas. When granting privileges to a new user, you may also optionally set the new password for the new named user at the same time using the `IDENTIFIED BY` clause.

REQUIRE *security_options* = { NONE | {SSL | X509} [CIPHER 'cipher_name'] [AND] } [ISSUER 'issuer_name' [AND]] [SUBJECT 'subject_name'] }

Specifies whether the user must connect with a secured connection or not, according to these security options:

REQUIRE NONE

The account has no special SSL or X509 requirements. This is the default if no `REQUIRE` subclause is specified.

REQUIRE SSL

Allows only SSL-encrypted connections for the account.

REQUIRE X509

Specifies that the client must have a valid certificate, though the certificate, issuer, and subject do not matter. You can specify X509 to also include one or more specific *SUBJECTs*, *ISSUERs*, and *CIPHERs*. The `AND` keyword is optional between the *security_options*.

REQUIRE CIPHER 'cipher_name'

Ensures that a specific cipher method and ciphers and key lengths of sufficient length are used.

REQUIRE ISSUER 'issuer_name'

Ensures that a valid X509 certificate containing a specific CA issuer is used.

REQUIRE SUBJECT 'subject_name'

Ensures that a valid X509 certificate containing a specific subject is used.

WITH with_option

Allows you to set one or more optional privileges:

GRANT OPTION

Entitles the grantee to assign the granted privilege (in fact, any privilege that that user/role possesses) to other users.

MAX_QUERIES_PER_HOUR count

Limits the number of queries not served from the cache that may be performed by the user per hour. Queries served from cache do not count against the limit. The default value of 0 means unlimited queries per hour.

MAX_UPDATES_PER_HOUR count

Limits the number of UPDATE statements that may be performed by the user per hour. The default value of 0 means unlimited updates per hour.

MAX_CONNECTIONS_PER_HOUR count

Limits the maximum number of simultaneously open connections that the user may hold per hour. The default value of 0 means the maximum number of simultaneous connections is determined by the `MAX_USER_CONNECTION` system variable.

Since MySQL is focused on speed, you can implement server-wide features that provide high-speed performance. For example, you can enable the `SKIP_GRANT_TABLES` startup option to disable checks for privileges. This can speed up queries, but obviously no permission checking is done, which means that all users have full access to all resources in the database.

The following are access privileges that are usable with tables: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `DROP`, `GRANT`, `WITH GRANT OPTION`, `CREATE VIEW`, `SHOW VIEW`, `INDEX`, `TRIGGER`, and `ALTER`. `INSERT`, `UPDATE`, and `SELECT` permissions are also grantable or revocable at the column level of a table. Thus, for example, you can grant `SELECT` permissions on a specific column of a specific table.

For routines (i.e., procedures and functions), you can specify `ALTER ROUTINE`, `EXECUTE`, and `WITH GRANT OPTION` permissions. `CREATE ROUTINE` is not actually a routine-level privilege, because a user must have this privilege in the first place to create a routine.

Several privileges can be granted only globally, using `ON *.*` syntax, including: `FILE`, `PROCESS`, `RELOAD`, `REPLICATION CLIENT`, `REPLICATION SLAVE`, `SHOW DATABASES`, `SHUTDOWN`, `SUPER`, and `CREATE USER`.

In any situation, table, database, and column names may each be up to 64 characters in length, and the hostname may be up to 60 characters long. The username (a.k.a. the `grantee_name`) may be up to 16 characters long.

MySQL also supports the possibility of granting rights to a specific user on a specific host if the `grantee_name` is in the form `USER@HOST`. Wildcards can be included in the

hostname of a *grantee_name* to provide the access privilege to a large number of users at one time. A missing hostname is considered the same as the “%” wildcard. For example, consider this GRANT statement:

```
GRANT SELECT ON employee TO katie@% IDENTIFIED BY 'audi',
  annalynn IDENTIFIED BY 'cadillac',
  cody@us% IDENTIFIED BY 'bmw';
```

This statement grants read privileges on the **employee** table to the user *katie* explicitly at any hostname. Similarly, it grants read privileges to the user *annalynn* implicitly at any hostname. Finally, this statement grants read privileges to the user *cody* explicitly at any hostname starting with “US”.

The *grantee_name* must be at most 16 characters long. When specifying the user, password protection may be enforced by including the *IDENTIFIED BY* clause.

The following example grants permissions to three users with passwords:

```
GRANT SELECT ON employee TO dylan IDENTIFIED BY 'porsche',
  kelly IDENTIFIED BY 'mercedes',
  emily IDENTIFIED BY 'saab';
```

If you grant permissions to a user that doesn’t exist in MySQL, MySQL creates that user. The two statements that follow create a user and password with global privileges and another user that has no privileges (the *USAGE* privilege):

```
GRANT SELECT ON *.* TO tony IDENTIFIED BY 'humboldt';
GRANT USAGE ON sales.* TO alicia IDENTIFIED BY 'dakota';
```



Failing to include the *IDENTIFIED BY* clause on a *GRANT* statement that creates a new user will create a user without a password. This practice is insecure and dangerous.

If you do create a new user using the *GRANT* command, and you forget to specify a password, you can (and probably should) set a password using the *SET PASSWORD* command.

MySQL also allows you to issue grants on remote hosts, specified with *@hostname*, and include the wildcard character percent (%). For example, the following grant provides all permissions on all tables of the current database across all hosts in the **notforprofit.org** domain to the user *annalynn*:

```
GRANT ALL ON * TO annalynn@'%notforprofit.org';
```

Privileges are written directly to system tables at four different levels:

Global level

Global privileges apply to all databases on a given server and are stored in the **mysql.user** system table.

Database level

Database privileges apply to all tables in a given database and are stored in the **mysql.db** system table.

Table level

Table privileges apply to all columns in a given table and are stored in the **mysql.tables_priv** system table.

Column level

Column privileges apply to single columns in a given table and are stored in the **mysql.columns_priv** system table.

Because of the way permissions are granted and the system tables involved, you may grant redundant permissions at different levels. For example, you could grant a user *SELECT* privileges on a specific table, and then grant global *SELECT* privileges on that same table to all users of the database. To ensure that that user no longer had the privilege of *SELECT*ing from the table, you would then have to revoke those privileges separately. Furthermore, MySQL does not automatically revoke any privileges when dropping a table or database, though it will drop routine privileges when dropping a procedure or function.

The system tables are also somewhat unprotected, so it is possible to change permissions by issuing *INSERT*, *UPDATE*, and *DELETE* statements against those tables rather than by using *GRANT* or *REVOKE*. Normally, all privileges are read into memory when MySQL starts. Database, table, and column privileges issued by a *GRANT* statement are available immediately and take effect right away. User privileges issued by a *GRANT* statement are noticed immediately and take effect the next time the user connects. If you directly modify permissions in the system tables, none of the changes will be noticed until you restart MySQL or issue a *FLUSH PRIVILEGES* command.

If high-level, certificate-based security is important, you should consider using the *REQUIRE* clause. You do not have to specify all elements of the *REQUIRE* clause if you simply want the user to have a valid SSL or X509 certificate. However, you may add more specifications if they're required to meet your needs. Here, for example, we grant privileges to the user *Tony* on the *localhost* MySQL server and require an X509 subject, issuer, and cipher:

```
GRANT SELECT ON *.* TO 'tony'@'localhost'
  IDENTIFIED BY 'humbolt'
  REQUIRE SUBJECT '/C=EE/ST=CA /L=Frisco/O=MySQL demo client certificate/
    CN=Tony Tubs/Email=tont@myorg.com'
  AND ISSUER '/C=FI/ST=CA /L=UC /O=MySQL /CN=Tony Tubs/Email=tont@myorg.com'
  AND CIPHER 'EDH-RSA-DES-CBC3-SHA';
```

Oracle

Oracle's implementation of the *GRANT* statement supports an enormous number of variations and permutations. The syntax is as follows:

```
GRANT { [object_privilege][, ...] | [system_privilege][, ...] | [role][, ...]
}
[ ON { [schema_name.][object][, ...] |
  [DIRECTORY directory_object_name] |
  [JAVA [ { SOURCE | RESOURCE } ] [schema_name.][object]] } ]
TO {grantee_name[, ...] | role_name[, ...] | PUBLIC}
  [WITH { GRANT | HIERARCHY } OPTION]
  [IDENTIFIED BY password] [WITH ADMIN OPTION];
```

You can grant multiple privileges with a single invocation of the command, but they must all be of the same type (object, system, or role).

For example, you could grant a user three object privileges on a single table in one *GRANT* statement, and then issue a separate grant of two system privileges to a role, and yet a third grant of several roles to a user, but you could not do all of this in one *GRANT* statement.

Following are the parameters to Oracle's *GRANT* statement:

GRANT object_privilege

Grants the specified privilege(s) to the specified *grantee_name* or *role_name* on the named schema object (for example, a table or view). You may combine several object privileges, schema objects, and grantees in a single statement. However, you may *not* grant system privileges or roles in the same statement as object privileges. The object privileges are:

ALL [PRIVILEGES]

Grants all available privileges on a named schema object. Available for use on tables.

ALTER

Grants the privilege to change an existing table using the *ALTER TABLE* statement. Available for use on tables and sequences.

DEBUG

Grants access to a table through a debugger. The access granted applies to any triggers on the table and to information in SQL that directly references the table. Available for use on tables, views, procedures, functions, packages, Java objects, and types.

EXECUTE

Grants the privilege to execute a stored procedure, user-defined function, indextype, library, or package. Available for use on procedures, functions, packages, Java objects, libraries, types, indextypes, and user-defined operators.

INDEX

Grants the privilege to create indexes on a table.

{ ON COMMIT REFRESH | QUERY REWRITE }

Grants the privilege to create a refresh-on-commit materialized view or to create a materialized view for query rewrite on the specified table. Available for materialized views only.

QUERY REWRITE

Grants the privilege to create a materialized view for query rewriting on a specific table. Available for materialized views only.

{ READ | WRITE }

Grants the privilege to read and write files against a named directory, including the full pathname of the operating system directory. Since Oracle has the ability to store files outside of the database, the Oracle server process must run under a user context with privileges on the applicable directories. You can enable Oracle to enforce security on individual users via this security mechanism. Note that *WRITE* is only useful in connection with an external table, such as a logfile or an error file.

REFERENCES

Grants the privilege to define referential integrity constraints. Available for use on tables.

{ SELECT | INSERT | DELETE | UPDATE }

Grants the privilege to execute each of the respective SQL commands against the named schema object. Available for use on tables and views, as well as sequences and materialized views (*SELECT* only). Note that you must grant *SELECT* privileges to a user or role that requires *DELETE* privileges.

You can grant column-level privileges (as a parentheses-encapsulated list following the object name) only when granting *INSERT*, *REFERENCES*, or *UPDATE* object privileges on a table or view.

UNDER

Grants the privilege to create a subview under the named view. Available for use on views and types only.

GRANT *system_privilege*

Grants the specified Oracle system privilege to one or more users or roles. For example, you could grant the *CREATE TRIGGER* privilege or the *ALTER USER* privilege; in each case, issuing a grant for a system privilege empowers the user or role to execute the command of the same name. Refer to Table 3-2 later in this section for a full list of system privileges.

role

Grants a role to the specified user or to another role. In addition to user-defined roles, there are a number of system roles that come predefined with Oracle:

CONNECT, RESOURCE, and DBA

Provided for backward compatibility with older versions of Oracle. Do not use these roles in current or newer versions of Oracle, since they may be dropped in the future.

DELETE_CATALOG_ROLE, EXECUTE_CATALOG_ROLE, and SELECT_CATALOG_ROLE

Grant members of these roles the privilege to delete, execute, and select from data dictionary views and packages.

EXP_FULL_DATABASE and IMP_FULL_DATABASE

Grant members of these roles the privilege to execute the import and export utilities.

AQ_USER_ROLE and AC_ADMINISTRATOR_ROLE

Grant members of these roles the privilege to use or administrate Oracle's Advanced Queuing functionality.

SNMPAGENT

Assigned only to Oracle Enterprise Manager and Intelligent Agent.

RECOVERY_CATALOG_OWNER

Grants the privilege to create users who own their own recovery catalogs.

HS_ADMIN_ROLE

Grants the privilege to access areas of the data dictionary used to support Oracle's heterogeneous services feature.

ON *schema_name*

Grants a user or role the named privilege(s) to on an object in the schema. Database objects include tables, views, sequences, stored procedures, user-defined functions, packages, materialized views, user-defined types, libraries, indextypes, user-defined operators, and any synonyms for any of these. If you do not include the schema name, Oracle assumes the schema of the current user.

Oracle also supports two additional keywords for special cases:

DIRECTORY

Grants privileges on a directory object, which is an Oracle object corresponding to a filesystem directory.

JAVA

Grants privileges on a Java SOURCE or a RESOURCE schema object.

TO {grantee_name | role_name | PUBLIC} [WITH { GRANT | HIERARCHY } OPTION]

Names the user or role that will get the specified privilege(s). PUBLIC may also be used to revoke privileges granted to the PUBLIC role. Multiple grantees can be listed with a comma between each.

WITH GRANT OPTION

Enables the grantee to grant the specific privilege(s) to other users or to PUBLIC, but not to roles.

WITH HIERARCHY OPTION

Enables the grantee to receive privileges on all subobjects when granted privileges on a parent object. This includes subobjects created in the future. You can use this option only when granting the *SELECT* object privilege.

IDENTIFIED BY *password*

Establishes or changes a password that the grantee must use to enable a *role* being granted.

WITH ADMIN OPTION

Enables the grantee to exercise control over a role that you are granting to that grantee. This clause enables the grantee to grant the role to and revoke it from other users or non-GLOBAL roles, and to drop the role or change the authorization needed to access the role.

Grants to users take effect right away. Grants to roles also take effect right away, assuming the roles are enabled. Otherwise, a grant to a role takes effect when the role is enabled. Note that you can grant roles to users and to other roles (including PUBLIC). For example:

```
GRANT sales_reader TO sales_manager;
```

To grant privileges on a view, you must have the specific privilege and the *WITH GRANT OPTION* on all of the base tables for the view.

Any time you want a privilege to be available to all users, simply grant it to PUBLIC:

```
GRANT SELECT ON work_schedule TO public;
```

However, there are some restrictions when granting system privileges and roles:

- A privilege or role should not appear more than once in the GRANT statement.
- Roles cannot be granted to themselves.
- Roles cannot be granted recursively. That is, you can't grant the role *sales_reader* to *sales_manager* and then grant the role *sales_manager* to *sales_reader*.

You can grant multiple privileges of the same type in a single statement. However, the grants must be on the same type of objects:

```
GRANT UPDATE (emp_id, job_id), REFERENCES (emp_id),
  ON employees
  TO sales_manager;
```

As an aside, granting any of the object permissions on a table allows the user (or role) to lock the table in any lock mode using Oracle's *LOCK TABLE* statement.

Recent versions of Oracle have added a number of business intelligence objects. Although the business intelligence features are beyond the scope of this book, granting privileges to them works in essentially the same way as you might grant privileges to any other object. The business intelligence objects include *MINING MODEL*, *CUBE*, *CUBE MEASURE FOLDER*, *CUBE DIMENSIONS*, and *CUBE BUILD PROCESS*.

Oracle also includes some additional advanced queuing features in other add-on components. The privilege syntax for these add-ons' objects is similar to that for objects such as *AQ_USER_ROLE* and *AQ_ADMINISTRATOR_ROLE*. Refer to Oracle's documentation for more information on business intelligence and advanced queuing features and privileges.

Nearly every supported Oracle feature or command is assignable with a GRANT command (as shown in Table 3-2). Privileges can be granted not only on database objects (such as tables and views) and system commands (such as *CREATE ANY TABLE*), but also on schema objects (such as *DIRECTORY*, *JAVA SOURCE*, and *RESOURCE*). The ANY option grants the privilege to execute a given statement against objects of a specific type owned by any user within the schema. Without the ANY option, the user can only execute the statement against objects in his own schema.

Table 3-2. Oracle system privileges

Options	Description
Create Alter Drop	
<i>CREATE CLUSTER</i>	Grants the privilege to create a cluster in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> } ANY <i>CLUSTER</i>	Grants the privilege to create, alter, or drop (respectively) a cluster in any schema.
{ <i>CREATE DROP</i> } ANY <i>CONTEXT</i>	Grants the privilege to create or drop (respectively) any context namespace.
<i>CREATE DIMENSION</i>	Grants the privilege to create dimensions in the grantee's own schema.
{ <i>CREATE ALTER DROP</i> } ANY <i>DIMENSION</i>	Grants the privilege to create, alter, or drop dimensions in any schema.
{ <i>CREATE DROP</i> } ANY <i>DIRECTORY</i>	Grants the privilege to create or drop directory database objects.
{ <i>CREATE ALTER DROP EXECUTE</i> } [ANY] <i>INDEX</i>	Grants the privilege to create, alter, drop, or execute a specific index in one or (using the ANY keyword) all schemas.
<i>CREATE INDEXTYPE</i>	Grants the privilege to create an indextype in the grantee's own schema.
{ <i>CREATE DROP EXECUTE</i> } ANY <i>INDEXTYPE</i>	Grants the privilege to create, alter, drop, or execute an indextype in any schema.
<i>CREATE DATABASE LINK</i>	Grants the privilege to create private database links in the grantee's own schema.
<i>CREATE EXTERNAL JOB</i>	Allows the grantee to create an executable scheduler job that runs on the OS within the grantee's schema.
<i>CREATE [ANY] JOB</i>	Grants the privilege to create, alter, or drop jobs, schedules, or programs in any schema or, without ANY, a single specific schema. Grant this with caution, because it allows the user to execute any code as if she were another user.
{ <i>CREATE DROP</i> } [ANY] <i>LIBRARY</i>	Grants the privilege to create or drop external procedure/function libraries in any or, without ANY, a single specific schema.
{ <i>CREATE ALTER DROP</i> } [ANY] <i>MATERIALIZED VIEW</i>	Grants the privilege to create, alter, or drop materialized views in any schema. <i>CREATE ANY</i> allows creation of a materialized view in any schema, while <i>CREATE</i> without ANY allows it only in the current schema.

Table 3-2. Oracle system privileges (continued)

Options	Description
<code>CREATE OPERATOR</code>	Grants the privilege to create an operator and its bindings in the grantee's own schema.
<code>{CREATE DROP ALTER EXECUTE} ANY OPERATOR</code>	Grants the privilege to create, drop, alter, or execute an operator and its bindings in any schema.
<code>CREATE PUBLIC DATABASE LINK</code>	Grants the privilege to create public database links.
<code>{CREATE ALTER DROP} [ANY] ROLE</code>	Grants the privilege to create a database role (do not use <code>ANY</code> with the <code>CREATE</code> keyword), as well as to alter or drop any existing role.
<code>{CREATE DROP ALTER} ANY OUTLINE</code>	Grants the privilege to create, alter, or drop outlines that can be used in any schema that uses outlines.
<code>SELECT ANY OUTLINE</code>	Grants the privilege to clone private outlines from public ones.
<code>{CREATE ALTER DROP EXECUTE} [ANY] PROCEDURE</code>	Grants the privilege to create, alter, drop, or execute stored procedures, functions, and packages in any schema, or, without <code>ANY</code> , to create a procedure in the current schema.
<code>{CREATE ALTER DROP} PROFILE</code>	Grants the privilege to create, alter, or drop profiles.
<code>ALTER RESOURCE COST</code>	Allows the grantee to set costs for session resources.
<code>{CREATE ALTER DROP GRANT} ANY ROLE</code>	Grants the privilege to create, alter, drop, or grant roles in the database.
<code>{CREATE ALTER DROP} ROLLBACK SEGMENT</code>	Grants the privilege to create, alter, or drop rollback segments.
<code>{CREATE ALTER DROP SELECT} [ANY] SEQUENCE</code>	Grants the privilege to create, alter, drop, or select sequences in any schema in the database or, without <code>ANY</code> , to create a sequence in the current schema.
<code>CREATE SNAPSHOT</code>	Grants the privilege to create snapshots (a.k.a. materialized views) in the grantee's own schema.
<code>{CREATE ALTER DROP} ANY SNAPSHOT</code>	Grants the privilege to create, alter, or drop snapshots in any schema.
<code>CREATE SYNONYM</code>	Grants the privilege to create synonyms in grantee's own schema.
<code>{CREATE DROP} ANY SYNONYM</code>	Grants the privilege to create or drop private synonyms in any schema.
<code>{CREATE DROP} PUBLIC SYNONYM</code>	Grants the privilege to create or drop public synonyms.
Execute	
<code>EXECUTE ANY PROGRAM</code>	Grants the privilege to run any program in a job within the grantee's schema.
<code>EXECUTE ANY CLASS</code>	Grants the privilege to run any job class in a job in the grantee's schema.
Session	
<code>CREATE SESSION</code>	Grants the privilege to connect to the database.
<code>ALTER SESSION</code>	Grants the privilege to issue <code>ALTER SESSION</code> statements.
<code>RESTRICTED SESSION</code>	Enforces that the grantee logs on after the instance is started after the instance is started using the <code>SQL*Plus STARTUP RESTRICT</code> statement.

Table 3-2. Oracle system privileges (continued)

Options	Description
<i>DEBUG CONNECT SESSION</i>	Grants the privilege to connect the current session to the Java Debug Wire Protocol (JDWP) debugger.
<i>DEBUG ANY PROCEDURE</i>	Grants the privilege to debug all PL/SQL objects in the database. Similar to granting <i>DEBUG</i> on all procedures, functions, and packages.
<i>DROP PUBLIC DATABASE LINK</i>	Grants the privilege to drop public database links.
<i>FLASHBACK ARCHIVE ADMINISTRATOR</i>	Grants the privilege to create, alter, or drop any flashback data archive.
<i>GRANT ANY ROLE</i>	Grants the privilege to grant a role to other users.
<i>MANAGE SCHEDULER</i>	Grants full privileges for jobs (i.e., privileges to drop, create, or alter any job class, window, or window group).
<i>ON COMMIT REFRESH</i>	Grants privileges to create a refresh-on-commit materialized view on any table in the database or alter any refresh-on-demand materialized view into a refresh-on-commit materialized view.
Table and Tablespace	
<i>FLASHBACK ANY TABLE</i>	Grants the privilege to issue a SQL flashback query on any table, view, or materialized view in any schema. This privilege is not required for <i>DBMS_FLASHBACK</i> procedures.
<i>CREATE ANY TABLE</i>	Grants the privilege to create tables in any schema. The owner of the schema containing the table must have enough free space in the tablespace to contain the table.
<i>ALTER ANY TABLE</i>	Grants the privilege to alter any table or view in the schema.
<i>BACKUP ANY TABLE</i>	Enables the use of the export utility to incrementally export objects from the schemas of other users.
<i>DELETE ANY TABLE</i>	Allows the deletion of rows from tables, table partitions, or views in any schema.
<i>DROP ANY TABLE</i>	Grants the privilege to drop or truncate tables or table partitions in any schema.
<i>INSERT ANY TABLE</i>	Grants the privilege to insert rows into tables and views in any schema.
<i>LOCK ANY TABLE</i>	Grants the privilege to lock tables and views in any schema.
<i>UPDATE ANY TABLE</i>	Grants the privilege to update rows in tables and views in any schema.
<i>SELECT ANY TABLE</i>	Grants the privilege to query tables, views, or snapshots in any schema.
<i>{CREATE ALTER DROP} TABLESPACE</i>	Grants the privilege to create, alter, or drop tablespaces.
<i>MANAGE TABLESPACE</i>	Grants the privilege to take tablespaces offline and online and begin and end tablespace backups.
<i>UNLIMITED TABLESPACE</i>	Grants the privilege to use an unlimited amount of any tablespace. This privilege overrides any specific quotas assigned. If you revoke this privilege from a user, the user's schema objects remain, but further tablespace allocation is denied unless authorized by specific tablespace quotas. You cannot grant this system privilege to roles.

Table 3-2. Oracle system privileges (continued)

Options	Description
Trigger	
<i>ADMINISTER DATABASE TRIGGER</i>	Grants the privilege to create a trigger on <i>DATABASE</i> . (The user/role must also have the <i>CREATE TRIGGER</i> or <i>CREATE ANY TRIGGER</i> privilege.)
<i>{CREATE ALTER DROP} [ANY] TRIGGER</i>	Grants the privilege to create, alter, or drop database triggers in any schema or, without <i>ANY</i> , to create, alter, or drop a trigger in the current schema.
Type	
<i>{CREATE ALTER DROP} [ANY] TYPE</i>	Grants the privilege to create, alter, or drop object types and object-type bodies in any schema or, without <i>ANY</i> , to create a type in the current schema.
<i>EXECUTE ANY TYPE</i>	If granted to a specific user, allows that user to use and reference object types and collection types in any schema and to invoke methods of an object type in any schema. If granted to a role, individual members of that role will <i>not</i> be able to invoke methods of an object type in any schema.
User	
<i>CREATE USER</i>	Grants the privilege to create users. This privilege also allows the grantee to: <ul style="list-style-type: none"> Assign quotas on any tablespace. Set default and temporary tablespaces. Assign a profile as part of a <i>CREATE USER</i> statement.
<i>ALTER USER</i>	Grants the privilege to alter any user. This privilege authorizes the grantee to: <ul style="list-style-type: none"> Change another user's password or authentication method. Assign quotas on <i>any</i> tablespace. Set default and temporary tablespaces. Assign a profile and default roles.
<i>BECOME USER</i>	Grants the privilege to become another user (required by any user performing a full database import).
<i>DROP USER</i>	Grants the privilege to drop users.
<i>UNDER ANY TYPE</i>	Allows the creation of subtypes under any parent object type, such as a table or view.
View	
<i>{CREATE DROP} [ANY] VIEW</i>	Grants the privilege to create or drop views in any schema or, without <i>ANY</i> , to create a view in the current schema.
<i>UNDER ANY VIEW</i>	Grants the privilege to create a subview under any parent view.
<i>MERGE ANY VIEW</i>	Grants the privilege to the optimizer to use view merging on behalf of the user to speed query performance.
Additional privileges	
<i>ADMINISTER [ANY] SQL TUNING SET</i>	Grants privileges to create, drop, select, load, or delete an owned SQL tuning set (without the <i>ANY</i> keyword) or any user's SQL tuning set (with the <i>ANY</i> keyword) through the <i>DBMS_SQLTUNE</i> package.
<i>ADVISOR</i>	Grants the privilege to access the advisor framework through PL/SQL packages.

Table 3-2. Oracle system privileges (continued)

Options	Description
<i>ALTER DATABASE</i>	Grants the privilege to alter the database.
<i>ALTER SYSTEM</i>	Grants the privilege to issue <i>ALTER SYSTEM</i> statements.
<i>ANALYZE ANY</i>	Grants the privilege to analyze any table, cluster, or index in any schema.
<i>ANALYZE ANY DICTIONARY</i>	Grants the privilege to analyze any data dictionary object.
<i>AUDIT ANY</i>	Grants the privilege to audit any object in any schema using <i>AUDIT schema_object</i> s statements.
<i>AUDIT SYSTEM</i>	Grants the privilege to issue <i>AUDIT sql_statements</i> statements.
<i>{DELETE EXECUTE SELECT} CATALOG_ROLE</i>	Grants privileges to access data dictionary views and packages.
<i>CHANGE NOTIFICATION</i>	Allows the grantee to register for and receive notifications of a change in a database or query.
<i>COMMENT ANY TABLE</i>	Allows comments on any table, view, or column in any schema.
<i>EXEMPT ACCESS POLICY</i>	Circumvents fine-grained access control. Generally, this is ill-advised since the user can then bypass any application-driven security policies.
<i>FORCE ANY TRANSACTION</i>	Forces the commit or rollback of any in-doubt distributed transaction in the local database and induces the failure of a distributed transaction.
<i>FORCE TRANSACTION</i>	Forces the commit or rollback of grantee's own in-doubt distributed transactions in the local database.
<i>{EXP IMP} FULL_DATABASE</i>	Grants the privilege to use the import and export utilities.
<i>GRANT ANY PRIVILEGE</i>	Grants any system privilege.
<i>GRANT ANY OBJECT PRIVILEGE</i>	Grants any object privilege available.
<i>RECOVERY_CATALOG_OWNER</i>	Grants the privilege to create and own recovery catalogs.
<i>RESUMABLE</i>	Enables resumable space allocations.
<i>SELECT ANY DICTIONARY</i>	Allows the user to query any data dictionary object in the <i>SYS</i> schema.
<i>SELECT ANY TRANSACTION</i>	Grants the privilege to view the contents of the <i>FLASHBACK_TRANSACTION_QUERY</i> view. (Grant this with caution, because it allows the grantee to see all data in the database!)
<i>SYSDBA</i>	Provides the user with <i>RESTRICTED SESSION</i> permission and authorizes the user to: <ul style="list-style-type: none"> • Perform <i>STARTUP</i> and <i>SHUTDOWN</i> operations • Issue <i>CREATE/ALTER DATABASE</i> commands and <i>OPEN/MOUNT/BACKUP</i> commands • Change the character set • Perform <i>ARCHIVELOG</i> and <i>RECOVERY</i> operations
<i>SYSOPER</i>	Provides the user with <i>RESTRICTED SESSION</i> permission and authorizes the user to: <ul style="list-style-type: none"> • Perform <i>STARTUP</i> and <i>SHUTDOWN</i> operations • Issue <i>ALTER DATABASE</i> and <i>OPEN/MOUNT/BACKUP</i> commands • Perform <i>ARCHIVELOG</i> and <i>RECOVERY</i> operations

Table 3-2. Oracle system privileges (continued)

Options	Description
<i>QUERY REWRITE</i>	Enables rewrites using a materialized view, or creates a function-based index, when that materialized view or index references tables and views that are in the grantee's own schema.
<i>GLOBAL QUERY REWRITE</i>	Enables rewrites using a materialized view, or creates a function-based index.

Any of the privileges shown in Table 3-2 containing the *ANY* keyword has a special meaning. In effect, *ANY* gives a user the privilege to execute the specified command or operation in any schema. If you want to include all user schemas but exclude the *SYS* system schema, keep the *O7_DICTIONARY_ACCESSIBILITY* initialization parameter at its default value of *FALSE*.

PostgreSQL

PostgreSQL supports a subset of the ANSI GRANT commands supporting object privileges against database objects such as tables, sequences, functions, and so forth. PostgreSQL's GRANT syntax is as follows:

```
GRANT { role_name | { ALL [PRIVILEGES] | SELECT | INSERT | DELETE | UPDATE |  
RULE |  
REFERENCES| TRIGGERS | CREATE | USAGE }[,...] }  
ON { [TABLE | SEQUENCE | DATABASE | FUNCTION | LANGUAGE | SCHEMA |  
TABLESPACE]  
object_name[,...]  
[WITH GRANT OPTION]  
TO {grantee_name | GROUP group_name | PUBLIC}[,...]  
[WITH ADMIN OPTION]
```

where:

GRANT role_name

Grants a previously created role to a specific user following the syntax *GRANT role_name TO grantee_name [WITH ADMIN OPTION]*. The grantee then receives all of the privileges inherent with the role. Role-based privileges may not be granted to *PUBLIC*.

ALL [PRIVILEGES]

Grants all privileges the grantor has available to grant. Using *ALL* is generally not a recommended approach, since it can encourage sloppy permissioning.

CREATE

Grants the privilege to create objects of the specified type. Usable with *ON DATABASE*, *ON SCHEMA*, and *ON TABLESPACE*. On databases, this privilege allows the creation of new schemas within the database. On schemas, it allows the creation of new objects and the renaming of existing objects in the schema. On tablespaces, this privilege allows the creation of tables and indexes within the tablespace.

CONNECT | TEMP[ORARY]

Grants the privilege to connect to a specific database and/or to create temporary tables when using the named database.

SELECT | INSERT | DELETE | UPDATE

Grants the specified privilege to a given user on the specified database object, such as a table or view. For column-level privileges, you may add a parenthetical list of specific columns of a given table; the permission then applies only to the columns of the table that are named.

RULE

Grants the privilege to create or drop a rule on a table or view.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

TRIGGERS

Grants the privilege to create a trigger to operate on a specific table and its columns.

USAGE

Grants privileges to use these objects: *CHARACTER SET*, *COLLATION*, *TRANSLATION*, and *DOMAIN* (a user-defined datatype). You can grant *USAGE* privileges on sequences, languages, and schemas.

ON { [TABLE] | SEQUENCE | DATABASE | FUNCTION | LANGUAGE | SCHEMA | TABLESPACE *object_name*, ... }

Specifies the object to which permissions are granted. The optional *TABLE* keyword is the default if omitted. You may grant permissions on more than one object at a time by specifying a comma-delimited list of object names, as long as the objects are of the same type.

WITH GRANT OPTION

Allows the grantee to grant any of his privileges in turn to any other users.

TO {grantee_name | GROUP *group_name* | PUBLIC}{, ...}

Names the user(s) or role(s) that will receive the specified privilege(s). In PostgreSQL, *PUBLIC* is a synonym for all users. PostgreSQL allows permissions to be assigned to a *GROUP* (essentially a role), provided it is a valid, pre-existing *group_name*.

WITH ADMIN OPTION

Grants the privilege for a member of a role to grant membership in that role to or revoke it from other users.

Multiple privileges may be granted at one time, in a comma-separated list. However, when granting privileges, they should be of a related type. You should not combine *ALL [PRIVILEGES]* with other privileges, but many of the remaining privileges may be combined in any order.

PostgreSQL does not support the *WITH* clauses or column-level permissions. Under PostgreSQL, the creator of an object has all privileges on the object. She may revoke most of her own privileges on that object, but the privilege for the creator of an object to assign privileges on that object cannot be revoked. Similarly, the right to drop an object is permanently assigned to the creator of the object and cannot be revoked. Users other than the creator of an object do not have permissions on an object until they are specifically granted.

Certain privileges are allowed only for certain objects, as shown here:

TABLE

May be granted *SELECT*, *INSERT*, *UPDATE*, *DELETE*, *REFERENCES*, *TRIGGER*, and *ALL [PRIVILEGES]*.

SEQUENCE

May be granted *USAGE*, *SELECT*, and *ALL [PRIVILEGES]*.

DATABASE

May be granted *CONNECT*, *CREATE*, *TEMP[ORARY]*, and *ALL [PRIVILEGES]*.

FUNCTION

May be granted *EXECUTE* and *ALL [PRIVILEGES]*.

LANGUAGE

May be granted *USAGE* and *ALL [PRIVILEGES]*.

SCHEMA

May be granted *CREATE*, *USAGE*, and *ALL [PRIVILEGES]*.

TABLESPACE

May be granted *CREATE* and *ALL [PRIVILEGES]*.

PostgreSQL's support for the *GRANT* statement is elementary. In the following examples, *INSERT* privileges are granted on the **publishers** table to the **PUBLIC** role and *SELECT* and *UPDATE* privileges are granted on the **sales** table to the users *Emily* and *Dylan*:

```
GRANT INSERT ON TABLE publishers TO PUBLIC;
GRANT SELECT, UPDATE ON sales TO emily, dylan;
```

The following example grants database control privileges to the user *Katie* and then assign the users *Katie* and *Anna* to a new group:

```
GRANT ALL ON DATABASE publishing TO katie;
GRANT manager_role ON katie, anna WITH ADMIN OPTION;
```

SQL Server

SQL Server's implementation of *GRANT* offers few variations from the ANSI standard. SQL Server does not support the ANSI standard's *FROM* clause or the *HIERARCHY* or *ADMIN* options, but it does offer the ability to grant specific system privileges, and to grant them under another user's security context. The syntax is as follows:

```
GRANT { [object_privilege][, ...] | [system_privilege] }
[ON { [definition_scope::[[schema_name.]object_name] [(column[, ...])] } ]
TO { grantee_name[, ...] | role[, ...] | PUBLIC | GUEST }
[WITH GRANT OPTION]
[AS {group | role}]
```

where:

GRANT object_privilege

Grants privileges for a variety of objects, which may all be combined in any order (with the exception of the *ALL [PRIVILEGES]* privilege). Object privileges may be granted for tables, views, functions (table-valued, scalar, and aggregate), procedures (stored and extended), service queues, and synonyms. The privileges that may be granted on objects include:

ALL [PRIVILEGES]

Grants all privileges currently assigned to the named user(s) and/or for the specified database objects. This is generally not a recommended approach, since it can encourage sloppy programming. *ALL* can only be used by members of the **SYSADMIN** and **DB_OWNER** system roles, or by the object owner. *ALL* is deprecated in SQL Server 2008 and only included for backward compatibility. *GRANT ALL* is now shorthand for the following privileges:

GRANT ALL on databases

Grants the privileges **BACKUP DATABASE**, **BACKUP LOG**, **CREATE DATABASE**, **CREATE DEFAULT**, **CREATE FUNCTION**, **CREATE PROCEDURE**, **CREATE RULE**, **CREATE TABLE**, and **CREATE VIEW**.

GRANT ALL on functions

Grants the privileges **EXECUTE** and **REFERENCES** for scalar functions and **DELETE**, **INSERT**, **REFERENCES**, **SELECT**, and **UPDATE** for table-valued functions.

GRANT ALL on procedures (includes both stored procedures and extended stored procedures)

Grants the **EXECUTE** privilege.

GRANT ALL on tables and views

Grants the privileges **DELETE**, **INSERT**, **REFERENCES**, **SELECT**, and **UPDATE**.

{SELECT | INSERT | DELETE | UPDATE}

Grants the specified privilege to a given user on the specified database object, such as a table or view. For column-level privileges, add a parenthetical list of table columns; the privileges will then be assigned only for the named columns.

REFERENCES

Grants the privilege to create or drop foreign key constraints referencing the database object as a parent object.

EXECUTE

Grants the privilege to execute a stored procedure, user-defined function, or extended stored procedure.

GRANT system_privilege

Grants the privilege to execute one of the corresponding statements (**EXECUTE**) or to select against one (or more) of the views in the **SYS** schema (**SELECT**). More details can be found in Table 3-3.

The privilege to issue a **CREATE** statement also implies the privilege to issue the equivalent **ALTER** or **DROP** statement. However, granting a single granular permission does not ensure that all required permissions have been granted—for example, granting **EXECUTE** permission on *sp_addlinkedserver* doesn't allow a user to create a linked server unless that user has also been added as a member to the **sysadmin** role.

ON [definition_scope::][[schema_name.]object_name] [(column[, ...])]

Specifies the object for which the privilege is granted. This clause is not required when granting system privileges. The *definition_scope* is required and indicates the type of object to which you will be granting privileges. *definition_scopes*

must always include the assignment symbols (::). The definition scope *OBJECT*:: is for tables, views, and other database objects. Other *definition_scopes* are applicable to objects of the same name: *APPLICATION ROLE*, *ASSEMBLY*, *ASYMMETRIC KEY*, *CERTIFICATE*, *ENDPOINT*, *FULLTEXT CATALOG*, *LOGIN*, *ROLE*, *SCHEMA*, service broker scopings (*CONTRACT*, *MESSAGE TYPE*, *REMOTE SERVICE BINDING*, *ROUTE*, and *SERVICE*), *SYMMETRIC KEY*, *TYPE*, *USER*, and *XML SCHEMA COLLECTION*. If the object is a table or view, you may optionally grant privileges on specific columns. You can grant *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and *REFERENCES* privileges on a table or view, but you can only grant *SELECT* and *UPDATE* privileges on the columns of a table, a view, or a user-defined function. You can grant *EXECUTE* privileges on a stored procedure or extended stored procedure, and *REFERENCES* privileges on user-defined functions. *REFERENCES* privileges are also required on functions or views created using the *WITH SCHEMABINDING* clause.

TO {grantee_name[, ...] | role[, ...] | PUBLIC | GUEST}

Names the user or role that will gain the privilege being granted. Multiple grantees can be listed in a comma-separated list. Use the keyword *PUBLIC* to grant privileges to the *PUBLIC* role, a role that implicitly includes all users. SQL Server also supports a *GUEST* role, which is the role used by all users who do not have any other roles in the database. Because SQL Server allows two different security models (one based on the database and one based on the Windows operating system), you may be granting privileges to a SQL Server user, a Windows user, a Windows group, or a SQL Server database role. The *grantee_name* may be any of these.

WITH GRANT OPTION

Enables the grantee to further grant the given permission to other users or roles. You may use *WITH GRANT OPTION* only when granting object permissions.

AS {group | role}

Specifies an alternative user or group that has the authority in the current database to execute the *GRANT* command. You can use the *AS* clause to grant privileges as if the granting session is part of a different group or role context from the current one.

Privileges may not be granted in any database other than the current database, nor may privileges be granted in more than one database at a time.



SQL Server has a somewhat different security model than the other platforms described here (and from the ANSI standard). Like the ANSI standard, SQL Server uses the *GRANT* command to assign specific privileges to a user or role and the *REVOKE* command to remove those permissions. However, SQL Server augments these commands with the *DENY* command, using essentially the same syntax as *REVOKE*.

DENY, under SQL Server, allows DBAs to declare certain privileges as strictly off-limits to a user or role. Any privileges that have been denied must be revoked before they can be granted. As such, *DENY* takes precedence over *GRANT* and *REVOKE*. *DENY* can be used to effectively take away any permission from an individual who might otherwise inherit permissions from membership in Windows groups or SQL Server database roles.

SQL Server applies precedence to permissions. Thus, if a user has a privilege granted at the user level, but the permission is revoked at the group level (when the user is a member of a group), the permission is revoked at both levels.

System privileges to *CREATE* or *ALTER* an object also imply the privilege to *DROP* the object. System privileges to *CREATE* an object additionally imply the privilege to *ALTER* the object. A complete list of system privileges is shown in Table 3-3.

Table 3-3. SQL Server system privileges

Options	Description
<i>ADMINISTER BULK OPERATIONS</i>	Grants the privilege to execute bulk operations, such as <i>BULK INSERT</i> , within the current database.
{ <i>ALTER ANY</i> { <i>APPLICATION ROLE</i> <i>DATABASE DDL TRIGGER</i> <i>DATASPACE</i> <i>USER</i> }}	Grants the privilege to alter any pre-existing object of the named type within the current database.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>ASSEMBLY</i>	Grants the privilege to <i>CREATE</i> an assembly, or to <i>ALTER</i> a specific assembly or (with <i>ANY</i>) all assemblies within the database.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>ASYMMETRIC KEY</i>	Grants the privilege to <i>CREATE</i> an asymmetric security key, or to <i>ALTER</i> a specific asymmetric security key or (with <i>ANY</i>) all asymmetric security keys within the database.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>CERTIFICATE</i>	Grants the privilege to <i>CREATE</i> a security certificate, or to <i>ALTER</i> a specific security certificate or (with <i>ANY</i>) all security certificates within the database.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>CONTRACT</i>	Grants the privilege to alter a specific service broker contract or (with <i>ANY</i>) all service broker contracts within the current database, or to create a new contract.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>DATABASE DDL EVENT NOTIFICATION</i>	Grants the privilege to alter a specific database DDL event notification or (with <i>ANY</i>) all database DDL event notifications, or to create a new database DDL notification.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>FULLTEXT CATALOG</i>	Grants the privilege to alter a specific full-text catalog or (with <i>ANY</i>) all full-text catalogs within the current database, or to create a new full-text catalog.
<i>CREATE FUNCTION</i>	Grants the privilege to create new functions.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>MESSAGE TYPE</i>	Grants the privilege to alter a specific service broker message type or (with <i>ANY</i>) all service broker message types within the current database, or to create new message type.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>REMOTE SERVICE BINDING</i>	Grants the privilege to alter a specific service broker service binding or (with <i>ANY</i>) all service bindings within the current database, or to create new service binding.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>ROLE</i>	Grants the privilege to alter a specific role or (with <i>ANY</i>) all roles on the server, or to create new role.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>ROUTE</i>	Grants the privilege to alter a specific service broker route or (with <i>ANY</i>) all service broker route within the current database, or to create new route.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>SCHEMA</i>	Grants the privilege to alter a specific schema or (with <i>ANY</i>) all schemas on the server, or to create new schema.
{ <i>ALTER</i> [<i>ANY</i>] <i>CREATE</i> } <i>SERVICE</i>	Grants the privilege to alter a specific service broker service or (with <i>ANY</i>) all service broker services within the current database, or to create new service.

Table 3-3. SQL Server system privileges (continued)

Options	Description
<code>{ALTER [ANY] CREATE} SYMMETRIC KEY</code>	Grants the privilege to alter a specific security symmetric key or (with ANY) all security symmetric keys within the current database, or to create new symmetric key.
<code>AUTHENTICATE [SERVER]</code>	Grants the privilege to connect to and authenticate a server.
<code>BACKUP {DATABASE LOG}</code>	Grants to a specific user or role the privilege to execute the <code>BACKUP DATABASE</code> or <code>BACKUP LOG</code> statements.
<code>CHECKPOINT</code>	Grants to a specific user or role the privilege to execute <code>CHECKPOINT</code> statements.
<code>CONNECT</code>	Grants permission to a service broker endpoint to connect to an instance of SQL Server.
<code>CONNECT REPPLICATION</code>	Grants permission to a replication scheme to connect to an instance of SQL Server.
<code>CONTROL [SERVER]</code>	Grants all permissions on all objects (for <code>CONTROL SERVER</code>) or all permissions on all objects in named database or schema (for <code>CONTROL</code>), as well as the ability to grant privileges to other users. When used with a single object, grants full control over the named object, as in <code>GRANT CONTROL ON USER::anna TO manager123;</code> .
<code>CREATE {AGGREGATE DATABASE DEFAULT PROCEDURE QUEUE RULE SYNONYM TABLE TYPE VIEW XML SCHEMA COLLECTION}</code>	Grants the privilege to create the object of the specified type within the current server, database, or schema.
<code>IMPERSONATE</code>	Grants a user the privilege to act as another user, as in <code>GRANT IMPERSONATE ON USER::katie TO manager123;</code> .
<code>RECEIVE</code>	Grants the privilege for a user to receive a given service broker queue, as in <code>GRANT RECEIVE ON query_notification_errorqueue TO emily;</code> .
<code>SEND</code>	Grants the privilege to send query notifications from a service broker queue to a named login, as in <code>GRANT SEND ON SERVICE::://mysvc TO dylan;</code> .
<code>SHOWPLAN</code>	Grants the privilege to see query execution plans.
<code>SHUTDOWN</code>	Grants the privilege to shut down the SQL server.
<code>SUBSCRIBE QUERY NOTIFICATIONS</code>	Grants the privilege to receive notifications from a given query notification queue, as in <code>GRANT SUBSCRIBE QUERY NOTIFICATIONS TO sam;</code> .
<code>TAKE OWNERSHIP</code>	Grants the privilege to transfer ownership of an XML schema collection from one user to another.
<code>VIEW {DATABASE STATE DEFINITION}</code>	Grants the privilege to view the metadata that describes all of a database, regardless of whether the user owns or may use that database (for <code>DATABASE STATE</code>) or the metadata about a server, database, schema, or individual object (for <code>DEFINITION</code>).

SQL Server also contains a number of fixed system roles with preset privileges on both objects and commands. The SQL Server system roles are:

SYSADMIN

Can perform any activity on the server and has access to all objects

SERVERADMIN

Can set server configuration options as well as shut down and restart the server

SETUPADMIN

Can start up procedures and linked servers

SECURITYADMIN

Can read error logs, change passwords, administrate logins, and grant *CREATE DATABASE* permissions

PROCESSADMIN

Can administrate processes running on the SQL server

DBCREATOR

Can create, alter, and drop databases

DISKADMIN

Can administrate disks and files

BULKADMIN

Can perform Bulk Copy Program (BCP) operations and execute *BULK INSERT* statements (and regular *INSERT* statements) against tables in the database.

The **SYSADMIN** system role can grant any permission in any database on the server. Members of the **DB_OWNER** and **DB_SECURITYADMIN** system roles can grant any permission on any statement or object in databases they own. Members of the **DB_DDLADMIN** system role, the **SYSADMIN** system role, and the database owner may grant system privileges.

Members of the system roles may add other logins or users to their roles, but not via a **GRANT** statement. Instead, to add a user to a system role, you must use a SQL Server system stored procedure called **sp_addsrvrolemember**.

In addition to the server-wide system roles, there are a number of database-wide system roles. That is, whereas the previous system roles have privileges that span every database on the server, the following system roles exist in each individual database and have privileges within the scope of a specific database:

DB_OWNER

Grants the privilege to perform all the functions of the other database system roles, plus other maintenance and configuration tasks for the database

DB_ACCESSADMIN

Grants the privilege to add Windows and SQL Server users and groups to the database, or to remove them

DB_DATAREADER

Grants the privilege to read data from all user tables in the database

DB_DATAWRITER

Grants privileges to read, add, change, or delete data in all user tables in the database

DB_DDLADMIN

Grants privileges to add, change, and drop objects in the database, as well as run all DDL statements

DB_SECURITYADMIN

Grants the privilege to administrate SQL Server roles and users as well as statement and object permissions throughout the database

DB_BACKUPOPERATOR

Grants the privilege to perform backups of the database

DB_DENYDATAREADER

Denies permission to read data in the database

DB_DENYDATAWRITER

Denies permission to change data in the database

Like the server-wide system roles, database-wide system roles cannot be granted to a user via the *GRANT* statement. Instead, you must use a SQL Server system stored procedure called **sp_addrolemember**.

In the following example, the *CREATE DATABASE* and *CREATE TABLE* system privileges are granted to the users *emily* and *sarah*. Next, numerous permissions on the *titles* table are granted to the *editors* group. The editors are then able to grant permissions to others:

```
GRANT CREATE DATABASE, CREATE TABLE TO emily, sarah
GO
GRANT SELECT, INSERT, UPDATE, DELETE ON titles
TO editors
WITH GRANT OPTION
GO
```

The following example grants permissions to the database user *sam* and the Windows user *jacob*:

```
GRANT CREATE TABLE TO sam, [corporate\jacob]
GO
```

Finally, the following example shows how to grant permissions using the optional *AS* keyword. In this example, the user *emily* owns the *sales_detail* table, and she grants *SELECT* privileges to the *sales_manager* role. The user *kelly*, who is a member of the *sales_manager* role, wants to grant *SELECT* privileges to *sam*, but she cannot because the permissions were granted to the *sales_manager* role and not to her explicitly. *kelly* can use the *AS* clause to get around this hurdle:

```
-- Initial grant
GRANT SELECT ON sales_detail TO sales_manager
WITH GRANT OPTION
GO
-- Kelly passes the privilege to Sam as a member of the sale_manager role
GRANT SELECT ON sales_detail TO sam AS sales_manager
GO
```

See Also

REVOKE

IN Operator

The *IN* operator provides a way to delineate a list of values, either explicitly listed or from a subquery, and compare a value against that list in a *WHERE* or *HAVING* clause. In other words, it gives you a way to say “Is value A *in* this list of values?”

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
{WHERE | HAVING | {AND | OR}}
value [NOT] IN ({comp_value1, comp_value2[, ...] | subquery})
```

Keywords

{WHERE | HAVING | {AND | OR}} *value*

IN is permitted under either the *WHERE* or the *HAVING* clause. The *IN* comparison may also be a part of an *AND* or *OR* clause in a multicondition *WHERE* or *HAVING* clause. *value* may be of any datatype, but is usually the name of a column of the table referenced by the transaction, or perhaps a host variable when used programmatically.

NOT

Optionally tells the database to look for a result set that contains values that are not in the list.

IN ({comp_value1, comp_value2[, ...] | subquery})

Defines the list of comparative values (hence, *comp_value*) to compare against. Each *comp_value* must be of the same or a compatible datatype as the initial *value*. They are also governed by standard datatype rules. For example, string values must be delimited by quotes, while integer values need no delimiters. As an alternative to listing specific values, you may use parentheses to enclose a subquery that returns one or more values of a compatible datatype.

In the following example, generated on SQL Server, we look for all employees in the **employee** table of the **HR** database who have a home state of Georgia, Tennessee, Alabama, or Kentucky:

```
SELECT *
FROM hr..employee
WHERE home_state IN ('AL','GA','TN','KY')
```

Similarly, we can look for all **employees** in the **HR** database who are authors in the **PUBS** database:

```
SELECT *
FROM hr..employee
WHERE emp_id IN (SELECT au_id FROM pubs..authors)
```

We can also use the *NOT* keyword to return a result set based upon the absence of a value. In the following case, the company headquarters is located in New York, and many workers commute in from neighboring states. We want to see all such workers:

```
SELECT *
FROM hr..employee
WHERE home_state
    NOT IN ('NY','NJ','MA','CT','RI','DE','NH')
```

Note that Oracle, while fully supporting the ANSI functionality, extends the functionality of the *IN* operator by allowing multiple argument matches. For example, the following *SELECT...WHERE...IN* statement is acceptable on Oracle:

```
SELECT *
FROM hr..employee e
WHERE (e.emp_id, e.emp_dept) IN
( (242, 'sales'), (442, 'mfg'), (747, 'mkt') )
```

See Also

ALL/ANY/SOME
BETWEEN
EXISTS
LIKE
SELECT
SOME/ANY

INSERT Statement

The *INSERT* statement adds rows of data to a table or view.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported
SQL Server	Supported, with variations

The *INSERT* statement allows rows to be written to a table through one of several methods:

- One or more rows can be inserted using the *DEFAULT* values specified for a column via the *CREATE TABLE* or *ALTER TABLE* statements.
- The actual values to be inserted into each column of the record can be declared (this is the most common method).
- The result set of a *SELECT* statement can be inserted into a table or view, populating it with many records simultaneously.

SQL2003 Syntax

```
INSERT INTO [ONLY] {table_name | view_name} [(column1[, ...])]
[OVERRIDE {SYSTEM | USER} VALUES]
{DEFAULT VALUES | VALUES (value1[, ...]) | select_statement}
```

Keywords**ONLY**

Used on typed tables only, this optional keyword ensures that the values inserted into *table_name* are not inserted into any subtables.

{table_name | view_name} [(column1[, ...])]

Declares the updatable target table or view into which the records will be inserted. You must have *INSERT* privileges on the table or, at a minimum, on the columns that will receive the inserted values. If no schema information is included, as in *scott.employee*, the current schema and user context are assumed. You may optionally include a list of the columns in the target table or view that will receive data.

OVERRIDE {SYSTEM | USER} VALUES

Requires the *SYSTEM* keyword when inserting a literal *value* into a column that would otherwise be given a system-generated value, such as an autogenerated sequence number. The *OVERRIDE USER VALUES* clause does the converse, by inserting system-supplied values even if a user has provided literal values to insert.

DEFAULT VALUES

Inserts all values declared via the *DEFAULT* column characteristic on the target table where they exist, and *NULLs* where they do not exist. (Of course, the *DEFAULT* column characteristic can specify *NULL* as a value, too.) This operation inserts a single record. It might encounter an error, depending on how the *PRIMARY KEY* constraint or *UNIQUE* constraint is constructed on the target table (assuming such constraints exist).

VALUES (value1[, ...])

Specifies the actual values to be inserted into the target table. The number of values must match the exact number of columns in the column list, if one is provided. Furthermore, the values must be datatype- and size-compatible with the columns of the target table. Each value in the value list corresponds to the column with the same ordinal number in the column list. Thus, the first column will get its data from the first value, the second column from the second value, and so on until all columns are satisfied. You may also optionally use the keywords *DEFAULT* to insert a column's default value and *NULL* to insert a *NULL* value.

select_statement

Inserts rows retrieved via the specified *SELECT* statement into the target table or view. The values retrieved by the fully formed *SELECT* statement's select item list correspond directly to the columns of the column list. The target table or view may not be referenced in the *SELECT* statement's *FROM* or *JOIN* clauses.

Rules at a Glance

You may insert values into tables, and into views built upon a single source table. The *INSERT...VALUES* statement adds a single row of data to a table, using literal values supplied in the statement. In the following example, a new row in the **authors** table is inserted for the author **Jessica Rabbit**:

```
INSERT INTO authors (au_id, au_lname, au_fname, phone,
    address, city, state, zip, contract)
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,
    '1717 Main St', NULL, 'CA', '90675', 1)
```

Every column in the table is assigned a specific, literal value except the **phone** column, which is assigned the default value (as defined in the *CREATE TABLE* or *ALTER TABLE* statement), and the **city** column, which is set to *NULL*.

It's important to remember that you may skip columns in the table and set them to *NULL*, assuming they allow *NULL* values. Inserts that leave some columns *NULL* are called *partial INSERTs*. Here is a partial *INSERT* that performs the same work as the preceding example:

```
INSERT INTO authors (au_id, au_lname, au_fname, phone, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT, 1)
```

The *INSERT* statement, combined with a nested *SELECT* statement, allows a table to be quickly populated with one or more rows from the result set of the *SELECT* statement. When using *INSERT...SELECT* between a target table and a source table, it is important to ensure that the datatypes returned by the *SELECT* statement are compatible with the datatypes in the target table. For example, to load data from the **sales** table into the **new_sales** table:

```
INSERT INTO sales (stor_id, ord_num, ord_date, qty, payterms,
    title_id)
```

INSERT Statement

```
SELECT
  CAST(store_nbr AS CHAR(4)),
  CAST(order_nbr AS VARCHAR(20)),
  order_date,
  quantity,
  SUBSTRING(payment_terms,1,12),
  CAST(title_nbr AS CHAR(1))
FROM new_sales
WHERE order_date >= 01-JAN-2005
-- Retrieve only the newer records
```

You must specify the columns in the table that will receive data by enclosing their names in parentheses, in a comma-delimited list. This *column_list* can be omitted, but all columns that are defined for the table are then assumed, in their ordinal positions. Any column with an omitted value will be assigned its default value (according to any *DEFAULT* column setting on the table) or, if no *DEFAULT* column setting exists, NULL. The columns in the column list may be in any order, but you may not repeat any column in the list. Furthermore, the columns and their corresponding *value* entries must agree in terms of datatype and size.

The first of the following two examples leaves off the column list, while the second uses only *DEFAULT* values:

```
INSERT INTO authors
VALUES ('111-11-1111', 'Rabbit', 'Jessica', DEFAULT,
      '1717 Main St', NULL, 'CA', '90675', 1)
INSERT INTO temp_details
DEFAULT VALUES
```

The first statement will succeed only if all the values in the value list correspond correctly with the datatypes and size limitations of the columns of the target table. Any inconsistencies will generate an error. The second statement will succeed only as long as defaults have been declared for the columns of the target table, or those columns allow NULL values.



Executing an *INSERT* statement without a column list is a “worst practice,” since the statement may fail if the target table ever changes.

Programming Tips and Gotchas

INSERT statements will *always* fail under the following circumstances:

- When a datatype mismatch occurs between a column and its value
- When a column is defined as *NOT NULL* and the insertion value is NULL
- When a duplicate value is inserted into a *UNIQUE* or *PRIMARY KEY* constraint
- When the inserted values do not meet the requirements of a *CHECK* constraint
- When an inserted value is constrained by a *FOREIGN KEY* constraint because the value is not derived from the declared primary key of another table

The most common error encountered when executing *INSERT* statements is a mismatch between the number of columns and the number of values. If you accidentally leave out a value that corresponds to a column, you are likely to encounter an error that will cause the statement to fail.

INSERT statements also fail when an inserted value is of a datatype that is a mismatch with the column of the target table. For example, an attempt to insert a string like “Hello World” into an integer column would fail. On the other hand, some database platforms automatically and implicitly convert certain datatypes. For example, SQL Server will automatically convert a date value to a character string for insertion into a *VARCHAR* column.

Another common problem encountered with the *INSERT* statement is a size mismatch between a value and its target column. For example, inserting a long string into a *CHAR(5)* target column or inserting a very large integer into a *TINYINT* column can cause problems. Depending on the platform you are using, the size mismatch may cause an outright error and rollback of the transaction, or the database server may simply trim the extra data. Either result is undesirable. Similarly, a problem can arise when an *INSERT* statement attempts to insert a *NULL* value into a target column that does not accept *NULL*s.



Most problems with *INSERT* statements occur because the programmer does not know the target table very well. Make sure you understand the target table or view before writing elaborate *INSERT* statements.

MySQL

MySQL supports several *INSERT* syntax options that engender this platform’s reputation for high speed:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [[database_name.]owner.]table_name [(column1[, ...])]
{VALUES { (value1 | DEFAULT)[, ...] } | select_statement |
SET [ON DUPLICATE KEY UPDATE] column1=value1, column2=value2[, ...]}
```

where:

LOW_PRIORITY | DELAYED | HIGH_PRIORITY

Defers the execution of *INSERT* until no other clients are reading from the table, for *LOW_PRIORITY*. This may result in a long wait. *LOW_PRIORITY* should not be used with MyISAM tables, because it disables concurrent inserts. The *DELAYED* keyword allows the client to continue immediately, even if the *INSERT* has not yet completed. *DELAYED* is ignored with *INSERT...SELECT* and *INSERT...ON DUPLICATE* variations. *HIGH_PRIORITY* merely overrides the effect of servers running in low-priority mode; it does not otherwise boost priority (or processing speed) for queries running normally.

IGNORE

Tells MySQL not to attempt to insert records that would duplicate a value in a primary key or unique key; without this clause, the *INSERT* will fail if such duplication occurs. If a duplicate is encountered while the *IGNORE* clause is in use, the duplicate records are ignored while all the other records are inserted.

SET column=value

An alternate syntax that allows you to specify values for target columns by name.

ON DUPLICATE KEY UPDATE

Causes an *INSERT* operation that would create a duplicate value of in a *UNIQUE* index or *PRIMARY KEY* to update the value of the existing row.



MySQL does *not* support the ANSI-standard *OVERRIDE* clauses.

MySQL trims any portion of a value that has a size or datatype mismatch. Thus, inserting the value “10.23 X” into a decimal datatype column will cause “...X” to be trimmed from the inserted value. If you attempt to insert into a column a numeric value that is beyond the range of the column, MySQL will trim the value. Inserting an illegal time or date value in a column will result in a zero value for the target column. Inserting the string “Hello World” into a *CHAR(5)* column will result in the value being trimmed to just the five characters in “Hello”. This string-trimming feature applies to *CHAR*, *VARCHAR*, *TEXT*, and *BLOB* columns.

MySQL supports a statement called *REPLACE* with similar syntax to *INSERT* that overwrites existing values rather than discarding rows that are duplicates. *REPLACE* is essentially the same as *INSERT...IGNORE*.

Oracle

Oracle’s implementation of the *INSERT* statement allows data insertion into a given table, view, partition, subpartition, or object table. It also supports additional extensions such as inserting records into many tables at once and conditional inserts. The syntax is:

```
-- Standard INSERT statement
INSERT [INTO] {table_name [ [SUB]PARTITION { (prtn_name) | (key_value) } ] |
          (subquery) [WITH {READ ONLY | CHECK OPTION
          [CONSTRAINT constr_name]}] | |
          TABLE (collection) [ (+) ] } [alias]
          [(column1[, ...])]

{VALUES (value1[, ...]) [RETURNING expression1[, ...]
          INTO variable1[, ...]] | |
          select_statement [WITH {READ ONLY |
          CHECK OPTION [CONSTRAINT constr_name]}]}

-- Conditional INSERT statement
INSERT {[ALL | FIRST]} WHEN condition
        THEN standard_insert_statement
        ELSE standard_insert_statement
        [LOG ERRORS [INTO [schema.]table_name] [( expression )]
        [REJECT LIMIT {int | UNLIMITED}]]
```

where:

INSERT [INTO]

Inserts one or more rows into a single table, view, materialized view, or subquery. The *INTO* keyword is optional. You insert a single row using the *VALUES* clause and many rows using a *subquery*.

table_name [[SUB]PARTITION { (prtn_name) | (key_value) }]

Identifies the target into which you will insert data. The target may be a table, view, materialized view, or subquery. To enable you to fully qualify the target, *table_name* can expand to *[schema.]table_name[@db_link]*. You may optionally identify the schema and remote address (via *@db_link*) of the target, but the current schema and local database are assumed if you do not otherwise specify them.

You may also optionally identify the *PARTITION* or *SUBPARTITION* (through the *prtn_name* parameter or a *key_value* for a hash partition or subpartition) into which the record(s) will be inserted, as long as the target is not an object table or object view.

subquery

Instructs Oracle to insert records into the base table or tables of the *subquery*, where the *subquery* is a normally formed *SELECT* statement. Essentially, you're using a subquery to construct a view on the fly, and the effect is the same as inserting into a view. This is the primary means for inserting values into multiple tables at one time. All of the columns defined by the subquery, across all tables, must have a corresponding value to insert, or a failure will occur. Multitable inserts must use the subquery format. The following options apply when using subqueries:

WITH READ ONLY

Indicates that the subqueried table or view cannot be updated until the statement completes.

WITH CHECK OPTION [CONSTRAINT constr_name]

Indicates that you cannot insert into the table or view rows that would not pass the *constr_name* check constraint.

TABLE (collection) [(+)] } [alias]

Tells Oracle that the *collection* should be treated like a standard target (i.e., a table or view), whether it be a subquery, a column, a function, or a collection constructor. In any event, the table collection must return a nested table or VARRAY set of values. Since constructions can be very long, you can provide an optional *alias*. Aliases are not allowed in multitable insert operations.

(column1[, ...])

Specifies the target column(s) into which data will be inserted. If you leave off the list of columns, Oracle assumes that the *VALUES* clause or columns of the subquery will perfectly match the columns of the target. Oracle will return an error if you do not insert a value for any columns marked as *NOT NULL* that do not have defined default values.

VALUES (value1[, ...]) [RETURNING expression1[, ...]] INTO variable1[, ...]]

Inserts values into the target table or tables. As with the ANSI standard, there must be a matching value for every column, though the value can be *DEFAULT* or, if the column accepts *NULLs*, the literal *NULL*. *DEFAULT* is not allowed when inserting into a view. On multitable insert operations, the *VALUES* clause must return a corresponding value for every item in the select list of the subquery. The syntax is as follows:

RETURNING expression1

Retrieves the rows inserted by the operation. The *expression* returned by the statement is often a value being inserted, but it may be another value. For example, you might use the *RETURNING* clause to find the value of an automatically generated primary key. Single-row operations store the results into host variables or PL/SQL variables, while multirow operations store them in bind arrays. You can use *RETURNING* against tables, views with a single base table, and materialized views. The *RETURNING* clause is not allowed with multitable insert operations.

INTO variable1

Specifies the variables that will hold the values returned as a result of the *RETURNING* clause. You must declare a corresponding host variable or PL/SQL variable for each expression in the *RETURNING* clause. You cannot use the *INTO* clause to hold a *LONG* datatype; with remote objects; on views that have *INSTEAD OF* triggers; or with parallel *INSERT*, *UPDATE*, or *DELETE* statements.

ALL

Performs a multitable *INSERT* statement. *ALL* is used only with the *subquery* format. Without a *WHEN* clause, *ALL* unconditionally inserts all the data retrieved by the subquery into the tables defined. With a *WHEN* clause, *ALL* performs conditional insert operations that tell Oracle to evaluate all *WHEN* clauses regardless of the results of any other *WHEN* operation. Each time a *WHEN* clause evaluates as *TRUE*, Oracle executes the corresponding *INTO* clause. Multitable inserts are not parallelized on index-organized tables or bitmap indexed tables. They are not allowed at all when:

- The target is a view or materialized view.
- The target is a remote table.
- The *INSERT* command uses a *TABLE* collection expression.
- The table needs more than 999 total target columns.
- The subquery uses a sequence.

FIRST

Tells Oracle to evaluate the *WHEN* clauses in order and, when it finds the first *TRUE* expression, to execute the corresponding *INTO* clause and skip all other *WHEN* clauses.

WHEN condition THEN standard_insert_statement

Sets a *condition* and, when the condition is *TRUE*, executes the *THEN* insert clause. The value of *condition* is evaluated for each column returned in the result set of the subquery. Up to 127 *WHEN* clauses are allowed.

ELSE standard_insert_statement

Executes when no *WHEN* clause evaluates as *TRUE*.

LOG ERRORS [INTO [schema.]table_name] [(expression)] [REJECT LIMIT {int | UNLIMITED}]

Captures DML errors and logs column values of the affected rows into an error-logging table. *INTO* specifies the name of the error-logging table. When omitted, Oracle inserts the affected rows into a table with a name of **ERR\$**_ prepended to the first 25 characters of the table name. *expression* is a literal string or general SQL expression (such as *TO_CHAR(SYSDATE)*) that you want inserted into the error-logging table. *REJECT LIMIT* allows an upper limit for the total number of errors allowed before terminating the DML operation and rolling back the transaction. (Note that you cannot track errors for *LONG*, *LOB*, or object type columns.)

Oracle allows the standard *INSERT* operations as described in the ANSI implementation section, such as *INSERT...SELECT* and *INSERT...VALUES*. However, it has a great many special variations.

When inserting into tables that have assigned sequences, be sure to use the *<sequence_name>.nextval* function call to insert the next logical number in the sequence.

For example, assume you want to use the `authors_seq` sequence to set the value of `au_id` when inserting a new row into the `authors` table:

```
INSERT authors (au_id, au_lname, au_fname, contract )
  VALUES (authors_seq.nextval, 'Rabbit', 'Jessica', 1)
```

When retrieving values during an *INSERT* operation, check for a one-for-one match between the expressions in the *RETURNING* clause and the variables of the *INTO* clause. The expressions returned by the clause do not necessarily have to be those mentioned in the *VALUES* clause. For example, the following *INSERT* statement places a record into the `sales` table, but places a completely distinct value into a bind variable:

```
INSERT authors (au_id, au_lname, au_fname, contract )
  VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1)
  RETURNING hire_date INTO :temp_hr_dt;
```

Notice that the *RETURNING* clause returns the `hire_date` even though `hire_date` is not one of the values listed in the *VALUES* clause. (In this example, it is reasonable to assume a default value was established for the `hire_date` column.)

An unconditional multitable *INSERT* statement into a lookup table that contains a list of all the approved `jobs` in the company looks like this:

```
INSERT ALL
  INTO jobs(job_id, job_desc, min_lvl, max_lvl)
    VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+1, job_desc, min_lvl, max_lvl)
    VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+2, job_desc, min_lvl, max_lvl)
    VALUES(job_id, job_desc, min_lvl, max_lvl)
  INTO jobs(job_id+3, job_desc, min_lvl, max_lvl)
    VALUES(job_id, job_desc, min_lvl, max_lvl)
  SELECT job_identifier, job_title, base_pay, max_pay
  FROM job_descriptions
  WHERE job_status = 'Active';
```

And just to make things more complex, Oracle allows multitable *INSERT* statements that are conditional:

```
INSERT ALL
  WHEN job_status = 'Active'      INTO jobs
  WHEN job_status = 'Inactive'    INTO jobs_old
  WHEN job_status = 'Terminated'  INTO jobs_cancelled
  ELSE INTO jobs
  SELECT job_identifier, job_title, base_pay, max_pay
  FROM job_descriptions;
```

Note that in the preceding example, you would have to follow each *INTO* clause with a *VALUES* clause if you were skipping *NOT NULL* columns in the target table. The following example shows this syntax:

```
INSERT FIRST
  WHEN job_status = 'Active'
    INTO jobs
    VALUES(job_id, job_desc, min_lvl, max_lvl)
  WHEN job_status = 'Inactive'
    INTO jobs_old
    VALUES(job_id, job_desc, min_lvl, max_lvl)
```

```

WHEN job_status = 'Terminated'
  INTO jobs_cancelled
  VALUES(job_id, job_desc, min_lvl, max_lvl)
WHEN job_status = 'Terminated'
  INTO jobs_outsourced
  VALUES(job_id, job_desc, min_lvl, max_lvl)
ELSE INTO jobs
  VALUES(job_id, job_desc, min_lvl, max_lvl)
SELECT job_identifier, job_title, base_pay, max_pay
  FROM job_descriptions;

```

Notice that in this example, the *FIRST* clause also tells Oracle to execute the first occurrence of `job_status = 'Terminated'` by inserting the records into the `jobs_cancelled` table and skipping the `jobs_outsourced` *INSERT* operation.

Oracle allows you to insert data into a table, partition, or view (also known as the *target*) using either a regular or a *direct-path* *INSERT* statement. In a regular insert, Oracle maintains referential integrity and reuses free space in the target. In a direct-path insert, Oracle appends data at the end of the target table without filling in any free space gaps elsewhere in the table. This method bypasses the buffer cache and writes directly to the datafiles; hence the term “*direct-path*.”



Oracle allows the use of hints to circumvent default query optimization for *INSERT* statements. For example, you can use the `APPEND` hint to ensure that an *INSERT* uses a direct-path approach. Refer to the platform documentation for more details on hints that are usable with *INSERT*.

The direct-path approach enhances performance on long, multirecord insert operations. However, if any of the following are true, Oracle will perform a regular *INSERT* instead of a direct-path *INSERT*:

- The data in the target is altered with an *UPDATE* or *DELETE* statement before the *INSERT* statement, in a single transaction. (*UPDATE* and *DELETE* are allowed after the direct-path *INSERT* statement.)
- The *INSERT* statement is or may become distributed.
- The target contains a *LOB* or object datatype column.
- The target has a clustered index or index-organized table.
- The target has triggers or referential integrity constraints.
- The target is replicated.
- The *ROW_LOCKING* initialization parameter is set to *INTENT*.

In addition, Oracle will not allow you to query (e.g., using *SELECT*) a table later in the same transaction after you perform a direct-path *INSERT* into that table until a *COMMIT* has been performed.

Oracle allows you to use parallel direct-path inserts into multiple tables, but you may only use subqueries to insert the data into the tables, not a standard *VALUES* clause.



Inserting *LOBs* and *BFILEs* is tricky. You should initialize such values to `NULL` before inserting. *RAW* columns are also tricky. If you insert a regular string into a *RAW* column, all future queries against the column will be forced to use a table scan.

PostgreSQL

PostgreSQL supports the ANSI standard for the *INSERT* statement, but it does not support the ANSI SQL clause *OVERRIDE SYSTEM GENERATED VALUES* and has added support for a *RETURNING* clause:

```
INSERT INTO table_name [(column1[, ...])]  
  {[DEFAULT] VALUES | VALUES {(value1[, ...]) | DEFAULT} | SELECT_statement}  
  [RETURNING { * | column_value [AS output_name] [, ...] }]
```

where:

(column1[, ...])

Identifies one or more columns in the target table. The list must be enclosed in parentheses, and commas must separate each item in the list. SQL Server automatically provides values for *IDENTITY* columns, *TIMESTAMP* columns, and columns with *DEFAULT* constraints.

DEFAULT

Tells the *INSERT* statement simply to create a new record using all of the default values specified for the target table.

*RETURNING { * | *column_value* [AS *output_name*] [, ...] }*

Retrieves the rows inserted by the operation. You may return all columns using an asterisk (*), or one or more columns of the table with the heading *output_name*. For example, you might use the *RETURNING* clause to find the value of an automatically generated primary key.

PostgreSQL attempts to perform automatic datatype coercion when the expressions in the *VALUE* clause or the select item list of a subquery do not match the datatypes defined for the target table or view.

SQL Server

SQL Server supports a few extensions to the ANSI standard for *INSERT*. Specifically, it supports several rowset functions (explained below), as well as the capability to insert the results from stored procedures and extended procedures directly into the target table. SQL Server's syntax is:

```
[WITH cte_expression[, ...]]  
INSERT [TOP ( number ) [PERCENT]]  
  [INTO] table_name [(column1[, ...])]  
  [OUTPUT expression INTO {@table_variable | output_table} [ (column_list[, ...  
]) ]]  
  {[DEFAULT] VALUES | VALUES {(value1[, ...]) | select_statement} |  
  EXEC[UTE] proc_name [[@param =] value] [OUTPUT] [, ...]}}
```

where:

WITH cte_expression

Defines the temporary named result set of a common table expression, derived from a *SELECT* statement, for the *DELETE* statement.

*INSERT [INTO] *table_name**

Tells SQL Server that the target is a table, a view, or a rowset function. When inserting into a view, an *INSERT* cannot affect more than one of the base tables in the view, if there are more than one. Rowset functions allow SQL Server to source data from special or external data sources such as XML streams, full-text search file structures (a special structure in SQL Server used to store things like MS Word documents and MS PowerPoint slideshows within the database), or

external data sources (such as an MS-Excel spreadsheet). Examples are shown later in this section. SQL Server currently supports the following *rowset functions* for the *INSERT* statement:

OPENQUERY

Executes a pass-through *INSERT* against a linked server. This is an effective means of performing a nested *INSERT* against a data source that is external to SQL Server. The data source must first be declared as a linked server.

OPENROWSET

Executes a pass-through *INSERT* statement against an external data source. This is similar to *OPENDATASOURCE*, except that *OPENDATASOURCE* only opens the data source; it does not actually pass through an *INSERT* statement. *OPENROWSET* is intended for occasional, ad hoc usage only.

TOP (*number*) [PERCENT]

Indicates that the statement should insert only the specified *number* of rows. If *PERCENT* is specified, only the first *number* percent of the rows are inserted. If *number* is an expression, such as a variable, it must be enclosed in parentheses. The expression should be of the *FLOAT* datatype with a range of 0 to 100 when using *PERCENT*. When not using *PERCENT*, the value of *number* should conform to the rules for the *BIGINT* datatype.

(*column1*[, ...])

Identifies one or more columns in the target table. The list must be enclosed in parentheses, and commas must separate each item in the list. SQL Server automatically provides values for *IDENTITY* columns, *TIMESTAMP* columns, and columns with *DEFAULT* constraints.

OUTPUT *expression* INTO {*@table_variable* | *output_table*} [(*column_list*[, ...])]

Retrieves the rows affected by the command (whereas *INSERT* normally only shows the number of rows affected), placing the rows you specify in *expression* into either a given *table_variable* or *output_table*. If the *column_list* is omitted for the *output_table*, the *output_table* must have the same number of columns as the target table. The *output_table* cannot have triggers, participate in a foreign-key constraint, or have any *CHECK* constraints.

DEFAULT

Tells the *INSERT* statement simply to create a new record using all of the default values specified for the target table.

EXEC[UTE] *proc_name* [[@*param* =] *value*] [OUTPUT][, ...]]

Tells SQL Server to execute a dynamic Transact-SQL statement, a stored procedure, a Remote Procedure Call (RPC), or an extended stored procedure and store the results in a local table. *proc_name* is the name of the stored procedure you wish to execute. You may optionally include any of the parameters of the stored procedure, as identified by *@param* (the at sign is required), assign a *value* to the parameter, and optionally designate the parameter as an *OUTPUT* parameter. The columns returned by the result set must match the datatypes of the columns in the target table.

Although SQL Server automatically assigns values to *IDENTITY* columns and *TIMESTAMP* columns, it does not do so for *UNIQUEIDENTIFIER* columns. Columns of either of the former datatypes can simply be skipped in the *columns* and *values* lists. However, you cannot do that with a *UNIQUEIDENTIFIER* column. Instead, you must use the *NEWID()* function to obtain and insert a globally unique ID (GUID):

```
INSERT INTO guid_sample (global_ID, sample_text, sample_int)
VALUES (NEWID(), 'insert first record','10000')
GO
```

When migrating between platforms, remember that inserting an empty string (‘’) into a SQL Server *TEXT* or *VARCHAR* column results in a zero-length string being stored. This is not the same as a *NULL* value, as some platforms interpret it. When inserting into a table using the *INSERT...SELECT* variant, *WITH* hints are allowed on the subquery as long as you do not use *READPAST*, *NOLOCK*, and *READUNCOMMITTED*.

The following example illustrates the *INSERT...EXEC* statements. It first creates a temporary table called **#ins_exec_container**. Then, the first *INSERT* operation retrieves a listing of the *c:\temp* directory and stores it in the temporary table, while the second *INSERT* executes a dynamic *SELECT* statement:

```
CREATE TABLE #ins_exec_container (result_text
    VARCHAR(300) NULL)
GO
INSERT INTO #ins_exec_container
EXEC master..xp_cmdshell "dir c:\temp"
GO
INSERT INTO sales
EXECUTE ('SELECT * FROM sales_2002_04')
GO
```

This functionality can be very useful when you want to build business logic using Transact-SQL stored procedures; for example, to determine the state of objects in or outside of the database and then act on those results using Transact-SQL.



SQL Server allows the use of hints to circumvent default query optimization for *INSERT* statements. However, this type of tuning is recommended only for the most advanced users. Refer to the vendor documentation for more details on hints that are usable with *INSERT*.

Common table expressions may be used with *SELECT*, *INSERT*, *UPDATE*, and *DELETE* statements, as well as the *CREATE VIEW* statement. Common table expressions offer a means of naming and defining a temporary result set from a *SELECT* statement, even allowing recursive behaviors. When defining a common table expression, you may not use *COMPUTE*, *COMPUTE BY*, *FOR XML*, *FOR BROWSE*, *INTO*, *OPTION*, or *ORDER BY* clauses. Multiple *SELECT* statements are allowed in a common table expression only if they are combined with set operators such as *UNION*, *UNION ALL*, *EXCEPT*, or *INTERSECT*.

The following is a simple *INSERT* statement using a common table expression:

```
WITH direct_reports (Manager_ID, DirectReports) AS
( SELECT manager_ID, COUNT(*)
    FROM hr.employee AS e
    WHERE manager_id IS NOT NULL
    GROUP BY manager_id )
DELETE FROM direct_reports
WHERE DirectReports <= 1;
```

INTERSECT Set Operator

Performing an *INSERT* that shows what records and column values were inserted is easy using the *OUTPUT* clause:

```
INSERT hr.employee
    OUTPUT INSERTED.employee_id, INSERTED.employee_lname, INSERTED.employee_
        fname
        INTO @my_temporary_table_variable
    VALUES ('Insert Error', GETDATE());
```

See Also

DELETE
MERGE
SELECT
UPDATE

INTERSECT Set Operator

The *INTERSECT* set operator retrieves the rows of two or more queries, where the rows of the result sets are identical in both the first and second (and possibly more) queries. In some ways, *INTERSECT* is a lot like an *INNER JOIN* operation (see the *JOIN* section for details).

INTERSECT is in a class of keywords called *set operators*. Other set operators include *EXCEPT* and *UNION*. All set operators are used to simultaneously manipulate the result sets of two or more queries; hence the term “set operators.”

Platform	Command
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

There is technically no limit to the number of queries that you may combine with the *INTERSECT* set operator. The general syntax is:

```
<SELECT statement1>
INTERSECT [ALL | DISTINCT]
[CORRESPONDING [BY (column1, column2, ...)]]]
<SELECT statement2>
INTERSECT [ALL | DISTINCT]
[CORRESPONDING [BY (column1, column2, ...)]]]
...
...
```

Keywords

ALL

Includes duplicate rows from all result sets.

DISTINCT

Drops duplicate rows from all result sets prior to the *INTERSECT* comparison. Columns containing a *NULL* value are considered duplicates. (If neither *ALL* nor *DISTINCT* is used, the *DISTINCT* behavior is the default.)

CORRESPONDING

Specifies that only columns with the same name in both queries are returned, even if both queries use the asterisk shortcut.

BY

Specifies that only the named columns are returned, even if more columns with corresponding names exist in the queries. Must be used with the CORRESPONDING keyword.

Rules at a Glance

There is only one significant rule to remember when using *INTERSECT*: the order and number of columns must be the same in all of the queries.

Also, while the datatypes of the corresponding columns do not have to be identical, they must be compatible (for example, CHAR and VARCHAR are compatible datatypes). By default, the result set will default to the largest of the columns in each ordinal position.

Programming Tips and Gotchas

None of the platforms support the ANSI CORRESPONDING [BY (column1, column2, ...)] clause.



On platforms that do not support *INTERSECT*, substitute a query using *FULL JOIN*.

The ANSI standard evaluates *INTERSECT* as higher priority than other set operators, but not all platforms evaluate set operator precedence the same way. You can explicitly control the precedence of set operators using parentheses. Otherwise, the DBMS might evaluate the expressions either from leftmost to rightmost or from first to last.

According to the standard, only one *ORDER BY* clause is allowed in the entire query. It should be included at the end of the last *SELECT* statement. To avoid column and table ambiguity, be sure to alias each column of each table with the same respective alias. For example:

```
SELECT a.au_lname AS 'lastname', a.au_fname AS 'firstname'  
  FROM authors AS a  
INTERSECT  
SELECT e.emp_lname AS 'lastname', e.emp_fname AS 'firstname'  
  FROM employees AS e  
  ORDER BY lastname, firstname
```

Also, be aware that while your column datatypes may be compatible throughout the queries in the *INTERSECT*, there may be some variation in behavior across the DBMS platforms with regard to varying length of the columns. For example, if the **au_lname** column in the first query is markedly longer than the **emp_lname** column in the second query, different platforms may apply different rules as to which length is used for the final result. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the columns across the tables have different names. In general, the column names from the first query are used.

MySQL

Not supported.

Oracle

Oracle supports the *INTERSECT* and *INTERSECT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
INTERSECT
<SELECT statement2>
INTERSECT
...

```

Oracle does not support the *CORRESPONDING* clause. *INTERSECT DISTINCT* is not supported, but *INTERSECT* is the functional equivalent. Oracle does not support *INTERSECT* on the following types of queries:

- Queries containing columns with *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatypes
- Queries containing a *FOR UPDATE* clause or a *TABLE* collection expression

If the first query in the set operation contains any expressions in the select item list, you should include the *AS* keyword to associate an alias with the column resulting from the expression. Also, only the first query in the set operation may contain an *ORDER BY* clause.

For example, you could find all store IDs that also have sales using this query:

```
SELECT stor_id FROM stores
INTERSECT
SELECT stor_id FROM sales
```

PostgreSQL

PostgreSQL supports the *INTERSECT* and *INTERSECT ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
INTERSECT [ALL]
<SELECT statement2>
INTERSECT [ALL]
...

```

PostgreSQL *does not* support *INTERSECT* or *INTERSECT ALL* on queries with a *FOR UPDATE* clause, nor does it support the *CORRESPONDING* clause. *INTERSECT DISTINCT* is not supported, but *INTERSECT* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *INTERSECT* or *INTERSECT ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the rightmost occurrence of *ORDER BY* or *LIMIT* will be assumed to apply to the entire set operation.

For example, you can find all authors who are also employees and whose last names start with “P”:

```
SELECT a.au_lname
FROM authors AS a
WHERE a.au_lname LIKE 'P%'
INTERSECT
```

```
SELECT e.lname
  FROM employee AS e
 WHERE e.lname LIKE 'W%';
```

SQL Server

SQL Server supports the *INTERSECT* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
INTERSECT [ALL]
<SELECT statement2>
INTERSECT [ALL]
```

The column names of the result set are those returned by the first query. Any column names or aliases referenced in an *ORDER BY* clause must appear in the first query. When using *INTERSECT* (or *EXCEPT*) to compare more than two result sets, each pair of result sets (i.e., each pair of queries) is compared before moving to the next pair in the order of expressions in parentheses first, *INTERSECT* set operators second, and *EXCEPT* and *UNION* last in order of appearance.

Also note that you can use *NOT IN* or *NOT EXISTS* operations in conjunction with a correlated subquery, as alternatives. Refer to the sections on *IN* and *EXISTS* for examples.

See Also

EXCEPT
SELECT
UNION

IS Operator

The *IS* operator determines whether a value is *NULL* or not.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

{*WHERE* | {*AND* | *OR*}} *expression* *IS* [NOT] *NULL*

Keywords

{*WHERE* | {*AND* | *OR*}} *expression* *IS* *NULL*

Returns a Boolean value of *TRUE* if the *expression* is *NULL*, and *FALSE* if the *expression* is not *NULL*. The *expression* evaluated for *NULL* can be preceded by the *WHERE* keyword or *AND* or *OR* keywords.

NOT

Inverses the predicate: the statement will instead return a Boolean *TRUE* if the value of *expression* is not *NULL*, and *FALSE* if the value of *expression* is *NULL*.

Rules at a Glance

Because the value of `NULL` is unknown, you cannot use comparison expressions to determine whether a value is `NULL`. For example, the expressions `X = NULL` and `X <> NULL` cannot be resolved because no value can equal, or not equal, an unknown.

Instead, you must use the `IS NULL` operator. Be sure that you do not put the word `NULL` within quotation marks, because if you do that, the DBMS will interpret the value as the word “`NULL`” and not the special value `NULL`.

Programming Tips and Gotchas

Some platforms support the use of a comparison operator to determine whether an expression is `NULL`. However, all platforms covered by this book now support the ANSI `IS [NOT] NULL` syntax.

Sometimes, checking for `NULL` will make your `WHERE` clause only slightly more complex. For example, rather than a simple predicate to test the value of `stor_id`, as shown here:

```
SELECT stor_id, ord_date
  FROM sales
 WHERE stor_id IN (6630, 7708)
```

you need to add a second predicate to accommodate the possibility that `stor_id` might be `NULL`:

```
SELECT stor_id, ord_date
  FROM sales
 WHERE stor_id IN (6630, 7708)
   OR stor_id IS NULL
```

See Also

```
SELECT
  WHERE
```

JOIN Subclause

The `JOIN` subclause enables you to retrieve rows from two or more logically related tables. You can define many different join conditions and types of joins, though the types of joins supported by the different platforms vary greatly.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL2003 Syntax

```
FROM table [AS alias] {CROSS JOIN |
  [NATURAL] [join_type] JOIN joined_table [[AS] alias]
  { ON join_condition1 [{AND | OR} join_condition2] [...] |
    USING (column1[, ...]) }
  [...]}
```

Keywords**FROM table**

Defines the first table or view in the join.

NATURAL

Specifies that the join (either inner or outer) should be assumed on the tables using all columns of identical name shared between the two tables. Consequently, you should *not* specify join conditions using the **ON** or **USING** clauses. The query will fail if you issue a natural join on two tables that do not contain any columns with the same name(s).

[join_type] JOIN joined_table

Specifies the type of **JOIN** and the second (and any subsequent) table(s) in the join. You may also define an *alias* on any of the *joined_tables*. The join types are:

CROSS JOIN

Specifies the complete cross product of two tables. For each record in the first table, all the records in the second table are joined, creating a *huge* result set. This command has the same effect as leaving off the join condition, and its result set is also known as a “Cartesian product.” *Cross joins are not advisable or recommended.*

[INNER] JOIN

Specifies that unmatched rows in either table of the join should be discarded. If no join type is explicitly defined in the ANSI style, this is the default.

LEFT [OUTER] JOIN

Specifies that all records be returned from the table on the left side of the join statement. If a record returned from the left table has no matching record in the table on the right side of the join, it is still returned. Columns from the right table return NULL values when there is no matching row. It is a good idea to configure all your outer joins as left outer joins (rather than mixing left and right outer joins) wherever possible, for consistency.

RIGHT [OUTER] JOIN

Specifies that all records be returned from the table on the right side of the join statement, even if the table on the left has no matching record. Columns from the left table return NULL values when there is no matching row.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

UNION JOIN

Specifies that all the columns of both tables and every row of both tables will be returned. Any columns that have no value in the corresponding joined table are assigned a NULL value.

[AS] alias

Specifies an alias or shorthand for the joined table. The **AS** keyword is optional when specifying an alias.

ON join_condition

Joins together the rows of the table shown in the **FROM** clause and the rows of the table declared in the **JOIN** clause. You may have multiple **JOIN** statements, all based on a common set of values. These values are usually contained in

columns of the same name and datatype appearing in both of the tables being joined. These columns, or possibly a single column from each table, are called the *join key* or *common key*. Most (but not all) of the time, the join key is the primary key of one table and a foreign key in another table. As long as the data in the columns match, the join can be performed.

join_conditions are syntactically depicted in the following form (note that join types are intentionally excluded in this example):

```
FROM table_name1
JOIN table_name2
  ON table_name1.column1 = table_name2.column2
    [{AND|OR} table_name1.column3 = table_name2.column4]
    [...]
JOIN table_name3
  ON table_name1.columnA = table_name3.columnA
    [{AND|OR} table_name1.column3 = table_name2.column4]
    [...]
  [JOIN...]
```

Use the *AND* operator and the *OR* operator to issue a *JOIN* with multiple conditions. It is also a good idea to use brackets around each pair of joined tables if more than two tables are involved, as this makes reading the query much easier.

USING (column[, ...])

Assumes an equality condition on one or more named *columns* that appear in both tables. The column (or columns) must exist, as named, in both tables. Writing a *USING* clause is a little quicker than writing ...*ON table1.columnA = table2.columnA*, but the results are functionally equivalent.

Rules at a Glance

Joins enable you to retrieve records from two (or more) logically related tables in a single result set. You can use an ANSI *JOIN* (detailed here) to perform this operation, or something called a *theta join*. Theta joins, which use a *WHERE* clause to establish the filtering criteria, are the “old” way to do join operations.

For example, you might have a table called **employee** that tracks information about everyone employed in your company. The **employee** table, however, doesn’t contain extensive information about the job an employee holds; instead, it holds only **job_ids**. All information about the job, such as its **description** and **title**, are stored in a table called **job**. Using a *JOIN*, you can easily return columns from both tables in a single set of records. The following sample queries illustrate the difference between a theta and an ANSI *JOIN*:

```
/* Theta join */
SELECT emp_lname, emp_fname, job_title
FROM employee, jobs
WHERE employee.job_id = jobs. job_id;
/* ANSI join */
SELECT emp_lname, emp_fname, job_title
FROM employee
JOIN jobs ON employee.job_id = jobs.job_id;
```

Whenever you reference multiple columns in a single query, the columns must be unambiguous. In other words, the columns must either be unique to each table or be referenced with a table identifier. In the preceding example, both tables have a **job_id**

column, so references to **job_id** must be qualified with a table identifier (the columns in the query that don't exist in both tables don't need to be qualified by table identifiers). However, queries like this are often very hard to read. The following variation of the previous ANSI join is in better form, because it uses the short, easy-to-read alias **e** to refer to the table:

```
SELECT e.emp_lname, e.emp_fname, j.job_title
  FROM employee AS e
 JOIN jobs AS j ON e.job_id = j.job_id;
```

The previous examples were limited to equi-joins, or joins using equality and an equals sign (**=**). However, most other comparison operators are also allowed: you can perform joins on **>**, **<**, **>=**, **<=**, **><**, and so forth.

You cannot join on large object binary datatypes (e.g., *BLOB*) or any other large object datatypes (e.g., *CLOB*, *NLOB*, etc.). Other datatypes are usually allowed in a join comparison.



Cartesian products (i.e., a join between two or more tables that returns all the data for all the rows in all possible variations) are a *really bad idea*. Refer to the following description of *CROSS JOINS* so you know what they look like, and then avoid them!

Following are examples of each type of join:

CROSS JOIN

Following are some cross join examples. The first is a theta join that simply leaves off the join conditions, the second is written using the *CROSS JOIN* clause, and the final query is similar in concept to the first, with a *JOIN* clause that omits the join conditions:

```
SELECT *
  FROM employee, jobs;
```

```
SELECT *
  FROM employee
CROSS JOIN jobs;
```

```
SELECT *
  FROM employee
    JOIN jobs;
```

INNER JOIN

Following is an inner join written using the newer syntax:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
  FROM authors AS a
 INNER JOIN publishers AS p ON a.city = p.city
 ORDER BY a.au_lname DESC
```

There are lots of authors in the **authors** table, but very few of them have cities that match their publishers' cities in the **publishers** table. For example, the preceding query executed in the **pubs** database on SQL Server produces results like the following:

JOIN Subclause

first name	last name	publisher
Carson	Cheryl	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems

The join is called an *inner join* because only those records that meet the join condition in both tables are said to be “inside” the join. You could also issue the same query, on platforms that support it, by substituting the *USING* clause for the *ON* clause:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
  FROM authors AS a
 INNER JOIN publishers AS p USING (city)
 ORDER BY a.au_lname DESC
```

The results for this above query would be the same.

LEFT [OUTER] JOIN

It is a good idea to configure all your outer joins as left outer joins, for greater consistency, rather than mixing left and right outer joins.

In the following example, we show a *LEFT OUTER JOIN* by asking for the publisher for each author (we could also substitute the *USING* clause for the *ON* clause, as shown in the earlier *INNER JOIN* example):

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
  FROM authors AS a
 LEFT OUTER JOIN publishers AS p ON a.city = p.city
 ORDER BY a.au_lname DESC
```

In this example every author will be returned, along with the publisher’s name where there is a match, or a *NULL* value where there is no match. For example, in the SQL Server *pubs* database, the query returns:

first name	last name	publisher
Yokomoto	Akiko	NULL
White	Johnson	NULL
Stringer	Dirk	NULL
Straight	Dean	NULL
...		

All of the data is returned in the lefthand columns (those from the **authors** table), and any column on the righthand side (those from the **publishers** table) that does not have a match returns a *NULL* value. The result set shows *NULL* values where no data matches in the join.

RIGHT [OUTER] JOIN

A *RIGHT OUTER JOIN* is essentially the same as a *LEFT OUTER JOIN*, except everything is weighted toward the table on the right side of the query. For example, the following query executed in the **pubs** database on SQL Server:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
```

```
FROM authors AS a
RIGHT OUTER JOIN publishers AS p ON a.city = p.city
ORDER BY a.au_lname DESC
```

returns the following result set:

first name	last name	publisher
Carson	Cheryl	Algodata Infosystems
Bennet	Abraham	Algodata Infosystems
NULL	NULL	New Moon Books
NULL	NULL	Binnet & Hardley

All of the data is returned on the righthand side of the query (hence the term “right outer join”), and any columns from the lefthand side that do not have a match return a NULL value. The result set shows NULL values where no data matches in the join.

NATURAL [INNER | {LEFT | RIGHT} [OUTER]] JOIN

Natural joins are a substitute for the *ON* or *USING* clause, so do not use *NATURAL* with those clauses. For example:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
  FROM authors AS a
 NATURAL RIGHT OUTER JOIN publishers AS p
 ORDER BY a.au_lname DESC
```

The preceding query will work the same as the earlier examples, but only if both tables possess a column called *city* and that is the only column that they hold in common. You could similarly perform any of the other types of joins (*INNER*, *FULL*, *OUTER*) using the *NATURAL* prefix.

FULL [OUTER] JOIN

Specifies that all rows from either table be returned, regardless of whether the records match. The result set shows NULL values where no data matches in the join. Note that the *OUTER* keyword is optional. If we take our earlier example query and render it as a *FULL JOIN*, it looks like this:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
  FROM authors AS a
 FULL JOIN publishers AS p ON a.city = p.city
 ORDER BY a.au_lname DESC
```

The result set returned by the query is actually the accumulation of the result sets of issuing separate *INNER*, *LEFT*, and *RIGHT* join queries (some records have been excluded for brevity):

first name	last name	publisher
Yokomoto	Akiko	NULL
White	Johnson	NULL
Stringer	Dirk	NULL
...		
Dull	Ann	NULL
del Castillo	Innes	NULL
DeFrance	Michel	NULL

Carson	Cheryl	Algodata Infosystems
Blotchet-Halls	Reginald	NULL
Bennet	Abraham	Algodata Infosystems
NULL	NULL	Binnet & Hardley
NULL	NULL	Five Lakes Publishin
NULL	NULL	New Moon Books
...		
NULL	NULL	Scootney Books
NULL	NULL	Ramona Publishers
NULL	NULL	GGG&G

As you can see, with a *FULL JOIN* you get some records with the NULLs on the right and data on the left (*LEFT JOIN*), some with all of the data (*INNER JOIN*), and some with NULLs on the left and data on the right (*RIGHT JOIN*).

Programming Tips and Gotchas

If an explicit *join_type* is omitted, an *INNER JOIN* is assumed. Note that there are many types of joins, each with their own rules and behaviors, as described in the preceding section.

In general, you should favor the *JOIN* clause over the *WHERE* clause for describing join expressions. This not only keeps your code cleaner, making it easy to differentiate join conditions from search conditions, but also avoids the possibility of buggy behavior resulting from some platform-specific implementations of outer joins specified using the *WHERE* clause.

In general, we do not recommend the use of labor-saving keywords like *NATURAL*, since the subclause will not automatically update itself when the structures of the underlying tables change. Consequently, statements using these constructs may fail when a table change is introduced without also changing the query.

Not all join types are supported by all platforms, so refer to the following sections for full details on platform-specific join support.



Joins involving more than two tables can be difficult. When joins involve three or more tables, it is a good idea to think of the query as a series of two table joins.

MySQL

MySQL supports most ANSI syntax, except that natural joins are supported only on outer joins, not on inner joins. MySQL's *JOIN* syntax is:

```
FROM table [AS alias]
{[STRAIGHT_JOIN joined_table] |
{ { [INNER] | [CROSS] |
[NATURAL] [ {LEFT | RIGHT | FULL} [OUTER] ] }
JOIN joined_table [AS alias]
{ ON join_condition1 [{AND|OR} join_condition2] [...] } |
USING (column1[, ...]) }
[...]
```

where:

STRAIGHT_JOIN

Forces the optimizer to join tables in the exact order in which they appear in the *FROM* clause.

The *STRAIGHT_JOIN* keyword is functionally equivalent to *JOIN*, except that it forces the join order from left to right. This option was supplied because MySQL might, rarely, join the tables in the wrong order.

Refer to the earlier section “Rules at a Glance” for examples.

Oracle

Oracle fully supports the entire ANSI-standard *JOIN* syntax. However, Oracle only began to support ANSI-standard *JOIN* syntax in version 9. Consequently, any older Oracle SQL code exclusively uses *WHERE* clause joins. Oracle’s old syntax for outer theta joins included adding “(+)” to the column names on the opposite side of the direction of the join. This comes from the fact that the table supplying the NULL value rows in effect has NULL value rows added to it.

For example, the following query does a *RIGHT OUTER JOIN* on the **authors** and **publishers** tables. The old Oracle syntax looks like this:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
  FROM authors a, publishers p
 WHERE a.city(+) = p.city
 ORDER BY a.au_lname DESC
```

while the same query in the ANSI syntax would look like this:

```
SELECT a.au_lname AS 'first name',
       a.au_fname AS 'last name',
       p.pub_name AS 'publisher'
  FROM authors AS a
RIGHT OUTER JOIN publishers AS p ON a.city = p.city
 ORDER BY a.au_lname DESC
```

Refer to the section “Rules at a Glance” for more *JOIN* examples.

Oracle is unique in offering partitioned outer joins, which are useful for filling gaps in result sets due to sparse data storage. For example, assume we store production records in a manufacturing table keyed on day and **product_id**. The table holds a row showing the quantity of each product produced during any day on which it is made, but there are no rows for the days it is not produced. This is considered sparse data, since a list of all rows will not show every day for every product. For calculation and reporting purposes, it’s very useful to be able to create result sets where each product has a row for every day, regardless of whether or not it was manufactured on that day. A partitioned outer join makes it simple to do that, since it lets you define a logical partition and apply an outer join to each partition value. The following example does a partitioned outer join with a **times** table to make sure each **product_id** has the full set of dates in a specified time range:

```
SELECT times.time_id AS time, product_id AS id, quantity AS qty
  FROM manufacturing
 PARTITION BY (product_id)
RIGHT OUTER JOIN times
  ON (manufacturing.time_id = times.time_id)
 WHERE manufacturing.time_id
   BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
     AND TO_DATE('06/10/05', 'DD/MM/YY')
 ORDER BY 2, 1;
```

Here is the output from this query:

time	id	qty
01-OCT-05	101	10
02-OCT-05	101	
03-OCT-05	101	
04-OCT-05	101	17
05-OCT-05	101	23
06-OCT-05	101	
01-OCT-05	102	
02-OCT-05	102	
03-OCT-05	102	43
04-OCT-05	102	99
05-OCT-05	102	
06-OCT-05	102	87

Getting these results without using a partitioned outer join would require much more complex and less efficient SQL.

PostgreSQL

PostgreSQL fully supports the ANSI standard. Refer to the section “Rules at a Glance” for examples.

SQL Server

SQL Server supports *INNER*, *OUTER*, and *CROSS* joins using the *ON* clause. SQL Server does not support *NATURAL* join syntax, nor the *USING* clause. SQL Server’s *JOIN* syntax is:

```
FROM table [AS alias]
{ {[INNER] | [CROSS] | [ {LEFT | RIGHT | FULL} [OUTER] ] }
  JOIN joined_table [AS alias]
    { ON join_condition1 [{AND|OR}
      join_condition2] [...] } }
[...]
```

Refer to the section “Rules at a Glance” for examples.

See Also

SELECT
WHERE

LIKE Operator

The *LIKE* operator enables specified string patterns in *SELECT*, *INSERT*, *UPDATE*, and *DELETE* statements to be matched, specifically in the *WHERE* clause. A specified pattern may include special wildcard characters. The specific wildcards supported vary from platform to platform.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```
WHERE expression [NOT] LIKE string_pattern
      [ESCAPE escape_sequence]
```

Keywords

WHERE *expression* LIKE

Returns a Boolean *TRUE* when the value of *expression* matches the *string_pattern*. The expression may be a column, a constant, a host variable, a scalar function, or a concatenation of any of these. It should not be a user-defined type, nor should it be certain types of *LOBs*.

NOT

Inverses the predicate: the statement returns a Boolean *TRUE* if the value of *expression* does not contain the *string_pattern* and returns *FALSE* if the value of *expression* contains the *string_pattern*.

ESCAPE *escape_sequence*

Allows you to search for the presence of characters that would normally be interpreted as wildcards.

Rules at a Glance

Matching string patterns is easy with *LIKE*, but there are a couple of simple rules to remember:

- All characters, including trailing and leading spaces, are important.
- Differing datatypes may be compared using *LIKE*, but they store string patterns differently. In particular, be aware of the differences between the *CHAR*, *VARCHAR*, and *DATE* datatypes.
- Using *LIKE* may negate indexes or force the DBMS to use alternative, less optimal indexes than a straight comparison operation.

The ANSI standard currently supports two wildcard operators that are supported by all of the platforms covered in this book:

`%`

Matches any string

`_ (underscore)`

Matches any single character

The first query in the following example retrieves any city record with “ville” in its name. The second query returns authors with a first name *not* like Sheryl or Cheryl (or Aheryl, Bheryl, Dheryl, 2heryl, and so forth):

```
SELECT * FROM authors
WHERE city LIKE '%ville%';
SELECT * FROM authors
WHERE au_fname NOT LIKE '_heryl';
```

Some of the platforms support additional wildcard symbols. These are described in the platform-specific sections that follow.

Use of the *ESCAPE* clause allows you to look for wildcard characters in the strings stored in your database. Using this mechanism, you designate a character—typically a character that does not otherwise appear in the pattern string—as your escape character. For example, you might designate the tilde (~) because you know it never appears in the pattern string. Any wildcard character preceded by the escape sequence is then treated not as a wildcard, but rather as the character itself. For example, we can look through the **comments** column of the **sales_detail** table (on SQL Server) to see whether any customers have mentioned a newly introduced discount using this query:

```
SELECT ord_id, comment
  FROM sales_detail
 WHERE comment LIKE '%~%%' ESCAPE '~'
```

In this case, the first and last %s are interpreted as wildcards, but the second % character is interpreted as just that (a % character), because it is preceded by the designated escape sequence.

Programming Tips and Gotchas

The usefulness of *LIKE* is based on the wildcard operators that it supports. *LIKE* returns a Boolean *TRUE* value when the comparison finds one or more matching values.

The default case sensitivity of the DBMS is very important to the behavior of *LIKE*. For example, Microsoft SQL Server is not case sensitive by default (though it can be configured that way). Thus, Microsoft SQL Server will evaluate the strings *DAD* and *dad* to be equal. Oracle, on the other hand, is case sensitive. Thus, on Oracle a comparison of *DAD* and *dad* would show them to be unequal. Here's an example query to better illustrate this point:

```
SELECT *
  FROM authors
 WHERE lname LIKE 'LARS%'
```

This query on Microsoft SQL Server would find authors whose last names are stored as *'larson'* or *'lars'*, even though the search was for the uppercase *'LARS%'*. Oracle, however, would not find *'Larson'* or *'Lars'*, because Oracle performs case-sensitive comparisons.

MySQL

MySQL supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards, as well as the *ESCAPE* clause.

MySQL also supports the special functions of *REGEXP* and *RLIKE*, plus *NOT REGEXP* and *NOT RLIKE* for the evaluation of regular expressions. MySQL, after version 3.23.4, is not case-sensitive by default.

Oracle

Oracle supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards, and *ESCAPE* sequences. Oracle's *LIKE* syntax is as follows:

```
WHERE expression [NOT] {LIKE | LIKEC | LIKE2 |
    LIKE4} string_pattern
    [ESCAPE escape_sequence]
```

The Oracle-specific syntax elements have the following meanings:

LIKEC

Uses UNICODE complete characters

LIKE2

Uses UNICODE USC2 codepoints

LIKE4

Uses UNICODE UCS4 codepoints

Since Oracle is case sensitive, you should enclose the *expression*, the *string_pattern*, or both with the *UPPER* function. That way, you are always comparing apples to apples.

PostgreSQL

PostgreSQL supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards and *ESCAPE* sequences.

PostgreSQL is case sensitive by default but provides the keyword *ILIKE* for case-insensitive pattern matching. You can also use the operators ~~ as an equivalent to *LIKE*, ~~* for *ILIKE*, and !~~ and !~~* for *NOT LIKE* and *NOT ILIKE*, respectively. These are all extensions to the ANSI SQL2003 standard.

For example, the following queries are functionally the same:

```
SELECT * FROM authors
WHERE city LIKE '%ville';
SELECT * FROM authors
WHERE city ~~ '%ville';
```

Since these queries are in lowercase, you might run into a case-sensitivity problem. That is, the queries are looking for a lowercase '%ville', but the table might contain uppercase (and unequal) values such as 'BROWNSVILLE', 'NASHVILLE', and 'HUNTSVILLE'. You can get around this as follows:

```
-- Convert the values to uppercase
SELECT * FROM authors
WHERE city LIKE UPPER('%ville');
-- Perform the pattern match using case insensitivity
SELECT * FROM authors
WHERE city ~~* '%ville';
SELECT * FROM authors
WHERE city ILIKE '%ville';
```

Although beyond the scope of this text, you should be aware that PostgreSQL also supports POSIX regular expressions. See the platform documentation for details.

SQL Server

SQL Server supports the ANSI syntax for *LIKE*. It supports the percent (%) and underscore (_) wildcards, and *ESCAPE* sequences. It also supports the following additional wildcard operators:

[]

Matches any value in the specified set, as in [abc], or range, as in [k-n].

[^]

Matches any characters not in the specified set or range.

MERGE Statement

Using SQL Server's additional wildcard operators, you have some added capabilities. For example, you can retrieve any author with a last name like Carson, Carsen, Karson, or Karsen:

```
SELECT * FROM authors
WHERE au_lname LIKE '[CK]ars[eo]n'
```

or you can retrieve any author with a last name that ends in “arson” or “arsen,” *but is not* Larsen or Larson:

```
SELECT * FROM authors
WHERE au_lname LIKE '[A-Z^L]ars[eo]n'
```



Remember that when you're performing string comparisons with *LIKE*, all characters in the pattern string are significant, including all leading and trailing blank spaces.

See Also

SELECT
UPDATE
DELETE
WHERE

MERGE Statement

The *MERGE* statement is sort of like a *CASE* statement for DML operations. It combines *UPDATE* and *INSERT* statements into a single atomic statement with either/or functionality.

Basically, *MERGE* examines the records of a source table and a target table. If the records exist in both the source and target tables, the records in the target table are updated with the values of the records in the source table, based upon predefined conditions. If records that do not exist in the target table do exist in the source table, they are inserted into the target table. The *MERGE* statement was added in the SQL2003 release of the ANSI standard.

Platform	Command
MySQL	Not supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

```
MERGE INTO {object_name | subquery} [ [AS] alias ]
USING table_reference [ [AS] alias ]
ON search_condition
WHEN MATCHED
    THEN UPDATE SET column = { expression | DEFAULT }[, ...]
WHEN NOT MATCHED
    THEN INSERT [( column[, ...] )] VALUES ( expression[, ...] )
```

Keywords

`MERGE INTO {object_name | subquery}`

Declares the target object of the merge operation. The target object may be a table or updatable view of `object_name`, or it may be a nested table subquery.

`[AS] alias`

Provides an optional alias for the target table.

`USING table_reference`

Declares the source table, view, or subquery of the merge operation.

`ON search_condition`

Specifies the condition or conditions on which a match between the source and target table is evaluated. The syntax is essentially the same as the `ON` subclause of the `JOIN` clause. For example, when merging records from the `new_hire_emp` table into the `emp` table, the clause might look like `ON emp.emp_id = new_hire_emp.emp_id`.

`WHEN MATCHED THEN UPDATE SET column = { expression | DEFAULT }[, ...]`

Declares that if a record from the source table has a matching record in the target table (based on the `search_condition`), one or more specified `columns` of the target table should be updated with the indicated value of `expression`.

`WHEN NOT MATCHED THEN INSERT [(column[, ...])] VALUES (expression[, ...])`

Declares that if a record from the source table does not have a matching record in the target table (based on the `search_condition`), a new record should be inserted into the target table using one or more specified `columns` with the value of `expression`.

Rules at a Glance

The rules for using `MERGE` are straightforward:

- The `WHEN MATCHED` and `WHEN NOT MATCHED` clauses are required, but may not be specified more than once.
- The target table can be a standard updatable table, an updatable view, or an updatable subquery.
- If the `table_reference` is a subquery, enclose it in parentheses.
- The `search_condition` clause should not contain any references to stored procedures or user-defined functions.
- The `search_condition` clause may contain multiple elements using the `AND` or `OR` operators.
- If the column list is omitted from the `WHEN NOT MATCHED` clause, a column list of all the columns in the target table, in ordinal position, is assumed.

Other important rules used by the `MERGE` statement are self-evident. For example, the columns referenced in the `WHEN MATCHED` clause must be updatable.

Programming Tips and Gotchas

The `MERGE` statement is sometimes nicknamed the “upsert” statement. That is because it allows, in a single operation, a set of records to be either inserted into a table or, if they already exist, updated with new values.

The only tricky aspect of the `MERGE` statement is simply getting used to the idea of the either/or processing of the `INSERT` and `UPDATE` statements.

Assume that we have two tables, **EMP** and **NEW_HIRE**. The **EMP** table contains all employees of the company who have successfully completed the mandatory 90-day probationary period at the start of their employment. Employees in the **EMP** table can also have several statuses, such as active, inactive, and terminated. Any new hire to the company is recorded in the **NEW_HIRE** table. After 90 days, they are moved into the **EMP** table like all other regular employees. However, since our company hires college interns every summer, it's very likely that some of our new hires will actually have a record in the **EMP** table from last year with a status of inactive. Using pseudocode, the business problem is summarized as:

```

For each record in the NEW_HIRE table
  Find the corresponding record in the EMP table
  If the record exists in the EMP table
    Update existing data in the EMP table
  Else
    Insert this record into the EMP table
  End If
End For

```

We could write a rather lengthy stored procedure that would examine all the records of the **NEW_HIRE** table and then conditionally perform *INSERT* statements for the entirely new employees or *UPDATE* statements for the returning college interns. However, the following ANSI-standard *MERGE* statement makes this process much easier:

```

MERGE INTO emp AS e
  USING (SELECT * FROM new_hire) AS n
  ON e.empno = n.empno
WHEN MATCHED THEN
  UPDATE SET
    e.ename  = n.ename,
    e.sal    = n.sal,
    e.mgr    = n.mgr,
    e.deptno = n.deptno
WHEN NOT MATCHED THEN
  INSERT ( e.empno, e.ename, e.sal, e.mgr, e.deptno )
  VALUES ( n.empno, n.ename, n.sal, n.mgr, n.deptno );

```

As you can see, the *MERGE* statement is very useful for data-loading operations.

MySQL

Not supported. However, you may use the syntactically and functionally similar *REPLACE* statement to do the same thing as a *MERGE* statement.

Oracle

Oracle supports the *MERGE* statement with only the tiniest variations, which are almost entirely evident by comparing the Oracle syntax diagram against the ANSI-standard syntax diagram:

```

MERGE INTO [schema.]{object_name | subquery} [alias]
  USING [schema.]table_reference [alias]
  ON ( search_condition )
WHEN MATCHED THEN
  UPDATE SET column = { expression | DEFAULT }[, ...]

```

```

WHEN NOT MATCHED THEN
  INSERT ( column[, ...] ) VALUES ( expression[, ...]
[LOG ERRORS [INTO [schema.]table_name] [( expression )]
[REJECT LIMIT { int | UNLIMITED }]]

```

The differences between the ANSI standard and Oracle's implementation include:

- The Oracle implementation does not allow the AS keyword with assigning an alias to the target or source table.
- Oracle requires parentheses around the *search_condition* clause.
- In Oracle the WHEN NOT MATCHED clause requires an insert column list, while the ANSI standard makes it optional.

Oracle supports error logging on the MERGE statement following the syntax LOG ERRORS [INTO [*schema*.]table_name] [(*expression*)] [REJECT LIMIT { int | UNLIMITED }]. This clause captures DML errors and logs column values of the affected rows into an error-logging table. INTO specifies the name of the error-logging table. When omitted, Oracle inserts the affected rows into a table with a name of **ERR\$** prepended to the first 25 characters of the table name. *expression* is a literal string or general SQL expression (such as TO_CHAR(SYSDATE)) that you want inserted into the error-logging table. REJECT LIMIT allows an upper limit for the total number of errors allowed before terminating the DML operation and rolling back the transaction. (Note that you cannot track errors for LONG, LOB, or object type columns.)

Refer to the examples in the earlier “Rules at a Glance” and “Programming Tips and Gotchas” sections for more information.

PostgreSQL

Not supported.

SQL Server

SQL Server supports its own distinctive variant of the MERGE statement, starting in SQL Server 2008. In most ways, it is the same as the SQL3 ANSI standard. The syntax follows:

```

[WITH common_table_expression[, ...]]
MERGE [TOP ( number ) [PERCENT]]
  [INTO] {object_name | subquery} [ [AS] alias ]
  USING ( table_reference ) [ [AS] alias ]
  ON search_condition
  WHEN MATCHED
    THEN { UPDATE SET column = { expression | DEFAULT }[, ...] | DELETE }
  WHEN {[ [TARGET] | SOURCE]} NOT MATCHED
    THEN INSERT [( column[, ...] )] [DEFAULT] VALUES ( expression[, ...] )
  [OUTPUT expression [INTO {@table_variable | output_table}
    [( column_list[, ...] )]]]

```

where:

WITH *cte_expression*

Defines the temporary named result set of a common table expression, derived from a SELECT statement, for the DELETE statement.

TOP (number) [PERCENT]

Indicates that the statement should insert only the specified *number* of rows. If *PERCENT* is specified, only the first *number* percent of the rows are inserted. If the *number* is an expression, such as a variable, it must be enclosed in parentheses. The expression should be of the *FLOAT* datatype with a range of 0 to 100 when using *PERCENT*. When not using *PERCENT*, the value of *number* should conform to the rules for the *BIGINT* datatype.

WHEN {[TARGET] | SOURCE} NOT MATCHED

Specifies the behavior of the transaction when a matching value is not discovered between the source and target tables. When neither keyword is specified, the *TARGET* behavior is assumed, so *WHEN NOT MATCHED* is equivalent to *WHEN TARGET NOT MATCHED*. The *WHEN SOURCE NOT MATCHED* clause is for use with additional search conditions, all of which must be matched for the condition to be considered satisfied. Otherwise, you should use *WHEN [TARGET] NOT MATCHED*. You may have two *WHEN SOURCE NOT MATCHED* clauses that specify different conditions for a *DELETE* operation and an *UPDATE* operation.

OUTPUT expression INTO {@table_variable | output_table} [(column_list[, ...])]

Retrieves the rows affected by the command (whereas *MERGE* normally only shows the number of rows affected), placing the rows you specify in *expression* into either a given *table_variable* or *output_table*. If the *column_list* is omitted for the *output_table*, the *output_table* must have the same number of columns as the target table. The *output_table* cannot have triggers, participate in a foreign-key constraint, or have any *CHECK* constraints.

The *MERGE* statement allows a few variations on simply specifying a column name to update or insert values into. The column may be *{DELETED | INSERTED | from_table_name}.[* | column_name]* or *\$ACTION* (*\$ACTION* is a keyword that outputs the actual *INSERT*, *UPDATE*, or *DELETE* statement used by the *MERGE* statement, depending on the action(s) it performs). SQL Server maintains the same *inserted* and *deleted* pseudotables that are used in triggers to maintain transactional consistency within the *MERGE* statement. Therefore, a column may be referenced in the *OUTPUT* clause using the *inserted* or *deleted* pseudotables. In addition, any *AFTER* triggers declared on the target table may fire according to the *INSERT*, *UPDATE*, or *DELETE* triggers defined upon it and the *INSERT*, *UPDATE*, or *DELETE* transaction initiated by the *MERGE* statement.

See Also

INSERT
JOIN
SELECT
SUBQUERY
UPDATE

OPEN Statement

The *OPEN* statement is one of four commands used in cursor processing, along with *DECLARE*, *FETCH*, and *CLOSE*. Cursors allow you to process queries one row at a time, rather than as a complete set. The *OPEN* statement opens a pre-existing server cursor created with the *DECLARE CURSOR* statement.

Cursors are especially important in relational databases because databases are set-based, while most client-centric programming languages are row-based. Cursors allow programmers and databases to perform operations a single row at a time, while the default behavior of a relational database is to operate on a whole set of records.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Not supported
SQL Server	Supported

SQL2003 Syntax

`OPEN cursor_name`

Keywords

`OPEN cursor_name`

Identifies and opens the previously defined cursor created with the *DECLARE CURSOR* command.

Rules at a Glance

At the highest level, a cursor must be:

1. Created using *DECLARE*
2. Opened using *OPEN*
3. Operated against using *FETCH*
4. Dismissed using *CLOSE*

By following these steps, you create a result set similar to that generated by a *SELECT* statement, except that you can operate against each individual row within the result set.

The following generic SQL example opens a cursor and fetches the first and last names of all of the authors from the *authors* table:

```
DECLARE employee_cursor CURSOR FOR
    SELECT au_lname, au_fname
    FROM pubs.dbo.authors
    WHERE lname LIKE 'K%'
OPEN employee_cursor
FETCH NEXT FROM employee_cursor
BEGIN
    FETCH NEXT FROM employee_cursor
END
CLOSE employee_cursor
```

Programming Tips and Gotchas

The most common error encountered with the *OPEN* statement is failing to close the cursor properly. Although the *OPEN* statement is detailed in isolation here, it should always be managed as a group with the *DECLARE*, *FETCH*, and *CLOSE* statements. You won't get an error message if you fail to close a cursor, but the cursor may continue to hold locks and consume memory and other resources on the server as long

as it is open. If you forget to close your cursors, you could end up creating a problem similar to a memory leak. Each cursor consumes memory until it is closed, so even if you're no longer use the cursors, they're still taking up memory that the database server might otherwise be using elsewhere. It's worth taking a little extra time to make sure that every declared and opened cursor is eventually closed.

Cursors are often used in stored procedures and in batches of procedural code. They are useful when you need to perform actions on individual rows rather than on entire sets of data at a time. But because cursors operate on individual rows and not on sets of data, they are often much slower than other means of accessing data. Make sure that you analyze your approach carefully. Many challenges, such as a convoluted *DELETE* operation or a very complex *UPDATE*, can be solved by using clever *WHERE* and *JOIN* clauses instead of cursors.

MySQL

Fully supports the SQL3 standard.

Oracle

Oracle fully supports the ANSI standard, and it allows parameters to be passed directly into the cursor when it is opened. Do this using the following format:

```
OPEN cursor_name [parameter1[, ...]]
```

PostgreSQL

PostgreSQL does not support the *OPEN CURSOR* statement. Instead, PostgreSQL implicitly opens a cursor when it is created using the *DECLARE* statement.

SQL Server

In addition to the standard *OPEN* statement, SQL Server allows “global” cursors using the following syntax:

```
OPEN [GLOBAL] cursor_name
```

where:

cursor_name

Specifies the name of a cursor (or is a string variable containing the name of a cursor) created earlier with the *DECLARE CURSOR* statement.

GLOBAL

Enables the cursor to be referenced by multiple users, even if they have not explicitly been assigned permissions to the cursor. If this keyword is omitted, a local cursor is assumed.

SQL Server allows you to declare several different kinds of cursors. If a cursor is *INSENSITIVE* or *STATIC*, the *OPEN* statement creates a temporary table to hold the cursor result set. Similarly, if the cursor is declared with the *KEYSET* option, a temporary table is automatically created to hold the keyset.

See Also

CLOSE
DECLARE
FETCH
SELECT

ORDER BY Clause

The *ORDER BY* clause specifies the sort order of the result set retrieved by a *SELECT* statement.

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL2003 Syntax

```
ORDER BY {sort_expression [COLLATE collation_name]
          [ASC | DESC]}[, ...]
```

Keywords

ORDER BY

Specifies the order in which rows should be returned by a query. You should not anticipate a specific ordering if you exclude the *ORDER BY* clause, even if you specify a *GROUP BY* clause and it appears that a sort has been done.

sort_expression

Specifies an item in the query that will help determine the order of the result set. You can have multiple *sort_expressions*. They are usually column names or column aliases from the query; however, they may also be expressions like *(salary * 1.02)*. SQL92 allowed the use of ordinal positions for *sort_expressions*, but this functionality has been deprecated and should not be used in SQL2003 queries.

COLLATE collation_name

Overrides the default collation of the *sort_expression* and applies the *collation_name* to the sort expression for the purposes of the *ORDER BY* clause.

ASC | DESC

Specifies that the *sort_expression* should be returned in either ascending order (ASC) or descending order (DESC).

Rules at a Glance

The *ORDER BY* clause should reference columns as they appear in the select item list of the *SELECT* statement, preferably using their aliases (if aliases exist). For example:

```
SELECT au_fname AS first_name, au_lname AS last_name
  FROM authors
 ORDER BY first_name, last_name
```

The *ORDER BY* clause uses a major-to-minor sort ordering. This means that the result set is ordered by the first column referenced; equal values in the first column are then ordered by the second column, equal values in the second column are ordered by the third column, and so forth.

The individual aspects of a column's ordering—*COLLATE* and *ASC/DESC*—are independent of the other columns in the *ORDER BY* clause. Thus, you could order a result set in ascending order by one column, and then flip the next column and order it in descending order.

ORDER BY Clause

```
SELECT au_fname AS first_name, au_lname AS last_name
  FROM authors
 ORDER BY au_lname ASC, au_fname DESC
```

NULLs are always grouped together (i.e., considered equal) for the purposes of sorting. Depending on your platform, NULLs will be clumped together at the top or at the bottom of the result set. The following query on SQL Server:

```
SELECT title, price
  FROM titles
 ORDER BY price, title
```

provides this result set (edited for brevity):

title	price
Net Etiquette	NULL
The Psychology of Computer Cooking	NULL
The Gourmet Microwave	2.9900
You Can Combat Computer Stress!	2.9900
Life Without Fear	7.0000
Onions, Leeks, and Garlic: Cooking Secrets of the Me	20.9500
Computer Phobic AND Non-Phobic Individuals: Behavior	21.5900
But Is It User Friendly?	22.9500

You can force NULLs to appear at the top or bottom of the result set using *ASC* or *DESC*. Of course, all the non-NULL rows of the result set are also ordered in ascending or descending order.

The ANSI standard for *ORDER BY* also supports for the *sort_expression* the use of columns that are not referenced in the select item list. For example, the following query is valid under SQL2003:

```
SELECT title, price
  FROM titles
 ORDER BY title_id
```

Looking at this example, you can see that although the query does not select *title_id*, that column is the primary *sort_expression*. The result set is returned in *title_id* order even though that column is not selected.

Programming Tips and Gotchas

When using set operators (*UNION*, *EXCEPT*, *INTERSECT*), only the last query may have an *ORDER BY* clause. You should not use *ORDER BY* in subqueries of any type.

A number of behaviors that were supported in SQL92 are deprecated in SQL2003. You should avoid these usages:

References to table aliases

For example, *ORDER BY e.emp_id* should be changed to *ORDER BY emp_id*. If there is an ambiguous column name, use an alias to compensate.

References to ordinal position

Use explicitly defined column aliases to compensate.

You may sort not only on columns, but also on expressions involving columns, or even literals:

```
SELECT SUBSTRING(title,1,55) AS title, (price * 1.15) as price
  FROM titles
 WHERE price BETWEEN 2 and 19
 ORDER BY price, title
```

When sorting on expressions from the select item list, you should use aliases to make the *ORDER BY sort_expression* column references easier.

MySQL

MySQL supports the ANSI standard, except for the *COLLATE* option.

You should not attempt to *ORDER BY* columns of the *BLOB* datatype, because only the first bytes, defined by the *MAX_SORT_LENGTH* setting, will be used in the sort. By default, MySQL sorts *NULL* values as lowest (first) for *ASC* order and highest (last) for *DESC* order.

Oracle

Oracle supports the ANSI standard, except for the *COLLATE* option. It also supports the *SIBLINGS* and *NULLS {FIRST | LAST}* options. Oracle's *ORDER BY* syntax is:

```
ORDER [SIBLINGS] BY
  {sort_expression [ASC | DESC] [NULLS {FIRST | LAST}]}[, ...]
```

where the Oracle-specific keywords are:

ORDER [SIBLINGS] BY sort_expression

Sorts the result set of the query in order of the *sort_expression*(s). A *sort_expression* may be a column name, an alias, an integer indicating a column's ordinal position, or another expression (e.g., *salary * 1.02*). The *ORDER SIBLINGS BY* clause tells Oracle to preserve any ordering specified by a hierarchical query clause (*CONNECT BY*), and to use the sort expression order for ordering of siblings in the hierarchy.

NULLS {FIRST | LAST}

NULLS FIRST and *NULLS LAST* specify that the records containing *NULLS* should appear either first or last, respectively. By default, Oracle places *NULLS* last for ascending-order sorts and first for descending-order sorts.

You can emulate the behavior of the *COLLATE* option for a single session by using the *NLSSORT* function with the *NLS_SORT* parameter. You can also emulate the behavior of the *COLLATE* option for all sessions on the server either explicitly, by using the *NLS_SORT* initialization parameter, or implicitly, with the *NLS_LANGUAGE* initialization parameter.

Oracle continues to support deprecated SQL92 features, such as sorting by ordinal position. However, you should *not* perform an *ORDER BY* on any *LOB* column, nested table, or *VARRAY*.

PostgreSQL

PostgreSQL supports the ANSI SQL2003 standard, with the exception of the *COLLATE* option. It also supports the *USING* extension:

```
ORDER BY {sort_expression1 [ASC | DESC | USING operator]}[, ...]
```

where:

USING operator

Specifies a specific comparison operator. Thus, you may sort by *>*, *<*, *=*, *>=*, *<=*, and so forth. Ascending order is the same as specifying *USING <*, while descending order is the same as *USING >*.

PostgreSQL sorts *NULLs* as the highest values. Thus, *NULLs* will appear at the end when sorting in ascending order, and at the beginning when sorting in descending order.

SQL Server

SQL Server fully supports the ANSI standard, including the `COLLATE` option. For example, the following query retrieves the authors' first names from the `authors` table in the `SQL_Latin1` collation:

```
SELECT au_fname
  FROM authors
 ORDER BY au_fname
  COLLATE SQL_Latin1_general_cp1_ci_as
```

SQL Server continues to support the deprecated SQL92 features of the `ORDER BY` clause, including the ability to specify `ORDER BY` columns by their ordinal positions.

By default, SQL Server sorts `NULL` values higher than all other values.

You should not use `TEXT`, `IMAGE`, or `NTEXT` columns as *sort_expressions* on SQL Server.

See Also

`SELECT`

RELEASE SAVEPOINT Statement

The `RELEASE SAVEPOINT` statement eliminates one or more previously created savepoints in the current transaction.

Platform	Command
MySQL	Supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Not supported

SQL2003 Syntax

`RELEASE SAVEPOINT savepoint_name`

Keywords

`savepoint_name`

Represents a named savepoint (or target specification) created earlier in the transaction with the `SAVEPOINT` statement. The `savepoint_name` must be unique within the transaction.

Rules at a Glance

Use the `RELEASE SAVEPOINT` statement within a transaction to destroy a named savepoint. Any savepoints that were created after the named savepoint will also be destroyed.

To illustrate the behavior of savepoints, the following example code inserts a few records, creates a savepoint named `first_savepoint`, and then releases it:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);
```

```
SAVEPOINT first_savepoint;
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);
RELEASE SAVEPOINT first_savepoint;
COMMIT;
```

In this example, the `first_savepoint` savepoint is destroyed and then all three records are inserted into the `authors` table.

In the next example, we perform the same action but with more savepoints:

```
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('111-11-1111', 'Rabbit', 'Jessica', 1);
SAVEPOINT first_savepoint;
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('277-27-2777', 'Fudd', 'E.P.', 1);
SAVEPOINT second_savepoint;
INSERT authors (au_id, au_lname, au_fname, contract )
VALUES ('366-36-3636', 'Duck', 'P.J.', 1);
SAVEPOINT third_savepoint;
RELEASE SAVEPOINT second_savepoint;
COMMIT;
```

In this example, when we release the savepoint called `second_savepoint` the database actually releases `second_savepoint` and `third_savepoint`, since `third_savepoint` was created after `second_savepoint`.

Once released, a savepoint name can be reused.

Programming Tips and Gotchas

Issuing either a `COMMIT` or a full `ROLLBACK` statement will destroy all open savepoints in a transaction. Issuing a `ROLLBACK TO SAVEPOINT` statement returns the transaction to its state at the specified savepoint; any savepoints declared afterward are nullified.

MySQL

Supports the SQL3 standard syntax.

Oracle

Not supported.

PostgreSQL

Supports the SQL3 standard syntax, although the keyword `SAVEPOINT` is optional:

```
RELEASE [SAVEPOINT] savepoint_name
```

SQL Server

Not supported.

See Also

`ROLLBACK`
`SAVEPOINT`

RETURN Statement

The *RETURN* statement terminates processing within a SQL-invoked function (as opposed to a host-invoked function) or stored procedure and returns the routine's result value.



Some vendors use *RETURNS* instead of *RETURN* (the SQL standard).

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported
PostgreSQL	Supported, with limitations
SQL Server	Supported

SQL2003 Syntax

`RETURN return_parameter_value | NULL`

Keywords

return_parameter_value

Represents a value returned by the routine code. A wide variety of value types are possible.

`NULL`

Terminates the function without returning an actual value.

Rules at a Glance

Use the *RETURN* statement within procedural code to terminate processing. For example, you might create a user-defined function that takes a complex and often-used CASE expression and, when passed a parameter, returns a single, easy-to-understand expression value.

Programming Tips and Gotchas

Although the *RETURN* statement is categorized as a separate command within SQL, it is deeply intertwined with the *CREATE FUNCTION* and *CREATE PROCEDURE* statements. Consequently, the *RETURN* statement is almost always found embedded in one of these other commands. Check each statement's section in this book, or your vendor documentation, to get a more complete understanding of each platform's implementation of *RETURN* within the context of each statement.

MySQL

MySQL supports the ANSI-standard syntax for *RETURN*, excluding the *NULL* keyword:

`RETURNS return_parameter_value`

Oracle

Oracle supports the ANSI-standard syntax for *RETURN*, excluding the *NULL* keyword. (Oracle does support the return of a *NULL* value, just not using the ANSI syntax.) Oracle allows the *RETURN* clause only in user-defined functions and user-defined operators. The returned value cannot be a *LONG*, *LONG RAW*, or *REF* datatype in the *CREATE OPERATOR* statement. PL/SQL user-defined functions fully support Boolean datatypes internally, but you cannot invoke a Boolean UDF from a SQL statement. To hold Boolean values, use *INT* (0 or 1) or *VARCHAR2* ('*TRUE*' or '*FALSE*').

The following example creates a function that returns the value stored in the *proj_rev* variable to the calling session:

```
CREATE FUNCTION project_revenue (project IN varchar2)
  RETURN NUMBER
  AS
    proj_rev NUMBER(10,2);
  BEGIN
    SELECT SUM(DECODE(action,'COMPLETED',amount,0) -
               SUM(DECODE(action,'STARTED',amount,0)) +
               SUM(DECODE(action,'PAYMENT',amount,0))
    INTO proj_rev
    FROM construction_actions
    WHERE project_name = project;
    RETURN (proj_rev);
  END;
```

PostgreSQL

PostgreSQL supports the ANSI-standard syntax for *RETURN*, excluding the *NULL* keyword:

```
RETURNS return_parameter_value
```

PostgreSQL allows you to define user-defined functions using either SQL code or C/C++. PostgreSQL does not currently allow stored procedures, though you can duplicate their functionality with user-defined functions. For the purposes of this discussion, we are interested only in SQL functions.

The output allowed by the *return_parameter_value* may be a base type, a complex type, a *SETOF* type, an *OPAQUE* modifier, or the same as the type of an existing column. The *SETOF* type modifier is used to return a set of items via the *RETURNS* statement, rather than a single data value. The *OPAQUE* modifier indicates that the *RETURNS* statement does not return a value. *OPAQUE* can be used only within triggers.

SQL Server

SQL Server supports the *RETURN* statement with this syntax:

```
RETURN [return_integer_value]
```

The *RETURN* command is typically used in stored procedures or user-defined functions. It causes an immediate and complete exit from the program and, optionally, returns an integer value upon exit. SQL Server procedures implicitly return a zero value if no *RETURN* clause is supplied in the procedure definition. Any commands that follow the *RETURN* statement are ignored.

REVOKE Statement

The *RETURN* statement in the following function returns an integer representing the calculated value:

```
CREATE FUNCTION metric_volume -- Input dimensions in centimeters.  
    (@length decimal(4,1),  
     @width decimal(4,1),  
     @height decimal(4,1) )  
RETURNS decimal(12,3) -- Cubic centimeters.  
AS  
BEGIN  
    RETURN ( @length * @width * @height )  
END  
GO  
REVOKE
```

This example creates a function that returns a calculated value—the metric volume of a space—to the calling session.

See Also

[CREATE/ALTER FUNCTION/PROCEDURE](#)
[CREATE/ALTER TRIGGER](#)

REVOKE Statement

The *REVOKE* statement takes two main forms. The first form of the statement removes specific statement permissions from a user, group, or role. The second form of the statement removes access permissions to specific database objects or resources.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The SQL2003 syntax for *REVOKE* takes this general form:

```
REVOKE { [special_options] | {privilege[, ...] | role[, ...]} }  
ON database_object_name  
FROM grantee_name[, ...]  
[GRANTED BY {CURRENT_USER | CURRENT_ROLE}]  
{CASCADE | RESTRICT}
```

Keywords

special_options

Allows the use of one of three optional *special_options*:

GRANT OPTION FOR

Undoes the *WITH GRANT OPTION* privilege assigned to a user, meaning that the user can no longer grant privileges to other users on the object. That user's own privileges on the object remain intact. (This clause is valid to revoke a privilege, but not to revoke a role.)

HIERARCHY OPTION FOR

Undoes the *WITH HIERARCHY OPTION* privilege that allows a user to *SELECT* not only from the named table, but also all of its subtables. (This clause is valid to revoke a privilege, but not to revoke a role.)

ADMIN OPTION FOR

Undoes the ability to grant a role to other users. (This clause is valid to revoke a privilege, but not to revoke a role.)

privilege

Revokes privileges for a variety of statements, which may all be combined in any order:

ALL PRIVILEGES

Revokes all privileges currently assigned to the named users and/or for the specified database objects. Generally not a recommended approach, since it can encourage sloppy coding.

EXECUTE

Revokes the privileges to execute a routine (i.e., a stored procedure, user-defined function, or method).

SELECT | INSERT | UPDATE | DELETE

Revokes the specific privilege for a given user on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked for *SELECT*, *INSERT*, and *UPDATE* privileges, to limit the action to those columns.

REFERENCES

Revokes the user's privilege to create any constraints or assertions referencing the database object as a parent object. You may add a parenthetical list of the columns on a given table where the privilege will be revoked, to limit the action to those columns.

TRIGGER

Revokes the user's ability to create triggers on the specified tables. As a side effect of the *REVOKE*, any triggers that depend on the privilege are also dropped.

UNDER

Revokes the privilege to create subtypes or typed tables.

USAGE

Revokes the privilege to use a domain, user-defined type, character set, collation, or translation.

role

Revokes a specific, predefined role from the named grantees identified in the *FROM* clause. For example, the database administrator might want to create a role called *Reporter* that has read-only access to several tables. When that role is granted to a user, the user is able to do all of the actions the role may do. See the section on the *GRANT* statement for more details.

ON *database_object_name*

Revokes privileges on a specific, predefined database object identified by the *database_object_name* provided. SQL2003 does not include support for system privileges, but many implementations provide a variation of this clause to revoke such privileges. (The *ON* subclause is not used at all when revoking system privileges or roles.) The *database_object_name* may be one of the following:

REVOKE Statement

```
{ [TABLE] object_name | DOMAIN object_name |  
    COLLATION object_name |  
    CHARACTER SET object_name | TRANSLATION object_name |  
    TYPE object_name | [SPECIFIC] {ROUTINE | FUNCTION |  
        PROCEDURE | METHOD}  
    object_name }
```

FROM *grantee_name*

Names the user(s) or role(s) who will lose the specified privilege(s). The *PUBLIC* keyword may be used to revoke privileges granted to the *PUBLIC* global user list. Multiple grantees can be listed with a comma between each.

FROM { CURRENT_USER | CURRENT_ROLE }

Optionally used to specify who granted the privilege to the user in the first place. For example, when using *FROM CURRENT_USER*, the privilege is revoked only if the current user granted the privilege. Otherwise, the statement will fail. When this clause is omitted, *CURRENT_USER* is the default.

RESTRICT | *CASCADE*

Limits a *REVOKE* operation to only the specified privilege (*RESTRICT*), or causes the specified privilege and all dependent privileges to be revoked (*CASCADE*) and possibly causes objects that depend on the privilege to be dropped. Note that *REVOKE...RESTRICT* will fail if dependent privileges exist. The dependent privileges must be revoked first.

Rules at a Glance

A specific privilege on a specific database object can be revoked for a single user using *REVOKE privilege_name ON object_name FROM grantee_name RESTRICT*. A specific privilege on a specific object may be revoked from all users via the *PUBLIC* global user list.

When revoking privileges from multiple grantees, simply place a comma between each. You may also revoke privileges from one or more grantees and the *PUBLIC* role in a single *REVOKE* statement. (*PUBLIC* is described in detail in the section on the *GRANT* statement. When a *GRANT* is issued to *PUBLIC*, that means everyone has the specified permissions.)

When revoking privileges on a table, the operation may be restricted to the column level by including a list of columns enclosed in parentheses after the table name.

Programming Tips and Gotchas

Most of the platforms isolate privileges at the user and role level. (Remember, a *role* is a group of privileges.) Thus, it's possible for an individual user who is assigned two roles to be granted the same permission three times. In this situation, to revoke that permission you would have to revoke the user's privilege directly and remove the user from both roles.

An important aspect of *REVOKE* (and its sister command *GRANT*) is that certain elements of the command are geared toward object-level permissions, while other options of the command are more oriented toward roles or administrative privileges. Generally, these operations are never combined. (The differences in object-level permissions and administrative privileges are fully explained in the database platform sections that follow.) For example, you might want to revoke certain object-level privileges for a given role (*salespeople*) or for individual users (*e_fudd* and *prince_edward*):

```
REVOKE SELECT ON TABLE authors FROM salespeople RESTRICT;
REVOKE ALL PRIVILEGES ON TABLE sales FROM e_fudd,
    prince_edward CASCADE;
```

A lot of table-level privileges can even be assigned down to the column level. Those privileges can be revoked as follows:

```
REVOKE INSERT(au_id, au_fname, au_lname)
    ON authors FROM e_fudd;
```

The special option clauses (*GRANT OPTION*, *HIERARCHY OPTION*, and *ADMIN OPTION*) all exist to allow you to revoke the ability of users to pass privileges or roles on to other users. However, these clauses do not prevent the first group of users themselves from exercising those privileges. For example, we can issue this command if we no longer want anyone in the role *manager* to grant their *UPDATE* privileges to other users:

```
REVOKE GRANT OPTION FOR UPDATE ON sales FROM manager CASCADE;
```

You can also remove role privileges from a user with the *REVOKE* command:

```
REVOKE manager FROM e_fudd CASCADE;
```

A common best practice is to keep *REVOKE* and *GRANT* statements as self-contained and logical as possible. Under this practice, you should avoid using the *CASCADE* and *ALL PRIVILEGES* clauses, because they perform work for you that may not be immediately evident within the scope of the command.

MySQL

MySQL supports many of the standard keywords of the ANSI standard. Notable exceptions include *TRIGGER*, *EXECUTE*, and *UNDER*. MySQL also has a nice shortcut to allow global assignment or revocation of privileges. Please refer back to the ANSI-standard description for those elements of the command not described below:

```
REVOKE [ { ALL [PRIVILEGES] |
    {SELECT | INSERT | UPDATE} [ (column_name[, ...]) ] | DELETE |
    REFERENCES [ (column_name[, ...]) ] } |
    { [ USAGE ] | [{ALTER | CREATE | DROP}] | [FILE] | [INDEX] | [PROCESS] |
    [RELOAD] | [SHUTDOWN] | [CREATE TEMPORARY TABLES] | [LOCK TABLES] |
    [REPLICATION CLIENT] | [REPLICATION SLAVE] | [SHOW DATABASES] |
    [SUPER] }[, ...]
    ON [object_type] {table_name | * | *.* | database_name.*}
    FROM user_name[, ...]
```

where:

ALL [PRIVILEGES]

Synonymous with *ALL PRIVILEGES*. On MySQL, this includes all privileges that apply at the level indicated by the *ON* clause, except the *GRANT OPTION*. (This includes privileges like *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and others.)

SELECT | INSERT | UPDATE | DELETE

Revoke the ability to read, write, modify, and erase data from tables (and possibly from specific columns of a table), respectively.

REFERENCES

Unimplemented.

USAGE

Revokes all of a user's privileges.

{ALTER | CREATE | DROP} ON *table_name*

Revokes the ability to alter, create, or drop tables and other database objects.

FILE

Revokes the ability to load data from or write data to files using the *SELECT INTO* and *LOAD DATA* commands.

INDEX

Revokes the ability to create or drop indexes.

PROCESS

Revokes the ability to view running processes using *SHOW FULL PROCESSLIST*.

RELOAD

Revokes the ability to invoke the *FLUSH* command.

SHUTDOWN

Revokes the ability to use the *MYSQLADMIN SHUTDOWN* command to kill the server process.

CREATE TEMPORARY TABLES

Revokes the ability to create temporary tables.

LOCK TABLES

Revokes the ability to use the MySQL command *LOCK TABLES* on tables where the user has *SELECT* privileges.

REPLICATION CLIENT

Revokes the privilege to see metadata about replication slaves and masters.

REPLICATION SLAVE

Revokes the right to read binlogs of a replication master from a replication slave.

SHOW DATABASES

Revokes the privilege to execute the MySQL command *SHOW DATABASES*.

SUPER

Revokes the user privilege of opening one connection even if the *MAX_CONNECTIONS* threshold has been reached. Also revokes from the users the privileges to execute these MySQL commands: *CHANGE MASTER*, *KILL*, *MYSQLADMIN DEBUG*, *PURGE [MASTER] LOGS*, and *SET GLOBAL*.

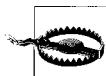
ON [object_type] {table_name | * | ..*database_name.*}

Removes privileges from the specified *table_name*, from all tables within the current database with ***, from all tables in the specified *database_name* with *database_name.**, or from all tables in all databases with *..**. The optional *object_type* may be used to revoke privileges on specific objects (namely, *TABLE*, *FUNCTION*, or *PROCEDURE* object types).

FROM

Removes the privileges of one or more users, separated by commas. Usernames may also include an *@host_name* suffix, if you wish to restrict the revocation to a specific host computer.

REVOKE has size limitations: usernames cannot be longer than 16 characters, while host, database, and database object names can be up to 60 characters long. Usernames may also be linked to a specific host. Refer to the section on the *GRANT* command for more details.



MySQL's implementation of *REVOKE* does not explicitly roll back permissions on objects that are dropped. Thus, it is necessary to explicitly *REVOKE* permissions on a table, even if the table is dropped. For example, let's say you drop a table without issuing a *REVOKE* on the existing permissions. If you later recreate the table, all of the old permissions will still be in place. Similarly, a user's privileges still persist in the database, even after the user has been dropped.

It is also important to note that MySQL allows multiple levels of permissions. Thus, a user might have access to a table from a table-level grant, and have an additional set of permissions to the same table because she has been given global database or server-wide permissions. Thus, you need to be careful when revoking permissions, because global-level permissions might continue to provide to a user permissions you thought you had revoked!

The first of the following commands revokes all privileges on the **sales** table for users *emily* and *dylan*, while the second command revokes *SELECT* privileges for the user *kelly* in the current database:

```
REVOKE ALL PRIVILEGES ON sales FROM emily, dylan;
REVOKE SELECT ON * FROM kelly;
```

The first of the next two commands removes *kelly*'s ability to grant privileges to other users on the **sales** table, while the second command removes privileges that the user *sam* has in the **pubs** database:

```
REVOKE GRANT OPTION ON sales FROM kelly;
REVOKE ALL ON pubs.* FROM sam;
```

Oracle

The *REVOKE* command can be used not only to immediately revoke object and system privileges, but also to revoke a role from a user (or a role from another role). Refer to the section on the *GRANT* statement for more information on the specific object and system privileges supported by the *REVOKE* command.



The two forms of the *REVOKE* command, *REVOKE object_privilege* and *REVOKE system_privilege*, are mutually exclusive. Do not attempt to do both operations in a single statement. Since the complete syntax for both forms of the command is extremely long, refer to the section on *GRANT* for more details on the different *object_privileges* and *system_privileges*.

Oracle provides an interesting twist on privileges. While other database platforms typically allow a user to have more than one context for a set of privileges (based on the individual user and again for any groups to which that user belongs), Oracle takes this one step further. More than one grantor may have given a user a particular privilege on a specific object. When this is the case, *all* grantors must revoke that privilege to effectively rid the user of the privilege. If even one grantor doesn't do this, the user still has the privilege.

In Oracle, use the following syntax with REVOKE:

```
REVOKE { [object_privilege][, ...] |
          [system_privilege] |
          [role] }
  [ON { [schema_name.][object] |
        [DIRECTORY directory_object_name] |
        [JAVA [ { SOURCE | RESOURCE } ] [schema_name.][object]] } ]
  FROM {grantee_name[, ...] | role_name[, ...] | PUBLIC}
  [CASCADE [CONSTRAINTS]] [FORCE];
```

where:

object_privilege

Revokes the specified privilege(s) for the given user(s) or role(s) on the named schema object (for example, a table or view).

ALL [PRIVILEGES]

Revokes all granted privileges on the named schema object. Since *ALL* also includes the *REFERENCES* privilege, you must include the *CASCADE* clause (see *REFERENCES* below).

ALTER

Revokes the privilege to change an existing table using the *ALTER TABLE* statement.

EXECUTE

Revokes the privilege to execute a stored procedure, user-defined function, or package.

INDEX

Revokes the privilege to create indexes on a table.

REFERENCES

Revokes the privilege to define referential integrity constraints. Requires that the *CASCADE CONSTRAINTS* clause also be used.

SELECT | INSERT | DELETE | UPDATE

Revokes the privilege to execute each of the respective SQL commands against the named schema object. Note that *DELETE* privileges are dependent on *SELECT* privileges.

system_privilege

Revokes the specified Oracle system privilege(s), such as *CREATE TRIGGER* or *ALTER USER*, for the given user(s) or role(s). Do not use the *ON* clause with the *REVOKE system_privilege* variation of the statement. Because there are so many system privileges, we won't list them again here. Please refer to the section on Oracle's implementation of the *GRANT* statement for a full list of system privileges.

role

Revokes a user's or role's membership in the named role.

ON

Revokes the named privilege for the given user(s) or role(s) from the named object. Objects include tables, views, sequences, stored procedures, user-defined functions, packages, materialized views, user-defined types, libraries, indextypes, user-defined operators, and synonyms for any of these. For example, you could revoke *SELECT* privileges for a given user on the **scott.authors** table. If you do not include the schema name, Oracle assumes the schema of the current user. Oracle also supports two additional keywords for special cases:

DIRECTORY *directory_object_name*

Identifies a directory object from which the privileges are revoked.

JAVA [{ **SOURCE** | **RESOURCE** }] [*schema_name.*] [*object*]

Identifies a Java source or a resource schema object from which the privileges are revoked.

FROM { *grantee_name* | *role_name* | **PUBLIC** }

Names the user or role that will lose the specified privilege. **PUBLIC** may also be used to revoke privileges granted to the **PUBLIC** role. Multiple grantees can be listed with a comma between each.

CASCADE [**CONSTRAINTS**]

Drops any referential integrity constraints that depend on the privilege being revoked. This clause is needed only when the **REFERENCES** or **ALL** [**PRIVILEGES**] clauses are used.

FORCE

Required to revoke **EXECUTE** object privileges on user-defined type objects with type or table dependencies.

Oracle automatically cascades the revocation from the defined *grantee_name* to all users that received their privileges from the grantee. In addition, any objects created by the grantee that depend on the privilege (such as stored procedures, triggers, views, and packages that depend on a **SELECT** privilege against a certain table) will become invalid.

Users who are granted the **GRANT ANY ROLE** system privilege may revoke any role. The **REVOKE** command can only revoke privileges specifically granted with the **GRANT** command, not privileges available through roles or the operating system. In those cases, you must use the **REVOKE** command to drop the privilege from the role. All users assigned to the role will then lose the privilege.

The following are examples of revoking a role from specific grantee and revoking a system privilege from a role:

```
REVOKE read_only FROM sarah;
REVOKE CREATE ANY SEQUENCE,
      CREATE ANY DIRECTORY FROM read_only;
```

Here's an example that revokes a **REFERENCES** privilege and cascades the revoked privileges:

```
REVOKE REFERENCES
  ON pubs_new_york.emp
  FROM dylan
  CASCADE CONSTRAINTS;
```

Finally, the following example grants all privileges on a specific table, and then revokes a privilege:

```
GRANT ALL PRIVILEGES ON emp TO dylan;
REVOKE DELETE, UPDATE ON emp FROM dylan;
```

PostgreSQL

PostgreSQL offers base-level features of the **REVOKE** statement, primarily for revoking privileges on specific tables, views, and sequences. It does not offer the ANSI-standard special options like **HIERARCHY OPTION FOR** or **ADMIN OPTION FOR**. The syntax is as follows:

```

REVOKE [GRANT OPTION FOR]
{ privileges | { ALL [PRIVILEGES] | {SELECT | INSERT | DELETE | UPDATE} | 
RULE |
  REFERENCES| TRIGGERS | CREATE | USAGE }[, ...] }
ON { [TABLE] | SEQUENCE | DATABASE | FUNCTION | LANGUAGE | SCHEMA | 
TABLESPACE ]
object_name[, ...] }
FROM {grantee_name | GROUP group_name | PUBLIC}[, ...]
[ {CASCADE | RESTRICT} ]

```

where:

REVOKE [GRANT OPTION FOR] *privileges*

Revokes privileges for a variety of statements, which may be combined in any order. *GRANT OPTION FOR* revokes the privilege to grant privileges the named user/role possesses to other users/roles.

ALL [PRIVILEGES]

Shorthand for every privilege the grantor has available to grant. Revokes all privileges currently assigned to the named user(s) and/or group(s) for the specified database object(s). Generally not a recommended approach, since it can encourage sloppy coding.

SELECT | INSERT | DELETE | UPDATE

Revokes the specified privilege for a given user or group on the specified database object, such as a table or view. You may add a parenthetical list of the columns on a given table where the privilege will be revoked.

RULE

Revokes the user or group's privilege to create or drop a rule on a table or view.

REFERENCES

Revokes the user or group's privilege to create or drop foreign key constraints that reference the database object as a parent object.

TRIGGERS

Revokes the user or group's privilege to create or drop triggers on a table.

CREATE

Revokes a user or group's privilege to create objects.

USAGE

Revokes privileges to use a domain, user-defined type, or character set (as opposed to the *UNDER* privilege, which controls the ability to create these objects).

FROM *grantee_name*[, ...] | PUBLIC | GROUP *group_name*

Names the user or role that will lose the specified privilege. *PUBLIC* may also be used to revoke privileges granted to the *PUBLIC* role (a role that implicitly includes all users). Multiple grantees can be listed with a comma between each.

CASCADE | RESTRICT

Limits a *REVOKE* operation to only the specified privilege (*RESTRICT*), or causes the specified privilege and all dependent privileges to be revoked (*CASCADE*). Only used with the *GRANT OPTION FOR* clause. When omitted, PostgreSQL assumes *RESTRICT* by default.

Refer to the section detailing PostgreSQL's implementation of the *GRANT* statement to see a full list of privileges allowed for each object type. You may, in turn, revoke any of these privileges that have been assigned to a user or role.

PostgreSQL's implementation of *REVOKE* is relatively straightforward. The only issue to be aware of is that PostgreSQL treats the term *GROUP* as a synonym for *ROLE*. For example, the following commands remove some privileges from the *PUBLIC* group and from the *READ-ONLY* group:

```
REVOKE ALL PRIVILEGES ON employee FROM public;
REVOKE SELECT ON jobs FROM read-only;
```

PostgreSQL *does not* support privileges on individual columns within a table or view.

When revoking the *GRANT OPTION FOR* clause, you should pay extra attention to dependencies. If you revoke a user's privilege with the *RESTRICT* keyword, the statement will fail if other users depend on the revokee. If you revoke a user's privilege with the *CASCADE* keyword, the statement will revoke privileges not only for the revokee, but for all users that depend on the revokee.

SQL Server

SQL Server implements the *REVOKE* statement as a means to undo any permission settings for a given user or role. This is significant because SQL Server supports an extension statement called *DENY*, which explicitly disallows a user from a specific resource. *REVOKE*, in SQL Server, can be used to undo permissions granted to a user with *GRANT*. If you want to explicitly prevent a user from having a certain privilege, you must use the *DENY* statement.

SQL Server does not support the *HIERARCHY OPTION* and *ADMIN OPTION* clauses of the ANSI standard. Although the *ADMIN OPTION* clause is not supported, a couple of administrative privileges (*CREATE* and *BACKUP*) are supported in SQL Server's version of *REVOKE*. Use the following syntax:

```
REVOKE [GRANT OPTION FOR]
{ [object_privilege][, ...] |
  [system_privilege] }
[ON [class:::][object] [(column[, ...])]]]
{TO | FROM} {grantee_name[, ...] | role[, ...] | PUBLIC | GUEST}
[CASCADE]
[AS {group_name | role_name}]
```

where:

GRANT OPTION FOR

Revokes the privilege of a user to grant a specific privilege to other users.

object_privilege

Revokes privileges for a variety of statements, which may be combined in any order. Refer to the discussion of SQL Server's implementation of the *GRANT* statement for a full list of object privileges.

system_privilege

Revokes privileges to execute commands and perform certain functions. Refer to the discussion of SQL Server's implementation of the *GRANT* statement for a full list of system privileges.

ON *object* [(*column1*, ...)])

Removes the privilege for the user on the named object. If the object is a table or view, you may optionally revoke privileges on specific columns. You may revoke *SELECT*, *INSERT*, *UPDATE*, *DELETE*, and *REFERENCES* privileges on a table or view, but only *SELECT* and *UPDATE* privileges on the columns of a table or view. You may revoke *EXECUTE* privileges on a stored procedure, user-defined function, or extended stored procedure.

/TO | FROM) {*grantee_name* | *role* | PUBLIC | GUEST}

Names the user or role that will lose the specified privilege. PUBLIC may also be used to revoke privileges granted to the PUBLIC role (a role that implicitly includes all users). Multiple grantees can be listed with a comma between each. SQL Server also supports the role known as GUEST, an account used by all users who do not have a user account in the database.

CASCADE

Revokes privileges from users who received their permissions under the *WITH GRANT OPTION* clause. Required when using the *GRANT OPTION FOR* clause.

AS {*group_name* | *role_name*}

Defines the authority by which the privilege is revoked. In certain circumstances, a user may need to temporarily assume the privileges of a group to revoke the specified privileges. To accommodate this scenario, you can use the AS clause to assume the authority of a given group.

The two forms of the REVOKE command, *REVOKE object_privilege* and *REVOKE system_privilege*, are mutually exclusive. Do not attempt to do both operations in a single statement. The key syntactical difference between the two versions is that you should not include the ON clause when revoking system privileges. For example, to drop a system privilege:

```
REVOKE CREATE DATABASE, BACKUP DATABASE FROM dylan, katie
```

If commands were granted to a user with *WITH GRANT OPTION* enabled, the privilege should be revoked using both *WITH GRANT OPTION* and *CASCADE*. For example:

```
REVOKE GRANT OPTION FOR
SELECT, INSERT, UPDATE, DELETE ON titles
TO editors
CASCADE
GO
```

REVOKE can be used only in the current database. Consequently, the *CURRENT_USER* and *CURRENT_ROLE* options in the ANSI variations are always implied. REVOKE is also used to disable any *DENY* settings.



SQL Server additionally supports the *DENY* statement, which is syntactically the same as *REVOKE*. However, it is conceptually different in that *REVOKE* neutralizes a user's privileges, while *DENY* explicitly prohibits a user from having certain privileges. You can use the *DENY* statement to keep a user or role from accessing a privilege, even if it is granted through a role or an explicit *GRANT* statement.

You must use *REVOKE* to remove previously granted *or* denied privileges. For example, say the user *kelly* took an extended leave of absence for maternity leave. During that time, her permissions on the **employee** table were denied. Now that she's returned, we'll lift (*REVOKE*) the denied privileges:

```
DENY ALL ON employee TO kelly
GO
REVOKE ALL ON employee TO kelly
GO
```

In this example, the *REVOKE* command did not remove *kelly*'s privileges; rather, it neutralized the *DENY* statement.

See Also

[GRANT](#)

ROLLBACK Statement

The *ROLLBACK* statement undoes a transaction to its beginning or to a previously declared savepoint. *ROLLBACK* also closes any open cursors.

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with variations

SQL2003 Syntax

```
ROLLBACK [WORK]
[AND [NO] CHAIN]
[TO SAVEPOINT savepoint_name]
```

Keywords

WORK

An optional keyword, but basically just noise.

AND [NO] CHAIN

AND CHAIN tells the DBMS to end the current transaction, but to share the common transaction environment (such as transaction isolation level) with the next transaction. *AND NO CHAIN* simply ends the transaction (and is effectively the same as not including the clause at all).

TO SAVEPOINT *savepoint_name*

Allows the transaction to be rolled back to a named savepoint (that is, a partial rollback) rather than rolling back the entire transaction. The *savepoint_name* may be a literal expression or a variable. If no savepoint of *savepoint_name* is active, the statement will return an error. When the *TO SAVEPOINT* clause is omitted, all cursors are closed. When the *TO SAVEPOINT* clause is included, only the cursors that were open within the savepoint are closed.

In addition to undoing a single data-manipulation operation such as an *INSERT*, *UPDATE*, or *DELETE* statement (or a batch of them), the *ROLLBACK* statement undoes transactions up to the last issued *START TRANSACTION*, *SET TRANSACTION*, or *SAVEPOINT* statement.

Rules at a Glance

ROLLBACK is used to undo a transaction. It can be used to undo explicitly declared transactions that are started with a *START TRAN* statement or a transaction-initiating statement. It can also be used to undo implicit transactions that are started without a *START TRAN* statement. *ROLLBACK* is mutually exclusive of the *COMMIT* statement.

Most people associate commands like *INSERT*, *UPDATE*, and *DELETE* with the term “transaction.” However, transactions encompass a wide variety of commands. The list of commands varies from platform to platform but generally includes any command that alters data or database structures and is logged to the database logging mechanism. According to the ANSI standard, all SQL statements can be undone with *ROLLBACK*.

Programming Tips and Gotchas

The most important gotcha to consider is that some database platforms perform automatic and *implicit* transactions, while others require *explicit* transactions. If you assume a platform uses one method of transactions, you may get bitten. Thus, when moving between database platforms, you should follow a standard, preset way of addressing transactions. We recommend an explicit approach, using *SET TRAN* or *START TRAN* to begin a transaction and *COMMIT* or *ROLLBACK* to end a transaction.

MySQL

MySQL supports a simple and direct rollback mechanism, as well as the ANSI SQL *CHAIN* keyword:

`ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE] TO [SAVEPOINT] savepoint_name`

The optional *RELEASE* clause allows you to specify that MySQL should automatically terminate the client connection when the current transaction has completed.

When creating a table in MySQL, beware that if you might issue a *ROLLBACK* against it, it must be *transaction-safe*. (A transaction-safe table is one declared with the InnoDB or NDB Cluster property. Refer to the *CREATE TABLE* statement for more information.) MySQL allows you to issue transaction-control statements like *COMMIT* and *ROLLBACK* against non-transaction-safe tables, but it will simply ignore them and autocommit as usual. In the case of a *ROLLBACK* against a non-transaction-safe table, the changes will not be rolled back.

MySQL, by default, runs in *AUTOCOMMIT* mode, causing all data modifications to automatically be written to disk. You can turn *AUTOCOMMIT* off by issuing the command *SET AUTOCOMMIT=0*. You can also control the autocommit behavior on a statement-by-statement basis using the *BEGIN* or *BEGIN WORK* command:

```
BEGIN;
SELECT @A:=SUM(salary) FROM employee WHERE job_type=1;
BEGIN WORK;
UPDATE jobs SET summary=@A WHERE job_type=1;
COMMIT;
```

MySQL automatically issues an implicit *COMMIT* upon the completion of any of these statements: *ALTER TABLE*, *BEGIN*, *CREATE INDEX*, *DROP DATABASE*, *DROP TABLE*, *RENAME TABLE*, and *TRUNCATE*.

MySQL supports rollbacks using savepoints starting with version 4.0.14.

Oracle

Oracle supports the ANSI-standard form of the *ROLLBACK* statement with the addition of the *FORCE* clause:

```
ROLLBACK [WORK] {[TO [SAVEPOINT] savepoint_name] | [FORCE 'text']};
```

ROLLBACK clears all data modifications made to the current open transaction (or to a specific, existing savepoint). It also releases all locks held by the transaction, erases all savepoints, undoes all the changes made by the current transaction, and ends the current transaction.

ROLLBACK...TO SAVEPOINT rolls back just the portion of the transaction after the savepoint, erases all savepoints that followed, and releases all table- and row-level locks acquired after the savepoint. Refer to the section on the *SAVEPOINT* statement later in this chapter for more information.

Oracle's implementation closely follows the SQL standard, with the exception of the *FORCE* option. *ROLLBACK FORCE* rolls back an in-doubt, distributed transaction. You must have the *FORCE TRANSACTION* privilege to issue a *ROLLBACK...FORCE* statement. *FORCE* cannot be used with *TO [SAVEPOINT]*. *ROLLBACK...FORCE* affects not the current transaction but the transaction named in 'text', where 'text' must be equal to the local or global transaction ID of the transaction you want to roll back. (These transactions and their ID names are detailed in the Oracle system view **DBA_2PC_PENDING**.)

For example, you might want to roll back your current transaction to the **salary_adjustment** savepoint. These two commands are equivalent:

```
ROLLBACK WORK TO SAVEPOINT salary_adjustment;
ROLLBACK TO salary_adjustment;
```

In the following example, you roll back an in-doubt distributed transaction:

```
ROLLBACK FORCE '45.52.67'
```

PostgreSQL

PostgreSQL supports the basic form of *ROLLBACK*, with savepoints:

```
ROLLBACK { [WORK] | [TRANSACTION] | PREPARED }
[TO [SAVEPOINT] savepoint_name ]
```

where:

WORK | TRANSACTION

Optional keywords that are not required.

PREPARED

Rolls back a transaction that was prepared earlier for two-phase commit (i.e., a *prepared transaction*). Only the superuser or the user who owns the prepared transaction may roll it back. Use the non-SQL3 PostgreSQL command *PREPARE TRANSACTION* to create a transaction for two-phase commit and the command *COMMIT PREPARED* to save the prepared transaction.

TO [SAVEPOINT] *savepoint_name*

Rolls back all commands that were executed after the savepoint was established. The savepoint stays active and can be reused later, if needed.

ROLLBACK clears all data modifications made to the current open transaction. It will return an error if no transaction is currently open. For example, to roll back all open changes, use:

```
ROLLBACK;
```

Be careful with cursors and rolling back to savepoints. For example, a cursor that is opened within a savepoint will be closed if the transaction is rolled back to that savepoint. If a cursor is open but has a savepoint midway through its *FETCH* processes, the cursor position will remain at the position that *FETCH* left it at (meaning that it won't be rolled back). A cursor will remain closed even if rolling back to a savepoint takes you back before the *CLOSE CURSOR* command was issued. Generally, it's a good idea not to mix cursors and savepoints.

Remember that only the command *RELEASE SAVEPOINT* permanently destroys a savepoint; otherwise, it remains active and reusable.

PostgreSQL supports *ABORT* as a synonym of *ROLLBACK*, in the form *ABORT [WORK]* or *ABORT [TRANSACTION]*.

SQL Server

SQL Server supports both the *WORK* and *TRAN* keywords. The only difference between them is that the *ROLLBACK WORK* statement doesn't allow rolling back of a named transaction or to a specific savepoint:

```
ROLLBACK { [WORK] | [TRANSACTION] } {transaction_name | savepoint_name};
```

If *ROLLBACK* is issued alone without the *WORK* or *TRAN* keywords, it rolls back all current open transactions. *ROLLBACK* normally frees locks, but it does not free locks when rolling back to a savepoint.

SQL Server allows you to name a specific *transaction_name* in addition to a specific *savepoint_name*. You may reference them explicitly, or you may use variables within Transact-SQL.

SQL Server does not allow rolling back transactions to a savepoint with a two-phase commit (i.e., a distributed transaction on SQL Server).

ROLLBACK TRANSACTION, when issued in a trigger, undoes all data modifications, including those performed by the trigger, up to the point of the *ROLLBACK* statement. Nested triggers are not executed if they follow a *ROLLBACK* within a trigger; however, any statements within the trigger that follow the rollback are not impacted by the rollback. *ROLLBACK* behaves similarly to *COMMIT* with regard to nesting, resetting the *@@TRANCOUNT* system variable to zero. (Refer to the section on the *COMMIT* statement earlier in this chapter for more information on transaction control within a SQL Server nested trigger.)

Following is a Transact-SQL batch using *COMMIT* and *ROLLBACK* in Microsoft SQL Server. In this example, the code inserts a record into the *sales* table. If the insertion fails the transaction is rolled back, but if the insertion succeeds the transaction is committed:

```
BEGIN TRAN -- Initializes a transaction
-- The transaction itself
INSERT INTO sales
```

```

VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832')
-- Some error-handling in the event of a failure
IF @@ERROR <> 0
BEGIN
    -- Raises an error in the event log and skips to the end
    RAISERROR 50000 'Insert of sales record failed'
    ROLLBACK WORK
    GOTO end_of_batch
END
-- The transaction is committed if no errors are detected
COMMIT TRAN
-- The GOTO label that enables the batch to skip to
-- the end without committing
end_of_batch:
GO
SAVEPOINT sales1

```

See Also

[COMMIT](#)
[RELEASE SAVEPOINT](#)
[SAVEPOINT](#)

SAVEPOINT Statement

This command breaks a transaction into logical breakpoints. Multiple savepoints may be specified within a single transaction. The main benefit of the *SAVEPOINT* command is that transactions may be partially rolled back to a savepoint marker using the *ROLLBACK* command.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported, with limitations

SQL2003 Syntax

`SAVEPOINT savepoint_name`

Keywords

`SAVEPOINT savepoint_name`

Establishes a savepoint named *savepoint_name* within the current transaction.

Some vendors allow duplicate savepoint names within a transaction, but this is not recommended by the ANSI standard.

SQL2003 supports the statement *RELEASE SAVEPOINT *savepoint_name**, enabling an existing savepoint to be eliminated. Refer to the “*RELEASE SAVEPOINT Statement*” section for more information about eliminating an existing savepoint.

Rules at a Glance

Savepoints are established within the scope of the entire transaction in which they are defined, and savepoint names should be unique within their scope. Always make sure to provide easy-to-understand names for your savepoints, because you'll be referencing them later in your programs. Furthermore, make sure you use *BEGIN* and *COMMIT* statements prudently, because accidentally placing a *BEGIN* statement too early or a *COMMIT* statement too late can have a dramatic impact on the way transactions are written to the database.

Programming Tips and Gotchas

Generally, reusing a savepoint name won't produce an error or warning, but a duplicate savepoint name will render the previous savepoint with the same name useless. So, be careful when naming savepoints!

When a transaction is initiated, resources (namely, locks) are expended to ensure transactional consistency. Make sure that your transaction runs to completion as quickly as possible so that the locks are released for others to use.

The following example performs several data modifications and then rolls back to a savepoint:

```
INSERT INTO sales
VALUES('7896','JR3435','Oct 28 1997',25,'Net 60','BU7832');
SAVEPOINT after_insert;
UPDATE sales SET terms = 'Net 90'
WHERE sales_id = '7896';
SAVEPOINT after_update;
DELETE sales;
ROLLBACK TO after_insert;
```

MySQL

MySQL fully supports the ANSI implementation.

Oracle

Oracle fully supports the ANSI implementation.

PostgreSQL

PostgreSQL fully supports the ANSI implementation.

SQL Server

SQL Server does not support the *SAVEPOINT* command. Instead, it uses the *SAVE* command:

```
SAVE TRAN[SACTION] savepoint_name;
```

In addition, rather than declaring the literal name of the savepoint, you can optionally reference a variable containing the name of the savepoint. If you use a variable, it must be of the *CHAR*, *VARCHAR*, *NCHAR*, or *NVARCHAR* datatype.

SQL Server allows you to have many different named savepoints in a single transaction. However, be careful—since SQL Server supports multiple savepoints in a single transaction, it might appear that SQL Server fully supports nested savepoints, but in fact it *does not*. Any time you issue a commit or savepoint in SQL Server, it only commits or rolls back to the last open savepoint.

When the *ROLLBACK TRAN savepoint_name* command is executed, SQL Server rolls the transaction back to the specified savepoint, then continues processing with the next valid Transact-SQL command following the *ROLLBACK* statement. The transaction must ultimately be concluded with a *COMMIT* or a final *ROLLBACK* statement.

See Also

COMMIT
RELEASE SAVEPOINT
ROLLBACK

SELECT Statement

The *SELECT* statement retrieves rows, columns, and derived values from one or many tables of a database.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

The full syntax of the *SELECT* statement is powerful and complex, but can be broken down into these main clauses:

```
SELECT [{ALL | DISTINCT}] select_item [AS alias][, ...]
FROM [ONLY | OUTER]
    {table_name [[AS] alias] | view_name [[AS] alias]}[, ...]
    [ [join_type] JOIN join_condition ]
    [WHERE search_condition] [ {AND | OR | NOT} search_condition [...] ]
    [GROUP BY group_by_expression {group_by_columns | ROLLUP group_by_columns |
        CUBE group_by_columns | GROUPING SETS ( grouping_set_list ) |
        () | grouping_set, grouping_set_list }
    [HAVING search_condition]]
    [ORDER BY {order_expression [ASC | DESC]}[, ...]]
```

Keywords

Each of the keywords shown below, except the *select_item* clause, is discussed in greater detail in the “Rules at a Glance” section that follows:

{ALL | DISTINCT} select_item

Retrieves values that compose the query result set. Each *select_item* may be a literal, an aggregate or scalar function, a mathematic calculation, a parameter or variable, or a subquery, but a *select_item* is most commonly a column from a table or view. A comma must separate each item in a list of such items.

The schema or owner name should be prefixed to a column’s name when it’s extracted from a context outside of the current user’s. If another user owns the table, that user must be included in the column reference. For example, assume the user *jake* needs to access data in the schema *katie*:

```
SELECT emp_id
    FROM katie.employee;
```

SELECT Statement

You can use the asterisk (*) shorthand to retrieve all columns in every table or view shown in the *FROM* clause. It's a good idea to use this shortcut on single-table queries only.

ALL, the default behavior, returns all records that meet the selection criteria. *DISTINCT* tells the database to filter out any duplicate records, thus retrieving only one instance of many identical records.

AS *alias*

Replaces a column heading (when in the *select_item* clause) or a table name or view name (when in the *FROM* clause) with a shorter heading or name. This clause is especially useful for replacing cryptic or lengthy names with short, easy to understand names or mnemonics, and for when the column contains only derived data, so you don't end up with a column called something like **ORA000189x7/0.02**. It is also very useful in self-joins and correlated subqueries where a single query references the same table more than once. When multiple items appear in the *select_item* clause or *FROM* clause, make sure to place the commas *after* the *AS alias* clauses. Also, be careful to always use an alias uniformly once you introduce it into the query.

FROM *table_name*

Lists all of the tables and/or views from which the query retrieves data. Separate table and view names using commas. The *FROM* clause also allows you to assign aliases to long table/or view names or subqueries using the *AS* clause. Using shorter aliases instead of longer table or view names simplifies coding. (Of course, this might thwart the DBA's carefully planned naming conventions, but the alias only lasts for the duration of the query. Refer to the following section, "Rules at a Glance," for more information on aliases.) A *FROM* clause may contain a subquery (refer to the *SUBQUERY* section later in this chapter for details).

ONLY

Specifies that only the rows of the named table or view (and no rows in subtables or subviews) will be retrieved in the result set. When using *ONLY*, be sure to enclose the *table_name* or *view_name* within parentheses. *ONLY* is ignored if the table or view has no subtables or subviews.

OUTER

Specifies that the rows of the named table or view, along with the rows and columns of any and all subtables or subviews, will be retrieved in the result set. Columns of the subtables (or subviews) will be appended to the right, in subtable hierarchy order according to the depth of the subtable. In extensive hierarchies, subtables with common parents are appended in creation order of their types. When using *OUTER*, be sure to enclose the *table_name* or *view_name* within parentheses. *OUTER* is ignored if the table or view has no subtables or subviews.

JOIN *join_condition*

Joins together the result set of the table shown in the *FROM* clause to another table that shares a meaningful relationship based on a common set of values. These values are usually contained in columns of the same name and datatype that appear in both tables being joined. These columns, or possibly a single column from each table, are called the *join key* or *common key*. Most—but not all—of the time, the join key is the primary key of one table and a foreign key in another table. As long as the data in the columns matches, the join can be performed. (Note that joins can also be performed using the *WHERE* clause. This technique is sometimes called a *theta join*.)

join_conditions are most commonly depicted in the form:

```
JOIN table_name2 ON table_name1.column1 <comparison operator>
    table_name2.column1
JOIN table_name3 ON table_name1.columnA <comparison operator>
    table_name3.columnA
[...]
```

When the *comparison_operator* is the equals sign (=), a join is said to be an *equijoin*. However, the comparison operator may be <, >, <=, >=, or even <>.

Use the *AND* operator to issue a *JOIN* with multiple conditions. You can also use the *OR* operator to specify alternate join conditions.

If an explicit *join_type* is omitted, an *INNER JOIN* is assumed. Note that there are many types of joins, each with its own rules and behaviors (as described in the upcoming “Rules at a Glance” section). Also be aware that an alternate approach to the join condition, via the *USING* clause, exists:

USING (column_name[, ...])

Acts as an alternative to the *ON* clause. With this clause, instead of describing the conditions of the join, you simply provide a *column_name* (or column names, separated by commas) that appears in both tables. The database then evaluates the join based on the column (or columns) of *column_name* that appear in both tables. (The column names must be identical in both tables.) In the following example, the two queries produce identical results:

```
SELECT emp_id
FROM employee
LEFT JOIN sales USING (emp_id, region_id);

SELECT emp_id
FROM employee AS e
LEFT JOIN sales AS s
ON e.emp_id = s.emp_id
AND e.region_id = s.region_id;
```

WHERE search_condition

Filters unwanted data from the result set of the query, returning only those records that satisfy the search conditions. A poorly written *WHERE* clause can ruin the performance of an otherwise useful *SELECT* statement, so mastering the nuances of the *WHERE* clause is of paramount importance. *search_conditions* are syntactically depicted in the form:

WHERE [schema.[table_name.]]column operator value

WHERE clauses usually compare the values contained in a *column* of the table. The values of the column are compared using an *operator* of some type (refer to Chapter 2 for more details). For example, a column might equal (=) a given *value*, be greater than (>) a given *value*, or be *BETWEEN* a range of *values*.

WHERE clauses may contain many search conditions concatenated together using the *AND* or *OR* Boolean operators, and parentheses can be used to impact the order of precedence of the search conditions. *WHERE* clauses can also contain subqueries (refer to the section on the *WHERE* subclause later in this chapter).

SELECT Statement

GROUP BY *group_by_expression*

Used in queries that utilize aggregate functions such as *AVG*, *COUNT*, *COUNT DISTINCT*, *MAX*, *MIN*, and *SUM* to group result sets into the categories you define in the *group_by_expression*. The *group_by_expression* of the GROUP BY clause has an elaborate syntax of its own:

[GROUP BY *group_by_expression*

where *group_by_expression* is:

{ (*grouping_column*[, ...]) | ROLLUP (*grouping_column*[, ...]) |
CUBE (*grouping_column*[, ...]) | GROUPING SETS (*grouping_set_list*) |
() | *grouping_set*, *grouping_set_list* }

Refer to the upcoming “Rules at a Glance” section for examples and more information on ROLLUP, CUBE, and GROUPING SETS.

HAVING *search_condition*

Adds search conditions on the results of the GROUP BY clause in a manner similar to the WHERE clause. HAVING does not affect the rows used to calculate the aggregates. HAVING clauses may contain subqueries.

ORDER BY *order_expression* [ASC | DESC]

Sorts the result set in either ascending (ASC) or descending (DESC) order using the *order_expression* specified. The *order_expression* is a list of columns, separated by commas, that you want to sort by.

Rules at a Glance

Each clause of the SELECT statement has a specific use. Thus, it is possible to speak individually of the FROM clause, the WHERE clause, the GROUP BY clause, and so forth. You can get more details and examples of SELECT statements by looking up the entries for each clause of the statement. However, not every query needs every clause. At a minimum, a query needs a SELECT item list and a FROM clause. Because the SELECT clause is so important and offers so many options, we’ve divided this “Rules at a Glance” section into the following detailed subsections:

- Aliases and WHERE clause joins
- The JOIN clause
- The WHERE clause
- The GROUP BY clause
- The HAVING clause
- The ORDER BY clause

Aliases and WHERE clause joins

Column names may need to be prefixed with their database, schema, and table names, particularly when the same column name may appear in more than one table in the query. For example, on an Oracle database, both the **jobs** table and scott’s **employee** table may contain **job_id** columns. The following example joins the **employee** and **jobs** tables using the WHERE clause:

```
SELECT scott.employee.emp_id,  
       scott.employee.fname,  
       scott.employee.lname,  
       jobs.job_desc
```

```

FROM      scott.employee,
          jobs
WHERE     scott.employee.job_id = jobs.job_id
ORDER BY  scott.employee.fname,
          scott.employee.lname

```

You can also use aliases to write such a query more simply and clearly:

```

SELECT  e.emp_id,
        e.fname,
        e.lname,
        j.job_desc
FROM    scott.employee AS e,
        jobs AS j
WHERE   e.job_id = j.job_id
ORDER BY e.fname,
        e.lname

```

These two queries also illustrate the following important rules about *WHERE* clause joins:

1. Use commas to separate multiple elements in the *select_item* list, tables in the *FROM* clause, and items in the *order_expression*.
2. Use the AS clause to define aliases.
3. Use aliases consistently throughout the *SELECT* statement once you define them.

In general, you should favor the *JOIN* clause (explained in a moment) over the *WHERE* clause for describing join expressions. This not only keeps your code cleaner, making it easy to differentiate join conditions from search conditions, but also allows you to avoid the counterintuitive behavior that may result from using the *WHERE* clause for outer joins in some implementations.

The JOIN clause

To perform the same query as in the previous example using an ANSI-style join, list the first table and the keyword *JOIN*, followed by the name of the table to be joined, the keyword *ON*, and the join condition that would have been used in the old-style query. The next example shows the preceding query in ANSI style:

```

SELECT  e.emp_id, e.fname, e.lname, j.job_desc
FROM    scott.employee AS e
JOIN    jobs AS j ON e.job_id = j.job_id
ORDER BY e.fname, e.lname;

```

Alternately, you could use the *USING* clause. Instead of describing the conditions of the join, simply provide one or more *column_names* (separated by commas) that appear in both of the joined tables. The database then evaluates the join based on the column (or columns) of *column_name* that appear in both tables. (The column names must be identical in both tables.) In the following example, the two queries (one using the *ON* clause and one using the *USING* clause) produce identical results:

```

SELECT emp_id
FROM employee
LEFT JOIN sales USING (emp_id, region_id);
SELECT emp_id
FROM employee AS e
LEFT JOIN sales AS s
  ON e.emp_id = s.emp_id
  AND e.region_id = s.region_id;

```

SELECT Statement

You can specify several different types of joins in ANSI style:

CROSS JOIN

Specifies the complete cross product of two tables. For each record in the first table, all the records in the second table are joined, creating a potentially *huge* result set. (Of course, the result set would be small if both tables had only four records each, but imagine if they each had four million!) This command has the same effect as leaving off the join condition and is also known as a “Cartesian product.” We do not recommend using cross joins.

INNER JOIN

Specifies that unmatched rows in either table of the join should be discarded. If no join type is explicitly defined in the ANSI style, this is the default.

LEFT [OUTER] JOIN

Specifies that all records be returned from the table on the left side of the join statement. If a record returned from the left table has no matching record in the table on the right side of the join, it is still returned, and the corresponding column from the right table returns a NULL value. Many professionals recommend configuring outer joins as left joins wherever possible for consistency.

RIGHT [OUTER] JOIN

Specifies that all records be returned from the table on the right side of the join statement, even if the table on the left has no matching record. In this case, the columns from the left table return NULL values.

FULL [OUTER] JOIN

Specifies that all rows from either table be returned, regardless of matching records in the other table. The result set shows NULL values where no matching data exists in the join.



Not all joins are supported by all platforms, so refer to the specific RDBMS sections for full details on platform-specific join support.

The WHERE clause

A poorly written *WHERE* clause can ruin an otherwise beautiful *SELECT* statement, so the nuances of the *WHERE* clause *must be mastered* thoroughly. Here is an example of a typical query and a multipart *WHERE* clause:

```
SELECT  a.au_lname,
        a.au_fname,
        t2.title,
        t2.pubdate
  FROM    authors a
  JOIN    titleauthor t1 ON a.au_id = t1.au_id
  JOIN    titles t2 ON t1.title_id = t2.title_id
 WHERE   (t2.type = 'business' OR t2.type = 'popular_comp')
        AND   t2.advance > 5500
 ORDER BY t2.title
```

In examining this query, note that the parentheses impact the order of processing for the search conditions. You can use parentheses to move search conditions up or down in precedence, just like you would in an algebra equation.



On some platforms, the database's default collation (also known as the *sort order*) impacts how the *WHERE* clause filters results for a query. For example, SQL Server is (by default) *dictionary-order* and *case insensitive*, making no differentiation between "Smith," "smith," and "SMITH." Oracle, however, is *dictionary-order* and *case sensitive*, finding the values "Smith," "smith," and "SMITH" to be unequal.

The *WHERE* clause offers many more specific capabilities than the preceding example illustrates. The following list references some of the more common capabilities of the *WHERE* clause:

NOT

Inverts a comparison operation using the syntax *WHERE NOT expression*. Thus, you might use *WHERE NOT LIKE...* or *WHERE NOT IN...* in a query.

Comparison operators

Compare any set of values, using the operations *<*, *>*, *<>*, *>=*, *<=*, and *=*. For example:

```
WHERE emp_id = '54123'
```

IS NULL or IS NOT NULL conditions

Search for any NULL or NOT NULL values using the syntax *WHERE expression IS [NOT] NULL*.

AND

Merges multiple conditions, returning only those records that meet all conditions, using the *AND* operator. The maximum number of multiple conditions is platform-dependent. For example:

```
WHERE job_id = '12' AND job_status = 'active'
```

OR

Merges alternative conditions, returning records that meet any of the conditions, using the *OR* operator. For example:

```
WHERE job_id = '13' OR job_status = 'active'
```

LIKE

Tells the query to use a pattern-matching string contained within quotation marks. The wildcard symbols supported by each platform are detailed in their individual sections. All platforms support the percent sign (%) for a wildcard symbol. For example, to find any phone number starting with the 415 area code:

```
WHERE phone LIKE '415%'
```

EXISTS

Used only with subqueries, *EXISTS* tests to see whether the subquery data exists. It is typically much faster than a *WHERE IN* subquery. For example, the following query finds all authors who are also employees:

```
SELECT au_lname FROM authors WHERE EXISTS (SELECT last_name FROM employees)
```

BETWEEN

Performs a range check to see whether a value is in between two values (inclusive of those two values). For example:

```
WHERE ytd_sales BETWEEN 4000 AND 9000.
```

SELECT Statement

IN

Performs a test to see whether an expression matches any one value out of a list of values. The list may be literal, as in *WHERE state IN ('or', 'il', 'tn', 'ak')*. The list of values may also be derived using a subquery:

```
WHERE state IN (SELECT state_abbr FROM territories).
```

SOME | ANY

Functions the same as the *EXISTS* operation, though with slightly different syntax. For example, the following query finds all authors who are also employees:

```
SELECT au_lname FROM authors WHERE au_lname = SOME(SELECT last_name FROM employees)
```

ALL

Performs a check to see whether all records in the subquery match the evaluation criteria, and returns *TRUE* when the subquery returns zero rows. For example:

```
WHERE city = ALL (SELECT city FROM employees WHERE emp_id = 54123)
```

The GROUP BY clause

The *GROUP BY* clause (and the *HAVING* clause) is needed only in queries that utilize *aggregate functions*.

The *GROUP BY* clause is used to report an aggregated value for one or more rows returned by a *SELECT* statement based on one or more non-aggregated columns called *grouping columns*. For example, here is a query that counts up how many people we hired each year during the years 1999 through 2004:

```
SELECT hire_year, COUNT(emp_id) AS nbr_emps
FROM employee
WHERE status = 'ACTIVE'
AND hire_year BETWEEN 1999 AND 2004
GROUP BY hire_year;
```

The results are:

hire_year	nbr_emps
1999	27
2000	17
2001	13
2002	19
2003	20
2004	32

Queries using aggregate functions provide many types of summary information. The most common aggregate functions include:

AVG

Returns the average of all non-NULL values in the specified column(s)

AVG DISTINCT

Returns the average of all unique, non-NULL values in the specified column(s)

COUNT

Counts the occurrences of all non-NULL values in the specified column(s)

COUNT DISTINCT

Counts the occurrences of all unique, non-NULL values in the specified column(s)

COUNT(*)

Counts every record in the table

MAX

Returns the highest non-NULL value in the specified column(s)

MIN

Returns the lowest non-NULL value in the specified column(s)

SUM

Totals all non-NULL values in the specified column(s)

SUM DISTINCT

Totals all unique, non-NULL values in the specified column(s)

Some queries that use aggregates return a sole value. Single-value aggregates are known as a *scalar aggregates*. Scalar aggregates do not need a *GROUP BY* clause. For example:

```
--Query
SELECT AVG(price)
FROM titles
--Results
14.77
```

Queries that return both regular column values and aggregate function values are commonly called *vector aggregates*. Vector aggregates use the *GROUP BY* clause and return one or many rows. There are a few rules to follow when using *GROUP BY*:

- Place *GROUP BY* in the proper clause order—after the *WHERE* clause and before the *ORDER BY* clause.
- Include all non-aggregate columns in the *GROUP BY* clause.
- Do not use column aliases in the *GROUP BY* clause, though table aliases are acceptable.

For example, let's suppose you need to know the total purchase amount of several purchases, with an **Order_Details** table that looks like this:

OrderID	ProductID	UnitPrice	Quantity
10248	11	14.0000	12
10248	42	9.8000	10
10248	72	34.8000	5
10249	14	18.6000	9
10249	51	42.4000	40
10250	41	7.7000	10
10250	51	42.4000	35
10250	65	16.8000	15
...			

The following example will give you the results:

```
SELECT OrderID, SUM(UnitPrice * Quantity) AS 'Order Amt'
FROM order_details
WHERE orderid IN (10248, 10249, 10250)
GROUP BY orderid
```

SELECT Statement

The results are:

OrderID	Order Amt
10248	440.0000
10249	1863.4000
10250	1813.0000

We could further refine the aggregations by using more than one grouping column. Consider the following query, which retrieves the average price of our products, grouped first by name and then by size:

```
SELECT name, size, AVG(unit_price) AS 'avg'  
FROM product  
GROUP BY name, size
```

The results are:

Name	Size	avg
Flux Capacitor	small	900
P32 Space Modulator	small	1400
Transmogrifier	medium	1400
Acme Rocket	large	600
Land Speeder	large	6500

In addition, the *GROUP BY* clause supports a number of very important subclauses:

GROUP BY [{*ROLLUP* | *CUBE*}] ([*grouping_column*[, ...]]) [, *grouping_set_list*]

Groups the aggregate values of the result set by one or more grouping columns. (Without *ROLLUP* or *CUBE*, the *GROUP BY* (*grouping_column*[, ...]) clause is the simplest and common form of the *GROUP BY* clause.)

ROLLUP

Produces subtotals for each set of grouping columns as a hierarchical result set, adding subtotal and grand total rows into the result set in a hierarchical fashion. *ROLLUP* operations return one row per grouping column, with *NULL* appearing in the grouping column to show the subtotalized or totaled aggregate value (illustrated in a moment).

CUBE

Produces subtotals and cross-tabulated totals for all grouping columns. In a sense, the *CUBE* clause enables you to quickly return multidimensional result sets from standard relational tables without a lot of programmatic work. *CUBE* is especially useful when working with large amounts of data. Like *ROLLUP*, *CUBE* provides subtotals of the grouping columns, but it also includes subtotal rows for all possible combinations of the grouping columns specified in the query.

GROUP BY GROUPING SETS [{*ROLLUP* | *CUBE*}] ([*grouping_column*[, ...]]) [, *grouping_set_list*]

Enables aggregated groups on several different sets of grouping columns within the same query. This is especially useful when you want to return only a portion of an aggregated result set. The *GROUPING SETS* clause also lets you select which grouping columns to compare, whereas *CUBE* returns all of the grouping columns and *ROLLUP* returns a hierarchical subset of the grouping columns. As the syntax shows, the ANSI standard also allows *GROUPING SETS* to be paired with *ROLLUP* or *CUBE*.

Consider the clauses shown in Table 3-4 and the result sets they return.

Table 3-4. GROUP BY syntax variations

GROUP BY syntax	Returns the following sets
GROUP BY (col_A, col_B, col_C)	(col_A, col_B, col_C)
GROUP BY ROLLUP (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) ()
GROUP BY CUBE (col_A, col_B, col_C)	(col_A, col_B, col_C) (col_A, col_B) (col_A) (col_B, col_C) (col_B) (col_C) ()
GROUP BY GROUPING SETS ((col_A, col_B), (col_A, col_C), (col_C))	Subquery: SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > '01-JAN-2004')

Each type of *GROUP BY* clause returns a different set of aggregated values and, in the case of *ROLLUP* and *CUBE*, totals and subtotals.

The concepts of *ROLLUP*, *CUBE*, and *GROUPING SETS* are much more intuitive when explained by example. In the following example, we query for data summarizing the number of *sales_orders* by *order_year* and by *order_quarter*:

```
SELECT order_year AS year, order_quarter AS quarter,  
       COUNT (*) AS orders  
  FROM order_details  
 WHERE order_year IN (2003, 2004)  
 GROUP BY ROLLUP (order_year, order_quarter)  
 ORDER BY order_year, order_quarter;
```

The results are:

year	quarter	orders
NULL	NULL	648 -- the grand total
2003	NULL	380 -- the total for year 2003
2003	1	87
2003	2	77
2003	3	91
2003	4	125
2004	NULL	268 -- the total for year 2004
2004	1	139
2004	2	119
2004	3	10

Adding grouping columns to the query provides more details (and more subtotaling) in the result set. Now let's modify the previous example by adding a *region* to the query (but since the number of rows increases, we'll only look at the first and second quarters):

SELECT Statement

```
SELECT order_year AS year, order_quarter AS quarter, region,
       COUNT (*) AS orders
  FROM order_details
 WHERE order_year IN (2003, 2004)
   AND order_quarter IN (1,2)
   AND region IN ('USA', 'CANADA')
 GROUP BY ROLLUP (order_year, order_quarter)
 ORDER BY order_year, order_quarter;
```

The results are:

year	quarter	region	orders	
NULL	NULL	NULL	183	-- the grand total
2003	NULL	NULL	68	-- the subtotal for year 2003
2003	1	NULL	36	-- the subtotal for all regions in q1 of 2003
2003	1	CANADA	3	
2003	1	USA	33	
2003	2	NULL	32	-- the subtotal for all regions in q2 of 2004
2003	2	CANADA	3	
2003	2	USA	29	
2004	NULL	NULL	115	-- the subtotal for year 2004
2004	1	NULL	57	-- the subtotal for all regions in q1 of 2004
2004	1	CANADA	11	
2004	1	USA	46	
2004	2	NULL	58	-- the subtotal for all regions in q2 of 2004
2004	2	CANADA	4	
2004	2	USA	54	

The *GROUP BY CUBE* clause is useful for performing multidimensional analyses on aggregated data. Like *GROUP BY ROLLUP*, it returns subtotals, but unlike *GROUP BY ROLLUP*, it returns subtotals combining all of the grouping columns named in the query. (As you will see, it also has the potential to increase the number of rows returned in the result set.)

In the following example, we query for data summarizing the number of **sales_orders** by **order_year** and by **order_quarter**:

```
SELECT order_year AS year, order_quarter AS quarter,
       COUNT (*) AS orders
  FROM order_details
 WHERE order_year IN (2003, 2004)
 GROUP BY CUBE (order_year, order_quarter)
 ORDER BY order_year, order_quarter;
```

The results are:

year	quarter	orders	
NULL	NULL	648	-- the grand total
NULL	1	226	-- the subtotal for q1 of both years
NULL	2	196	-- the subtotal for q2 of both years
NULL	3	101	-- the subtotal for q3 of both years
NULL	4	125	-- the subtotal for q4 of both years
2003	NULL	380	-- the total for year 2003
2003	1	87	
2003	2	77	
2003	3	91	
2003	4	125	

```

2004 NULL    268    -- the total for year 2004
2004 1       139
2004 2       119
2004 3       10

```

The *GROUP BY GROUPING SETS* clause lets you aggregate on more than one group in a single query. For each group set, the query returns subtotals with the grouping column marked as NULL. While the *CUBE* and *ROLLUP* clauses place predefined subtotals into the result set, the *GROUPING SETS* clause allows you to control what subtotals to add to the query. The *GROUPING SETS* clause does not return a grand total.

Using a similar example query to the ones shown with *ROLLUP* and *CUBE*, this time we'll subtotal by **year** and **quarter** and separately by **year**:

```

SELECT order_year AS year, order_quarter AS quarter, COUNT (*) AS orders
  FROM order_details
 WHERE order_year IN (2003, 2004)
 GROUP BY GROUPING SETS ( (order_year, order_quarter), (order_year) )
 ORDER BY order_year, order_quarter;

```

The results are:

```

year quarter orders
-----
2003 NULL    380    -- the total for year 2003
2003 1       87
2003 2       77
2003 3       91
2003 4       125
2004 NULL    268    -- the total for year 2004
2004 1       139
2004 2       119
2004 3       10

```

Another way to think of *GROUPING SETS* is to consider them to be like a *UNION ALL* of more than one *GROUP BY* query that references different parts of the same data. You can tell the database to add subtotals to a *GROUPING SET* by simply adding in the *ROLLUP* or *CUBE* clause according to how you would like subtotaling to occur.

GROUPING SETS can also be concatenated to concisely generate large combinations of groupings. *Concatenated GROUPING SETS* yield the cross product of groupings from each of the sets within a *GROUPING SETS* list. Concatenated *GROUPING SETS* are compatible with *CUBE* and *ROLLUP*, but since they perform a cross product of all *GROUPING SETS*, they will generate a very large number of final groupings from even a small number of concatenated groupings. This is demonstrated in the example in Table 3-5.

Table 3-5. *GROUP BY* syntax variations

GROUP BY syntax	Returns the following sets
GROUP BY (col_A, col_B, col_C)	(col_A, col_B, col_C)
...	...
GROUP BY GROUPING SETS (col_A, col_B) (col_Y, col_Z)	(col_A, col_Y) (col_A, col_Z) (col_B, col_Y) (col_B, col_Z)

SELECT Statement

You can imagine how large the result set would be if the concatenated *GROUPING SETS* contained a large number of groupings! However, the information returned can be very valuable and hard to reproduce.

The *HAVING* clause

The *HAVING* clause adds search conditions on the result of the *GROUP BY* clause. *HAVING* works very much like the *WHERE* clause, but it applies to the *GROUP BY* clause. The *HAVING* clause supports all the same search conditions as the *WHERE* clause shown earlier. For example, using the same query as before, say we now want to find only those jobs that are performed by more than three people:

```
--Query
SELECT  j.job_desc "Job Description",
        COUNT(e.job_id) "Nbr in Job"
  FROM    employee e
  JOIN    jobs j ON e.job_id = j.job_id
 GROUP BY j.job_desc
 HAVING  COUNT(e.job_id) > 3
--Results
Job Description          Nbr in Job
-----
Acquisitions Manager      4
Managing Editor           4
Marketing Manager         4
Operations Manager         4
Productions Manager        4
Public Relations Manager   4
Publisher                  7
```

Note that the ANSI standard does not *require* that an explicit *GROUP BY* clause appear with a *HAVING* clause. For example, the following query against the *employee* table is valid because it has an implied *GROUP BY* clause:

```
SELECT COUNT(dept_nbr)
  FROM employee
 HAVING COUNT(dept_nbr) > 30;
```

Even though it's valid, though, this application for the *HAVING* clause is rather rare.

The *ORDER BY* clause

A result set can be sorted through the *ORDER BY* clause, in accordance with the database's sort order. Each column of the result set may be sorted in either ascending (*ASC*) or descending (*DESC*) order. (Ascending order is the default.) If no *ORDER BY* clause is specified, most implementations return the data either according to the physical order of the data within the table or according to the order of an index utilized by the query. However, when no *ORDER BY* clause is specified, there is no guarantee as to the order of the result set. Following is an example of a *SELECT* statement with an *ORDER BY* clause on SQL Server:

```
SELECT  e.emp_id "Emp ID",
        e.fname "First",
        e.lname "Last",
        j.job_desc "Job Desc"
  FROM    employee e,
          jobs j
```

```

WHERE      e.job_id = j.job_id
      AND      j.job_desc = 'Acquisitions Manager'
ORDER BY e.fname DESC,
      e.lname ASC
  
```

The results are:

Emp ID	First	Last	Job Desc
MIR38834F	Margaret	Rancé	Acquisitions Manager
MAS70474F	Margaret	Smith	Acquisitions Manager
KJJ92907F	Karla	Jablonski	Acquisitions Manager
GHT50241M	Gary	Thomas	Acquisitions Manager

After the result set is pared down to meet the search conditions, it is sorted by the authors' first names in descending order. Where the authors' first names are equal, the results are sorted in ascending order by last name.



You may write an *ORDER BY* clause using columns in the table that do not appear in the select list. For example, you might query all `emp_ids` from the `employee` table, yet *ORDER BY* the employee's first and last name.

Programming Tips and Gotchas

Once you've assigned an alias to a table or view in the *FROM* clause, use it exclusively for all other references to that table or view within the query (in the *WHERE* clause, for example). Do *not* mix references to the full table name and the alias within a single query. You should avoid mixed references for a couple of reasons. First, it is simply inconsistent and makes code maintenance more difficult. Second, some database platforms return errors on *SELECT* statements containing mixed references. (Refer to the section on *SUBQUERY* later in this chapter for special instructions on aliasing within a subquery.)

MySQL, PostgreSQL, and SQL Server support certain types of queries that do not need a *FROM* clause. Use these types of queries with caution, since the ANSI standard requires a *FROM* clause. Queries without a *FROM* clause must be manually migrated either to the ANSI-standard form or to a form that also works on the target database. Certain platforms do not support the ANSI-style *JOIN* clause. Refer to the entry for each clause to fully investigate the varying degrees of support offered by the different database vendors for the various options of the *SELECT* command.

MySQL

MySQL's implementation of *SELECT* includes partial *JOIN* support, the *INTO* clause, the *LIMIT* clause, and the *PROCEDURE* clause. MySQL does *not* support subqueries prior to version 4.0. Its syntax follows:

```

SELECT [DISTINCT | DISTINCTROW | ALL]
      [STRAIGHT_JOIN] [ {SQL_SMALL_RESULT | SQL_BIG_RESULT} ] [SQL_BUFFER_
RESULT]
      [ {SQL_CACHE | SQL_NO_CACHE} ] [SQL_CALC_FOUND_ROWS]
      [HIGH_PRIORITY] select_item AS alias[, ...]
      [INTO {OUTFILE | DUMPFILE | variable[, ...]} 'file_name' options]
      [FROM table_name AS alias[, ...]
      [ { USE INDEX (index1[, ...]) | IGNORE INDEX (index1[, ...]) } ]
  
```

```
[join_type] [JOIN table2] [ON join_condition]
[WHERE search_condition]
[GROUP BY {unsigned_integer | column_name | formula}
  [ASC | DESC][, ...] [WITH ROLLUP]]
[HAVING search_condition]
[ORDER BY {unsigned_integer | column_name | formula} [ASC | DESC][, ...]]
[LIMIT { [offset_position,] number_of_rows } | number_of_rows
  OFFSET offset_position }]
[PROCEDURE procedure_name (param[, ...])]
[ {FOR UPDATE | LOCK IN SHARE MODE}];
```

where:

STRAIGHT_JOIN

Forces the optimizer to join tables in the exact order in which they appear in the *FROM* clause.

SQL_SMALL_RESULT | *SQL_BIG_RESULT*

Tells the optimizer to expect a small or large result set on a *GROUP BY* clause or a *DISTINCT* clause, respectively. MySQL builds a temporary table when a query has a *DISTINCT* or *GROUP BY* clause, and these optional clauses tell MySQL whether to build a fast temporary table in memory (for *SQL_SMALL_RESULT*) or a slower, disk-based temporary table (for *SQL_BIG_RESULT*) to process the worktable.

SQL_BUFFER_RESULT

Forces the result set into a temporary table so that MySQL can free table locks earlier and speed the result set to the client.

SQL_CACHE | *SQL_NO_CACHE*

Controls caching of the result sets of the query. *SQL_CACHE* stores the result set in the query cache, assuming that the result set is cachable and the value of *query_cache_type* is 2 or *DEMAND*. *SQL_NO_CACHE* causes the result set not to reside in the query cache. For queries with *UNIONs* or subqueries or views, *SQL_NO_CACHE* applies everywhere if it is used even for just one query, while *SQL_CACHE* applies only if it appears after the first query.

SQL_CALC_FOUND_ROWS

Calculates how many rows are in the result set (regardless of a *LIMIT* clause), which can then be retrieved using *SELECT FOUND_ROWS()*.

HIGH_PRIORITY

Gives the query a higher priority than statements that modify data within the table. This should be used only for special, high-speed queries.

select_item

Retrieves the expressions or columns listed. Columns may be listed in the format *[database_name.]**[table_name.]**column_name*. If the database and/or table names are left out, MySQL assumes the current database and table.

FROM...*USE INDEX* | *IGNORE INDEX*

Indicates the table from which rows will be retrieved. The table may be described as *[database_name.]**[table_name]*. MySQL will treat the query as a join if more than one table appears in the *FROM* clause. The optional *FORCE INDEX*, *USE INDEX* (*index1*, *index2*, ...), and *IGNORE INDEX* (*index1*, *index2*, ...) clauses enable you to tell MySQL to use or ignore specific indexes on a table, respectively.

INTO {OUTFILE | DUMPFILE | variable[, ...]} 'file_name'

Writes the result set of the query to a file named '*file_name*' on the host file-system with the *OUTPUT* option. The *file_name* must not already exist on the filesystem. The *DUMPFILE* option writes a single continuous line of data without column terminations, line terminations, or escape characters. This option is used mostly for *BLOB* files. Specific rules for using this clause are detailed below. The *INTO variable* clause allows you to list one or more variables (one for each column returned). If using *INTO variable*, do not also use the '*file_name*' keyword.

LIMIT { [offset_position,] number_of_rows] | number_of_rows OFFSET offset_position}

Constrains the number of rows returned by the query, starting at the *offset_position* and returning *number_of_rows*. If only one integer is supplied, the specified number of records is returned, and a default offset (or starting position) of zero is assumed. The alternative syntax *number_of_rows OFFSET* is included for interoperability with PostgreSQL.

PROCEDURE procedure_name (param[, ...])

NAMES a procedure that processes the data in the result set. The procedure is an external procedure (usually C++), not an internal database stored procedure.

FOR UPDATE | LOCK IN SHARE MODE

Issues a write lock on the rows returned by the query (*FOR UPDATE*) for its exclusive use (provided the table is of InnoDB or BDB type), or issues read locks on the rows returned by the query (*LOCK IN SHARE*), such that other users may see the rows but may not modify them.

Keep a couple of rules in mind when using the *INTO* clause. First, the output file cannot already exist, since overwrite functionality is not supported. Second, any file created by the query will be readable by everyone that can connect to the server. (When using *SELECT...INTO OUTFILE*, you can then turn around and use the MySQL command *LOAD DATA INFILE* to quickly load the data.)

You can use the following options to better control the content of the output file when using *SELECT...INTO OUTFILE*:

- *ESCAPED BY*
- *FIELDS TERMINATED BY*
- *LINES TERMINATED BY*
- *OPTIONALLY ENCLOSED BY*

The following example illustrates the use of these optional commands via a MySQL query that returns a result set in a comma-delimited output file:

```
SELECT job_id, emp_id, lname+fname
INTO OUTFILE "/tmp/employees.text"
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY "'"
LINES TERMINATED BY "\n"
FROM employee;
```

MySQL also allows *SELECT* statements without a *FROM* clause when performing simple arithmetic. For example, the following queries are valid on MySQL:

```
SELECT 2 + 2;
SELECT 565 - 200;
SELECT (365 * 2) * 52;
```

For interoperability with Oracle, MySQL also supports selection from the pseudotable called **dual**:

```
SELECT 565 - 200 FROM dual;
```

MySQL is very fluid in the way it supports joins. You can use several different syntaxes to perform a join; for example, you can explicitly declare a join in a query using the *JOIN* clause, but then show the join condition in the *WHERE* clause. The other platforms force you to pick one method or the other and do not allow you to mix them in a single query. However, we think it's bad practice to mix methods, so our examples use ANSI *JOIN* syntax.

MySQL supports the following types of *JOIN* syntax:

```
FROM table1, table2
FROM table1 {
    [CROSS] JOIN table2 |
    STRAIGHT_JOIN table2 |
    INNER JOIN table2 [ {ON join_condition |
        USING (column_list)} ] |
    LEFT [OUTER] JOIN table2 {ON join_condition |
        USING (column_list)} |
    NATURAL [LEFT [OUTER]] JOIN table2 |
    RIGHT [OUTER] JOIN table2 [ {ON join_condition |
        USING (column_list)} ]|
    NATURAL [RIGHT [OUTER]] JOIN table2 }
```

where:

[CROSS] JOIN table2

Retrieves all records from both *table1* and *table2*. Refer to the section on the *JOIN* subclause, earlier in this chapter, for more information on the different types of joins.

STRAIGHT_JOIN table2

Retrieves all records of both *table1* and *table2*, as any other *JOIN* statement would, but with some special behavior. Normally, MySQL evaluates the table order of a join operation based on choices made by the query optimizer. However, the optimizer can occasionally choose an ineffective table order. The *STRAIGHT_JOIN* keyword forces the optimizer to evaluate the tables in the order listed in the query.

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* is semantically equivalent to an inner join. When using *FROM table1, table2*, be sure include the join condition(s) in the *WHERE* clause.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause that names all columns that exist in both tables. (An example of the *USING* clause appears later.)

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all records in a table even when there are no counterparts in the joined table.

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table, regardless of whether there is a matching record in the leftmost table. A *RIGHT JOIN* is nothing more than a *LEFT JOIN* written from the opposite direction. As a best practice, write all outer join queries as either *LEFT* or *RIGHT OUTER JOIN* queries, but do not use both types, since they require a shift in thinking to understand at first glance.

MySQL offers an interesting alternative to the ANSI SQL standard for querying tables—the *HANDLER* statement. The *HANDLER* statement works a lot like *SELECT*, except that *HANDLER* provides very rapid data reads that circumvent the SQL query engine in MySQL. However, since the *HANDLER* statement is not a SQL statement, we'll refer you to the MySQL documentation for more information.

Oracle

Oracle allows a very large number of extensions to the ANSI *SELECT* statement. For example, since both nested tables and partitioned tables are allowed in Oracle (see *CREATE TABLE*), the *SELECT* statement allows queries to those types of structures:

```
[WITH query_name AS (subquery)[, ...])
  SELECT { {[ALL | DISTINCT]} | [UNIQUE] } [optimizer_hints]
    select_item [AS alias][, ...]
    [INTO {variable[, ...] | record}]
  FROM {[ONLY] {[schema.][table_name | view_name |
    materialized_view_name]}}[@database_link] [AS [OF]
      {SCN | TIMESTAMP} expression] |
    subquery [WITH {READ ONLY | CHECK OPTION [CONSTRAINT constraint_name]})] |
    {[VERSIONS BETWEEN {SCN | TIMESTAMP} {exp | MINVALUE} AND
      {exp | MAXVALUE}] AS OF {SCN | TIMESTAMP} expression} |
    TABLE (nested_table_column) [(+)]
      {[PARTITION (partition_name) | SUBPARTITION (subpartition_name)) |
      [SAMPLE [BLOCK] [sample_percentage] [SEED (seed_value)]]} [AS alias]
      [, ...]
    [ [join_type] JOIN join_condition [PARTITION BY expression[, ...]] ]
  [WHERE search_condition
    [ {AND | OR} search_condition[, ...] ]
    {[START WITH value] CONNECT BY [PRIOR] condition}]
  [GROUP BY group_by_expression [HAVING search_condition]]
  [MODEL model_clause]
  [ORDER [SIBLINGS] BY order_expression {[ASC | DESC]}]
    {[NULLS FIRST | NULLS LAST]}]
  [FOR UPDATE [OF [schema.][table.]column][, ...]
    {[NOWAIT | WAIT (integer)]}]
```

Unless otherwise noted, the clauses shown here follow the ANSI standard. Similarly, elements of the clauses are identical to those in the ANSI standard unless otherwise noted. For example, Oracle's *GROUP BY* clause is nearly identical to the ANSI standard, including its component elements, such as *ROLLUP*, *CUBE*, *GROUPING SETS*, concatenated *GROUPING SETS*, and the *HAVING* clause.

The parameters are:

*WITH *query_name* AS (*subquery*)[, ...]*

References a named query. Oracle allows you to assign names to subqueries, and you can then reference the subquery in multiple places by specifying only its name. (Oracle optimizes this process by treating the subquery as a nested table subquery or a temporary table.)

ALL | DISTINCT | UNIQUE

ALL and *DISTINCT* are identical to the ANSI *ALL* and *DISTINCT* clauses (see the ANSI SQL syntax described earlier). *UNIQUE* is a synonym for *DISTINCT*. *DISTINCT* cannot be used on *LOB* columns.

optimizer_hint

Overrides the default behavior of the query optimizer with user-specified behaviors. For example, hints can force Oracle to use an index that it might not otherwise use or to avoid an index that it might otherwise use. Refer to the vendor documentation for more information about optimizer hints.

select_item

May be an expression or a column from a named query, table, view, or materialized view in the format *[schema.]***[table_name.]***column_name*. If you omit the schema, Oracle assumes the context of the current schema. Oracle also allows for *named queries* (explained earlier, under the *WITH* keyword) that may be referenced much like nested table subqueries (refer to the section on *SUBQUERY* later in this chapter). Oracle refers to using named queries as *subquery factoring*. In addition to named queries, Oracle also supports subqueries and the asterisk (*), shorthand for all columns, in the *select_item* list.

INTO {variable[, ...] | record}

Retrieves the result set values into PL/SQL variables or into a PL/SQL record.

FROM [ONLY]

Identifies the table, view, materialized view, partition, or subquery from which the result set is retrieved. The *ONLY* keyword is optional and applies only to views belonging to a hierarchy. Use *ONLY* when you want to retrieve records from a named view only, and not from any of its subviews.

AS [OF] {SCN | TIMESTAMP} expression

Implements SQL-driven flashback, whereby system change numbers (or timestamps) are applied to each object in the *select_item* list. Records retrieved by the query are only those that existed at the specific system change number (SCN) or time. (This feature can also be implemented at the session level using the *DBMS_FLASHBACK* built-in package.) *SCN expression* must equal a number, while *TIMESTAMP expression* must equal a timestamp value. Flashback queries cannot be used on linked servers.

subquery [WITH {READ ONLY | CHECK OPTION [CONSTRAINT constraint_name]}]

Mentioned separately because Oracle allows you extra ways to control a subquery. *WITH READ ONLY* indicates that the target of the subquery cannot be updated. *WITH CHECK OPTION* indicates that any update to the target of the subquery must produce rows that would be included in the subquery. *WITH CONSTRAINT* creates a *CHECK OPTION* constraint of *constraint_name* on the table. Note that *WITH CHECK OPTION* and *WITH CONSTRAINT* are usually used in *INSERT...SELECT* statements.

[[VERSIONS BETWEEN {SCN | TIMESTAMP} {exp | MINVALUE} AND {exp | MAXVALUE}]] AS OF {SCN | TIMESTAMP} expression

Specifies a special kind of query using a *flashback_query_clause* to retrieve the history of changes made to data from a table, view, or materialized view. The *VERSIONS_XID* pseudocolumn shows the identifier corresponding to the transaction that made the change. A *flashback_query_clause* requires that you specify an *SCN* (system change number) or a *TIMESTAMP* value for each object in the *select_item* list. (Implement SQL-driven session-level flashback using the Oracle *DBMS_FLASHBACK* package.)

The optional subclause *VERSIONS BETWEEN* is used to retrieve multiple versions of the data specified, either using an upper and lower boundary of an SCN (a number) or *TIMESTAMP* (a timestamp value), or using the *MINVALUE* and *MAXVALUE* keywords. Without this clause, only one past version of the data is returned. (Oracle also provides several version query pseudocolumns for additional versioning information.)

The *AS OF* clause, discussed earlier in this list, determines the SCN or moment in time from which the database issues the query when used with the *VERSIONS* clause.

You cannot use flashback queries with the *VERSIONS* clause against temporary tables, external tables, tables in a cluster, or views.

TABLE

Required when querying a hierarchically declared nested table.

PARTITION

Restricts a query to the specified partition of the table. In essence, rows are retrieved only from the named partition, not from the entire table.

SUBPARTITION

Restricts a query to the specified subpartition of the table. I/O is reduced because the rows are retrieved from only the named subpartition instead of the entire table.

SAMPLE [BLOCK] [*sampling_percentage*] [SEED (*seed_value*)]

Tells Oracle to select records from a random sampling of rows within the result set, as either a percentage of rows or blocks, rather than from the entire table. *BLOCK* tells Oracle to use block sampling rather than row sampling. The *sampling_percentage*, telling Oracle the total block or row percentage to be included in the sample, may be anywhere between .000001 and 99. The optional *SEED* clause is used to provide limited repeatability. If you specify a seed value, Oracle will attempt to return the same sample from one execution of the query to the next. The seed value can be between 0 and 4294967295. When *SEED* is omitted, the resulting sample will change from one execution of the query to the next. Sampling may be used only on single-table queries.

JOIN

Merges the result sets of two or more tables in a single query. Described in detail in the upcoming “Oracle” subsection.

PARTITION BY *expression*[, ...]

Defines a special kind of query called a *partitioned_outer_join* that extends the conventional outer join syntax by applying a right or left outer join to a partition of one or more rows specified by the *expression*. This is especially useful for querying sparse data along a particular dimension of data, thereby returning rows that otherwise would be omitted from the result set. For an example, see the upcoming subsection “Partitioned outer joins.” The *PARTITION BY* clause can be used on either side of an outer join, resulting in a *UNION* of the outer joins of each of the partitions in the partitioned result set and the table on the other side of the join. (When this clause is omitted, Oracle treats the entire result set as a single partition.) *PARTITION BY* is not allowed with *FULL OUTER JOIN*.

WHERE... [[START WITH *value*] CONNECT BY [PRIOR] *condition*]

Filters records returned in the result set. Oracle allows the use of hierarchical information within tables, whose filtering can be controlled with the *START WITH* clause. *START WITH* identifies the rows that will serve as the parent rows

in the result set. *CONNECT BY* identifies the relationship *condition* between the parent rows and their child rows. The *PRIOR* keyword is used to identify the parent rows instead of the child rows.

Hierarchical queries use the *LEVEL* pseudocolumn to identify the root node (1), the child nodes (2), the grandchild nodes (3), and so forth. Other pseudocolumns available in hierarchical queries are *CONNECT_BY_ISCYCLE* and *CONNECT_BY_ISLEAF*. Hierarchical queries are mutually exclusive of the *ORDER BY* and *GROUP BY* clauses. Do not use those clauses in a query containing *START WITH* or *CONNECT BY*. You can order records from siblings of the same parent table by using the *ORDER SIBLINGS BY* clause.

ORDER [SIBLINGS] BY *order_expression* {NULLS FIRST | NULLS LAST}

Sorts the result set of the query in order of *order_expression*. The *order_expression* may be the column names, their aliases, or integers indicating their ordinal positions. The *ORDER SIBLINGS BY* clause is used when querying hierarchical tables with the *CONNECT BY* clause. *ORDER SIBLINGS BY* tells Oracle to preserve any ordering specified in the *CONNECT BY* clause and applies the ordering only to siblings (i.e., rows of equal level in the hierarchy). *NULLS FIRST* and *NULLS LAST* specify that the records containing NULLs should appear either first or last, respectively.

FOR UPDATE [OF [schema.]... {[NOWAIT | WAIT (*integer*)]}

Locks the rows of the result set so that other users cannot lock or update them until you're finished with your transaction. *FOR UPDATE* cannot be used in a subquery, in queries using *DISTINCT* or *GROUP BY*, or in queries with set operators or aggregate functions. Child rows in a hierarchical table are not locked when this clause is issued against the parent rows. The *OF* keyword is used to lock only the selected table or view. Otherwise, Oracle locks all the tables or views referenced in the *FROM* clause. When using *OF*, the columns are not significant, though real column names (not aliases) must be used. The *NOWAIT* and *WAIT* keywords tell Oracle either to return control immediately if a lock already exists or to wait *integer* seconds before returning control to you, respectively. If neither *NOWAIT* nor *WAIT* is specified, Oracle waits until the rows become available.

Unlike some other database platforms, Oracle does not allow a *SELECT* statement without a *FROM* clause. The following query, for example, is invalid:

```
SELECT 2 + 2;
```

As a workaround, Oracle has provided a special-purpose table called **dual**. Any time you want to write a query that does not retrieve data from a user-created table, such as to perform a calculation, use *FROM dual*. All of the following queries are valid:

```
SELECT 2 + 2
FROM dual;
SELECT (((52-4) * 5) * 8)
FROM dual;
```

Oracle's implementation of *SELECT* is quite straightforward if you want to retrieve data from a table. For example, Oracle allows the use of *named queries*. A named query is, in a sense, an alias to an entire query that can save you time when you're writing a complex multi-subquery *SELECT* statement. For example:

```
WITH pub_costs AS
  (SELECT pub_id, SUM(job_lvl) dept_total
   FROM employees e
   GROUP BY pub_id),
```

```

avg_costs AS
  (SELECT SUM(dept_total)/COUNT(*) avg
   FROM employee)
SELECT * FROM pub_costs
  WHERE dept_total > (SELECT avg FROM avg_cost)
  ORDER BY department_name;

```

In this example, we create two named subqueries—**pub_costs** and **avg_costs**—which are later referenced in the main query. The named queries are effectively the same as subqueries; however, subqueries must be written out in their entirety each time they’re used, while named queries need not.

Oracle allows you to select rows from a single partition of a partitioned table using the **PARTITION** clause, or to retrieve only a statistical sampling of the rows (as a percentage of rows or blocks) of a result set using **SAMPLE**. For example:

```

SELECT *
  FROM sales PARTITION (sales_2004_q3) sales
 WHERE sales.qty > 1000;
SELECT *
  FROM sales SAMPLE (12);

```

Flashback queries are a feature of Oracle that enable retrieval of point-in-time result sets. For example, you could find out what everyone’s salary was yesterday before a big change was applied to the database:

```

SELECT job_lvl, lname, fname
  FROM employee
 AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY);

```

Another interesting Oracle extension of the standard query format is the *hierarchic query*. Hierarchic queries return the results of queries against hierarchically designed tables in the order you define. For example, the following query returns the names of the employees and their positions in the hierarchy (represented by the position in the **org_char** column), employee IDs, manager IDs, and job IDs:

```

SELECT LPAD(' ',2*(LEVEL-1)) || lname AS org_chart,
      emp_id, mgr_id, job_id
  FROM employee
 START WITH job_id = 'Chief Executive Officer'
CONNECT BY PRIOR emp_id = mgr_id;
  ORG_CHART      EMPLOYEE_ID MANAGER_ID JOB_ID
-----
Cramer          101        100 Chief Executive Officer
  Devon          108        101 Business Operations Mgr
    Thomas        109        108 Acquisitions Manager
      Koskitalo    110        108 Productions Manager
        Tonini      111        108 Operations Manager
      Whalen        200        101 Admin Assistant
        Chang        203        101 Chief Financial Officer
        Gietz        206        203 Comptroller
      Buchanan      102        101 VP Sales
        Callahan     103        102 Marketing Manager

```

In the previous query, the **CONNECT BY** clause defines the hierarchical relationship of the **emp_id** value as the parent row equal to the **mgr_id** value in the child row, while the **START WITH** clause specifies where in the hierarchy the result set should begin.

Oracle supports the following types of *JOIN* syntax (refer to the section on the *JOIN* subclause for more details):

```
FROM table1 {
  CROSS JOIN table2 |
  INNER JOIN table2 [ {ON join_condition |
    USING (column_list)} ] |
  LEFT [OUTER] JOIN table2 [ {ON join_condition |
    | USING (column_list)} ] |
  NATURAL [LEFT [OUTER]] JOIN table2 |
  RIGHT [OUTER] JOIN table2 [ {ON join_condition |
    | USING (column_list)} ] |
  NATURAL [RIGHT [OUTER]] JOIN table2
  FULL [OUTER] JOIN table2 }
```

[CROSS] JOIN

Retrieves all records of both *table1* and *table2*. This is syntactically the same as *FROM table1, table2* with no join conditions in the *WHERE* clause.

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* with join conditions in the *WHERE* clause is semantically equivalent to an inner join.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause containing all columns that are in common between the two joined tables. (Be careful if columns have the same names but not the same datatypes or the same sort of values!) *LOB* columns cannot be referenced in a natural join. Referencing a *LOB* or collection column in a *NATURAL JOIN* will return an error.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, NULL values are substituted for that table's columns. You can use this type of join to retrieve all the records in a table, even when there are no counterparts in the joined table. For example:

```
SELECT j.job_id, e.lname
  FROM jobs j
  LEFT OUTER JOIN employee e ON j.job_id = e.job_id
  ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table, regardless of whether there is a matching record in the leftmost table. A right join is the same as a left join, except that the optional table is on the left.

FULL [OUTER] JOIN

Specifies that all rows from both tables be returned, regardless of whether a row from one table matches a row in the other table. Any columns that have no value in the corresponding joined table are assigned a NULL value.

ON *join_condition*

Declares the condition(s) that join the result set of two tables together. This takes the form of declaring the columns in *table1* and *table2* that must match the join condition. When multiple columns must be compared, use the *AND* clause.

USING (column_list)

Acts as an alternative to the *ON* clause. Instead of describing the conditions of the join, simply provide a column name (or columns separated by commas) that appears in both tables. The column names must be identical in both tables and cannot be prefixed with a table name or alias. *USING* cannot be used on *LOB* columns of any type. The following two queries produce identical results. One is written with a *USING* clause and the other specifies join conditions using ANSI syntax:

```
SELECT column1
  FROM foo
  LEFT JOIN poo USING (column1, column2);
```

```
SELECT column1
  FROM foo
  LEFT JOIN poo ON foo.column1 = poo.column1
    AND foo.column2 = poo.column2;
```

Note that older Oracle syntax for joins centered around the *WHERE* clause, and outer joins were described using a (+) marker. ANSI *JOIN* syntax only became available with Oracle 9i Release 1. Therefore, you will often see existing code using the old syntax. For example, the following query is semantically equivalent to the earlier example of a left outer join:

```
SELECT j.job_id, e.lname
  FROM jobs j, employee e
 WHERE j.job_id = e.job_id (+)
 ORDER BY d.job_id
```

This older syntax is sometimes problematic and more difficult to read. You are strongly advised to use the ANSI-standard syntax.

Partitioned outer joins

Partitioned outer joins are useful for retrieving sparse data that might otherwise not be easily seen in a result set. (The ANSI standard describes partitioned outer joins, but Oracle is the first to support them.) For example, our **product** table keeps track of all products we produce, while the **manufacturing** table shows when we produce them. Since we're not continuously making every product at all times, the joined data between the two tables may be sparse at times:

```
SELECT manufacturing.time_id AS time, product_name AS name,
       quantity AS qty
  FROM product
 PARTITION BY (product_name)
  RIGHT OUTER JOIN times ON (manufacturing.time_id =
    product.time_id)
 WHERE manufacturing.time_id
   BETWEEN TO_DATE('01/10/05', 'DD/MM/YY')
     AND TO_DATE('06/10/05', 'DD/MM/YY')
 ORDER BY 2, 1;
```

returns the following:

time	name	qty
01-OCT-05	flux capacitor	10
02-OCT-05	flux capacitor	

```

03-OCT-05 flux capacitor
04-OCT-05 flux capacitor
05-OCT-05 flux capacitor
06-OCT-05 flux capacitor 10
06-OCT-05 flux capacitor 8
01-OCT-05 transmogrifier 10
01-OCT-05 transmogrifier 15
02-OCT-05 transmogrifier
03-OCT-05 transmogrifier
04-OCT-05 transmogrifier 10
04-OCT-05 transmogrifier 11
05-OCT-05 transmogrifier
06-OCT-05 transmogrifier

```

The example query and result set show that partitioned outer joins are useful for retrieving result sets that might otherwise be hard to query due to sparse data.

Flashback queries

Oracle 9i and later versions also supports flashback queries—queries that keep track of previous values of the result requested by the *SELECT* statement. In the following set of example code, we'll issue a regular query on a table, change the values in the table with an *UPDATE* statement, and then query the flashback version of the data. First, the regular query:

```

SELECT salary FROM employees
WHERE last_name = 'McCreary';

```

The results are:

```

SALARY
-----
3800

```

Now, we'll change the value in the **employees** table and query the table to confirm the current value:

```

UPDATE employees SET salary = 4000
WHERE last_name = 'McCreary ';
SELECT salary FROM employees
WHERE last_name = 'McCreary ';

```

The results are:

```

SALARY
-----
4000

```

Finally, we'll perform a flashback query to see what the **salary** value was in the past:

```

SELECT salary FROM employees
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' DAY)
WHERE last_name = 'McCreary ';

```

The results are:

```

SALARY
-----
3800

```

If we wanted to be more elaborate, we could find out all of the values of **salary** for a given time period, say, the last two days:

```
SELECT salary FROM employees
VERSIONS BETWEEN TIMESTAMP
    SYSTIMESTAMP - INTERVAL '1' MINUTE AND
    SYSTIMESTAMP - INTERVAL '2' DAY
WHERE last_name = 'McCreary';
```

The results are:

```
SALARY
```

```
4000
3800
```

The MODEL clause

Oracle Database 10g introduced a powerful new clause, called *MODEL*, which enables spreadsheet-like result sets from a *SELECT* statement. The *MODEL* clause, in particular, is designed to alleviate the need for developers to extract data from the database and put it into a spreadsheet, like Microsoft Excel, for further manipulation. It creates a multidimensional array in which cells can be referenced by dimension values. For instance, you might dimension an array on product and time, specifying column values that you wish to access via combinations of those two dimensions. You can then write rules that are similar in concept to spreadsheet formulas, that are executed in order to change values in your model, or that create new values, and perhaps even new rows, in your model.

Syntactically, the *MODEL* clause appears after the *GROUP BY* and *HAVING* clauses and before the *ORDER BY* clause. The earlier syntax diagram for Oracle's *SELECT* statement shows the position of the clause, and the syntax details are presented here:

```
MODEL
  [{IGNORE | KEEP} NAV] [UNIQUE {DIMENSION | SINGLE REFERENCE}]
  [ RETURN {UPDATED | ALL} ]
  [ REFERENCE reference_model_name ON (subquery)
    [PARTITION BY (column [AS alias][, ...])]
    DIMENSION BY (column [AS alias][, ...])
    MEASURES (column [AS alias][, ...])
    [{IGNORE | KEEP} NAV] [UNIQUE {DIMENSION | SINGLE REFERENCE}] ]
  [MAIN main_model_name]
  [PARTITION BY (column [AS alias][, ...])]
  DIMENSION BY (column [AS alias][, ...])
  MEASURES (column [AS alias][, ...])
  [{IGNORE | KEEP} NAV] [UNIQUE {DIMENSION | SINGLE REFERENCE}]]
  model_rules_clause
  [RULES [UPsert [ALL] | UPDATE] [{AUTOMATIC | SEQUENTIAL} ORDER]]
  [ITERATE (int) [UNTIL (ending_condition)]]
  ( [ {UPsert [ALL] | UPDATE} ]
  measure [...]
  [FOR { dimension | ( dimension[, ...] ) }
  { [IN ({subquery | literal[, ...]})]
    [LIKE pattern] FROM start_literal TO end_literal
    {INCREMENT | DECREMENT} diff_literal }[, ...]
  [ORDER [SIBLINGS] BY (order_column [ASC | DESC]
    [NULLS FIRST | NULLS LAST][, ...])]]
  = expr[, ...] )
```

The parameters of this *MODEL* clause are as follows:

{IGNORE | KEEP} NAV

Specifies whether NULL or absent values (NAV) are retained as NULLS (KEEP), or whether they are replaced with suitable defaults (IGNORE): zero for numeric types, 1-Jan-2000 for date types, an empty string for character types, and NULL for anything else.

UNIQUE {DIMENSION | SINGLE REFERENCE}

Specifies the scope within which the database ensures that a given cell reference points to a unique data value. Use *DIMENSION* to require that each *possible* cell reference, whether on the left or right side of a rule, represent a single value. Use *SINGLE REFERENCE* to perform that check only for those cell references that appear on the righthand side of a rule.

RETURN {UPDATED | ALL} ROWS

Specifies whether all rows are returned from model processing, or whether only updated row are returned.

reference_model_name

Defines a reference model on which you cannot perform calculations, but containing values that you can reference from within your main query.

reference_model_name ON (subquery)

Specifies the name and rowsource for a reference model.

alias

Specifies an alias for a partition.

DIMENSION BY (column[, column...])

Specifies the dimensions for a model. Values from these columns represent the set of index values that are used to identify cells in the multidimensional addressing space.

MEASURES (column[, column...])

Specifies the values associated with each unique combination of dimensions (e.g., with each cell of the model).

PARTITION BY [(]column[, column...][)]

Splits a model into independent partitions based on the columns given. You can not partition reference models.

main_model_name

Represents the model on which you perform work. Rows from your containing *SELECT* feed into this model, rules are applied, and the resulting rows are returned.

MAIN model_name

Begins the definition of the main model, and also gives that model a name.

RULES [UPsert | UPDATE]

Specifies whether rules may both create new cells and update existing cells (UPsert), or whether they much only update existing cells (UPDATE). If you want your model to be able to create new rows in your result set, specify UPsert. The default is UPsert. You can also control this behavior on a rule-by-rule basis; see *rule* in the syntax.

{AUTOMATIC | SEQUENTIAL} ORDER

Specifies whether the optimizer determines the order in which rules are evaluated (AUTOMATIC), or whether rules are evaluated in the order in which you list them (SEQUENTIAL). The default is SEQUENTIAL.

ITERATE (int)

Requests that entire set of rules be evaluated repeatedly, *int* times. The default is to evaluate the set of rules just once.

UNTIL (ending_condition)

Specifies a condition that, when met, causes iteration to end. You must still specify a *int*, which serves as a safeguard against infinite loops.

measure[...]

A reference to one of the measures listed in the *MEASURES* clause. When you reference a measure, the square brackets are part of the syntax. You must specify all dimensions, either via a subquery or by listing them, and the specific value of the measure associated with those dimensions will be returned, or referenced.

FOR ...

A *FOR*-loop iterating over one or many dimensions. The multi-iterating *FOR*-loop is much like a subquery where each row of the result set represents a specific combination of dimensions.

dimension_indexes

A list of values, whether from columns or expressions, that collectively identify a unique cell in the model.

IN ({subquery | literal[, literal...]})

The source of values for a for-loop may be a subquery, or it may be a specific list of literal values.

LIKE pattern

Allows you to insert dimension values into a pattern. Use a percent-sign to mark the location at which you want dimension values to be inserted. For example, use *FOR x LIKE 'A%B' FROM 1 TO 3 INCREMENT 1* to generate values such as 'A1B', 'A2B', 'A3B'.

FROM start_literal TO end_literal {INCREMENT | DECREMENT} diff_literal

Defines the starting and ending for-loop values, and also the difference between each subsequent value as the loop iterates from start to end.

ORDER BY (order_column[, ...])

Imposes an order of evaluation with respect to the cells referenced from the left side of a rule. Use this clause if you want a rule to be applied to cells in order. Otherwise, you have no guarantee as to the order in which the rule is applied to the cells that it affects.

Following is a list of functions that have been designed specifically for use in the *MODEL* clause:

CV() or CV(dimENSION_column)

Returns the current value of a dimension column. May be used only on the right-hand side of an expression in a rule. When the *CV()* form is used, the dimension column is determined implicitly based on the function call's position in a list of dimension values.

PRESENTNNV(measure[dimension, dimension...], not_null, was_null)

Returns either *not_null* or *was_null*, depending on whether the specified measure was NULL when model processing began. This function may be used only from the righthand side of a rule expression.

PRESENTV(measure[dimension, dimension...], did_exist, didnt_exist)

Returns either *did_exist* or *didnt_exist*, depending on whether the specified measure existed when model processing began. This function may be used only from the righthand side of a rule expression. Be aware that whether a measure existed is a completely separate question from whether that measure was NULL.

ITERATION_NUMBER

Returns zero on the first iteration through the rules, 1 on the second iteration, and so forth. This is useful when you want to base rule calculations on the number of iterations.

The following example demonstrates that the *MODEL* clause gives a normal *SELECT* statement the ability to construct a multidimensional array as a result set and calculate inter-row and inter-array values interdependently. The newly calculated values are returned as part of the *SELECT* statement's result set:

```
SELECT SUBSTR(region,1,20) country, SUBSTR(product,1,15) product,
       year, sales
  FROM sales_view
 WHERE region IN ('USA','UK')
  MODEL RETURN UPDATED ROWS
    PARTITION BY (region)
    DIMENSION BY (product, year)
    MEASURES (sale sales)
    RULES (
      sales['Bounce',2006] = sales['Bounce',2005] + sales['Bounce',2004],
      sales['Y Box', 2006] = sales['Y Box', 2005],
      sales['2_Products',2006] = sales['Bounce',2006]
        + sales['Y Box',2006] )
    ORDER BY region, product, year;
```

In this example, a query against the **SALES_VIEW** materialized view returns the sum of sales over the course of a few years for the regions 'USA' and 'UK'. The *MODEL* clause then falls between the *WHERE* clause and the *ORDER BY* clause. Since **SALES_VIEW** currently holds data for the years 2004 and 2005, we provide it rules to calculate figures for the year 2006.

The subclause *RETURN UPDATED ROWS* limits the result set to the rows that were created or updated by the query. Next, the example defines the logical divisions of the data using data elements from the materialized view and using the *PARTITION BY*, *DIMENSION BY*, and *MEASURES* subclauses. The *RULES* subclause then references individual measures of the model by referring to combinations of different dimension values much like a spreadsheet macro references worksheet cells with specific lookups and references to ranges of values.

Oracle (and SQL Server using a somewhat different technique) both support a non-ANSI, non-ISO query known as a *pivot* query. Although you should refer to the vendor documentation for exactly how to write a pivot (or unpivot) query, an example here will help you take advantage of this useful technique. A pivot query turns the result set on its side, enabling you to extract more value from the data. In Oracle, you must first create your pivot table. By using a pivot table, you can now turn the result "on its side" so that the *order_type* column becomes the column headings:

```
CREATE TABLE pivot_table AS
SELECT * FROM (SELECT year, order_type, amt FROM sales)
PIVOT SUM(amt) FOR order_type IN ('retail', 'web');

SELECT * FROM pivot_table ORDER BY YEAR;
```

where the results are:

YEAR	RETAIL	WEB
2004	7014.54	
2005	9745.12	
2006	16717.88	10056.6
2007	28833.34	39334.9
2008	66165.77	127109.4

PostgreSQL

PostgreSQL supports a straightforward implementation of the *SELECT* statement. It supports *JOIN* and subquery applications. PostgreSQL also allows the creation of new temporary or permanent tables using the *SELECT...INTO* syntax.

```
SELECT [ALL | DISTINCT [ON (select_item[, ...])]]  
[AS alias [(alias_list)]][, ...]  
[INTO [[TEMP]ORARY] [TABLE] new_table_name]  
[FROM [ONLY] table1[*] [AS alias][, ...] ]  
[ [join type] JOIN table2 {[ON join_condition] |  
  [USING (column_list)]}  
  [WHERE search_condition]  
  [GROUP BY group_by_expression]  
  [HAVING having_condition]  
  [ORDER BY order_by_expression [{ASC | DESC | USING operator}[, ...] ]]  
  [FOR UPDATE [OF column[, ...]]]  
  [LIMIT {count | ALL} ][OFFSET number_of_records]] ]
```

where:

ALL | DISTINCT [ON (select_item[, ...])]

Supports the *ALL* and *DISTINCT* keywords of the ANSI SQL standard, where *ALL* (the default) returns all rows (including duplicates) and *DISTINCT* eliminates duplicate rows. In addition, *DISTINCT ON* eliminates duplicates on only the specified *select_items*, not on all of the *select_items* in the query (example below).

select_item

Includes the standard elements of a *select_item* list supported by the ANSI SQL standard. In addition to the asterisk (*) shorthand to retrieve all rows, you can use *table_name.** to retrieve all rows from an individual table.

AS alias [(alias_list)]

Creates an alias or a list of aliases for one or more columns (or tables in the *FROM* clause). *AS* is required for *select_item* aliases, but not for *FROM* table aliases. (Some other database platforms treat the *AS* as an option when declaring an alias.)

INTO [[TEMP]ORARY] [TABLE] new_table_name

Creates a new table from the result set of the query. Both *TEMP* and *TEMPORARY* are acceptable usages to create a temporary table that is automatically dropped at the end of the session. Otherwise, the command creates a permanent table. Permanent tables created with this statement must have new, unique names but temporary tables may have the same name as an existing table. If you create a temporary table with the same name as an existing permanent table, the temporary table is used to resolve all operations against that table name while in the same session as the one that created it. Other sessions will continue to see the existing permanent table.

FROM [ONLY] *table1*, ...]

Specifies one or more source tables where the data resides. (Be sure to specify a join condition or a theta *WHERE* clause so that you don't get a Cartesian product of all records in all tables.) PostgreSQL allows inheritance in child tables of declared parent tables. Use the *ONLY* keyword to suppress rows from the child tables of your source table. (You can turn off this default inheritance globally with the command *SET SQL_Inheritance TO OFF*.) PostgreSQL also supports nested table subqueries (see the section on *SUBQUERY* later in this chapter). The *FROM* clause is not needed when used for computation:

```
SELECT 8 * 40;
```

PostgreSQL will also include an implicit *FROM* on *SELECT* statements that include schema-identified columns. For example, the following query is acceptable (though not recommended):

```
SELECT sales.stor_id WHERE sales.stor_id = '6380';
```

GROUP BY group_by_expression

Allows *group_by_expressions* that can be the column name or the ordinal number of the column (as determined by its position in the *select_item* list). An example illustrates this concept under the *ORDER BY* entry just below.

ORDER BY order_by_expression

Allows *order_by_expressions* that can be the column name, its alias, or the ordinal number of the column (as determined by its position in the *select_item* list.) For example, the following two queries are functionally identical:

```
SELECT stor_id, ord_date, qty AS quantity
FROM sales
ORDER BY stor_id, ord_date DESC, qty ASC;
```

```
SELECT stor_id, ord_date, qty
FROM sales
ORDER BY 1, 2 DESC, quantity;
```

In single-table *SELECT* statements, you may also order by columns of the table that do not appear in the *select_item* list. For example:

```
SELECT *
FROM sales
ORDER BY stor_id, qty;
```

ASC and *DESC* are ANSI standard. If not specified, *ASC* is the default. PostgreSQL sorts *NULL* values as higher than any other value, causing *NULL* values to appear at the end of *ASC* sorts and at the beginning of *DESC* sorts.

FOR UPDATE [OF column[, ...]] LIMIT {count | ALL} [OFFSET [number_of_records]]

Limits the number of rows returned by the query to the maximum specified by the integer *count*. The optional *OFFSET* keyword tells PostgreSQL to skip *number_of_records* before starting to return rows. You should make sure you have an *ORDER BY* clause when using *LIMIT* or you may get an unexpected result set. (PostgreSQL 7.0 and greater treat *LIMIT/OFFSET* as a hint and use it to create optimal, possibly very different query plans.)

PostgreSQL supports a handy variation of the *DISTINCT* clause—*DISTINCT ON (select_item[, ...])*. This variation allows you to pick and choose the exact columns that are considered for elimination of duplications. PostgreSQL chooses the result set in a manner much like it does for *ORDER BY*. You should include an *ORDER BY* clause so that there's no unpredictability as to which record is returned. For example:

```
SELECT DISTINCT ON (stor_id), ord_date, qty
  FROM sales
 ORDER BY stor_id, ord_date DESC;
```

The above query retrieves the most recent sales report for each store based on the most recent order date. However, there would be no way to predict what single record would have been returned without the *ORDER BY* clause.

PostgreSQL supports only these types of *JOIN* syntax:

```
FROM table1[, ...] {
  CROSS JOIN table2 |
  [INNER] JOIN table2 [ {ON join_condition |
    USING (column_list)} ] |
  LEFT [OUTER] JOIN table2 [ {ON join_condition |
    USING (column_list)} ] |
  NATURAL [LEFT [OUTER]] JOIN table2 |
  RIGHT [OUTER] JOIN table2 [ {ON join_condition |
    USING (column_list)} ] |
  NATURAL [RIGHT [OUTER]] JOIN table2
  FULL [OUTER] JOIN table2 }
```

where:

[CROSS] JOIN

Retrieves all records of both *table1* and *table2*. This is syntactically the same as *FROM table1, table2*. The results are a Cartesian product (usually a very bad thing!).

INNER JOIN

Retrieves those records of both *table1* and *table2* where there are matching values in both tables according to the join condition. Note that the syntax *FROM table1, table2* is semantically equivalent to an inner join without a join condition (i.e., a Cartesian product). When using *FROM table1, table2*, be sure include the join condition(s) in the *WHERE* clause. If you leave out a join type, PostgreSQL assumes you want an inner join.

NATURAL

Shortcuts the need to declare a join condition by assuming a *USING* clause that names all columns that have the same name in both tables. (Be careful if columns have the same names but not the same datatypes or the same sort of values!) *NATURAL* is mutually exclusive of a specified join condition via *ON* or *USING*.

LEFT [OUTER] JOIN

Retrieves all records in the leftmost table (i.e., *table1*) and matching records in the rightmost table (i.e., *table2*). If there isn't a matching record in *table2*, a *NULL* value is substituted. You can use *LEFT OUTER JOIN* to find all the records in a table even when there are no counterparts in the joined table. For example:

```
SELECT j.job_id, e.lname
  FROM jobs j
 LEFT OUTER JOIN employee e ON j.job_id = e.job_id
 ORDER BY d.job_id
```

RIGHT [OUTER] JOIN

Retrieves all records in the rightmost table regardless of whether there are matching records in the leftmost table. Analogous to a *LEFT JOIN*, but the join is performed in the opposite direction.

FULL [OUTER] JOIN

Retrieves all records in the inner join in addition to all records that have no matching records on both the left and right sides.

ON *join_condition*

Declares the condition(s) that join the result set of two (or more) tables together. This takes the form of declaring the columns in *table1* and *table2* that must be equal. When multiple columns must be compared, use the *AND* clause.

SQL Server

SQL Server supports most of the basic elements of the ANSI *SELECT* statement, including all of the various join types. It also offers several variations on the *SELECT* statement, including optimizer hints, the *INTO* clause, the *TOP* clause, *GROUP BY* variations, *COMPUTE*, and *WITH OPTIONS*:

```
[WITH cte_expression[, ...]]
SELECT {[ALL | DISTINCT] | [TOP number [PERCENT] [WITH TIES]]}
    select_item [AS alias]
    [INTO new_table_name]
    [FROM {[rowset_function | table1[, ...]} [AS alias]]
    [ [join type] JOIN table2 [ON join_condition] ]
    [WHERE search_condition]
    [GROUP BY [GROUPING SETS] {grouping_column[, ...] | ALL}]
    [WITH { CUBE | ROLLUP }]
    [HAVING search_condition]
    [ORDER BY order_by_expression [ ASC | DESC ]]
    [COMPUTE {aggregation (expression)}[, ...]
        [BY expression[, ...]]]
    [FOR {BROWSE | XML {RAW | AUTO | EXPLICIT}
        [, XMLDATA][, ELEMENTS][, BINARY base64]}]
    [OPTION (hint[, ...])]]
```

where:

WITH *cte_expression*

Defines the temporary named result set of a common table expression, derived from a *SELECT* statement, for the *SELECT* statement.

TOP *number* [PERCENT] [WITH TIES]

Indicates that only the specified *number* of rows should be retrieved in the query result set. If *PERCENT* is specified, only the first *number* percent of the rows are retrieved. *WITH TIES* is used only for queries with an *ORDER BY* clause. This variation specifies that additional rows are returned from the base result set using the same value in the *ORDER BY* clause, appearing as the last of the *TOP* rows.

INTO *new_table_name*

Creates a new table from the result set of the query. You can use this command to create temporary or permanent tables. (Refer to SQL Server's rules for creating temporary or permanent tables in the section.) The *SELECT...INTO* command quickly copies the rows and columns queried from other table(s) into a new table using a non-logged operation. Since it is not logged, *COMMIT* and *ROLLBACK* statements do not affect it.

FROM {[rowset_function | table1[, ...]]}

Supports the standard behavior of the ANSI SQL FROM clause, including nested table subqueries. In addition, SQL Server supports a set of extensions called *rowset functions*. Rowset functions allow SQL Server to source data from special or external data sources such as XML streams, full-text search file structures (a special structure in SQL Server used to store things like MS Word documents and MS PowerPoint slide shows within the database), or external data sources (like an MS Excel spreadsheet).

(See the SQL Server documentation for the full description of the available FROM {[rowset_function | table1[, ...]]} options. Among the many possibilities, SQL Server currently supports the following *rowset functions*:

CONTAINSTABLE

Returns a table derived from a specified table that contains at least one full-text index *TEXT* or *NTEXT* column. The records derived are based upon either a precise, fuzzy, weighted-match, or proximity-match search. The derived table is then treated like any other FROM data source.

FREETEXTTABLE

Similar to *CONTAINSTABLE*, except that records are derived based upon a meaning search of '*freetext_string*'. *FREETEXTTABLE* is useful for ad hoc queries against full-text tables, but less accurate than *CONTAINSTABLE*.

OPENDATASOURCE

Provides a means of sourcing data external to SQL Server via OLE DB without declaring a linked server, such as an MS Excel spreadsheet or a Sybase Adaptive Server database table. It is intended for the occasional ad hoc query; if you frequently retrieve result sets from external data sources, you should declare a linked server.

OPENQUERY

Executes a pass-through query against a linked server. It is an effective means of performing a nested table subquery against a data source that is external to SQL Server. The data source must first be declared as a linked server.

OPENROWSET

Executes a pass-through query against an external data source. It is similar to *OPENDATASOURCE*, except that *OPENDATASOURCE* only opens the data source; it does not actually pass through a *SELECT* statement. *OPENROWSET* is intended for occasional, ad hoc usage only.

OPENXML

Provides a queryable, table-like view to an XML string.

[GROUP BY [GROUPING SETS] {grouping_column[, ...] | ALL}] [WITH {CUBE | ROLLUP}]

SQL Server supports the ANSI SQL standard, with some variations. The first noticeable difference is in syntax: where the ANSI standard clause is *GROUP BY {[CUBE | ROLLUP]} (grouping_column[, ...])*, SQL Server uses *GROUP BY [ALL] (group_by_expression) [WITH {CUBE | ROLLUP}]*. The *ALL* keyword is the default behavior and may not be explicitly used with grouping sets, cubes, or rollups. To use grouping sets, you must explicitly use the *GROUPING SETS* subclause. Note that grouping sets may not be nested. (SQL Server also supports the clause *GROUP BY ()*, which returns a grand total for the result set.)

GROUP BY ALL tells SQL Server to provide *group_by* categories even when the matching aggregation is NULL (normally, SQL Server does not return a category whose aggregation is NULL). It must be used only in conjunction with a *WHERE* clause. *WITH {CUBE | ROLLUP}* tells SQL Server to perform additional higher-level aggregates of the summary categories. In the simplest terms, *ROLLUP* produces subtotals for the categories, while *CUBE* produces cross-tabulated totals for the categories.



The *GROUPING* function can be used to help differentiate normally occurring NULLs from the NULLs generated by *ROLLUP* and *CUBE* behavior.

This example groups **royalty** and aggregate **advance** amounts. The *GROUPING* function is applied to the **royalty** column:

```
USE pubs
SELECT royalty, SUM(advance) 'total advance',
       GROUPING(royalty) 'grp'
  FROM titles
 GROUP BY royalty WITH ROLLUP
```

The result set shows two NULL values under **royalty**. The first NULL represents the group of NULL values from this column in the table. The second NULL is in the summary row added by the *ROLLUP* operation. The summary row shows the total **advance** amounts for all **royalty** groups and is indicated by 1 in the **grp** column.

Here is the result set:

royalty	total advance	grp
NULL	NULL	0
10	57000.0000	0
12	2275.0000	0
14	4000.0000	0
16	7000.0000	0
24	25125.0000	0
NULL	95400.0000	1

ORDER BY

Functions normally, as specified by the ANSI SQL standard. However, you should remember that SQL Server allows a variety of *collations* that can affect how the result set is evaluated. Thus, under certain collations “SMITH” and “smith” might evaluate and sort differently. Also note that you cannot *ORDER BY* columns of *TEXT*, *NTEXT*, or *IMAGE* datatype.

COMPUTE {aggregation (expression)}[,...] [BY expression[,...]]

Generates additional aggregations—usually totals—that appear at the end of the result set. *BY expression* adds subtotals and control breaks to the result set. *COMPUTE* and *COMPUTE BY* can be used simultaneously in the same query. *COMPUTE BY* must be coupled with an *ORDER BY* clause, though the *expression* used by *COMPUTE BY* can be a subset of the *order_by_expression*. The aggregation may be any of these function calls: *AVG*, *COUNT*, *MAX*, *MIN*, *STDEV*, *STDEVP*, *VAR*, *VARP*, and *SUM*. Examples are shown later in this section.

COMPUTE, in any form, does not work with the *DISTINCT* keyword or with *TEXT*, *NTEXT*, or *IMAGE* datatypes.

FOR {BROWSE | XML {RAW | AUTO | EXPLICIT}|, XMLDATA|, ELEMENTS|, BINARY BASE64|}

FOR BROWSE is used to allow updates to data retrieved in a DB-Library browse mode cursor. (DB-Library is the original access methodology for SQL Server and has since been supplanted by OLE DB in most applications.) *FOR BROWSE* can only be used against tables with a unique index and a column with the *TIMESTAMP* datatype. *FOR BROWSE* cannot be used in *UNION* statements or when a *HOLDLOCK* hint is active.

FOR XML

Used to extract the result set as an XML document to the SQL Server 2000 client only. You must further define the resulting XML document as either *RAW*, *AUTO*, or *EXPLICIT*. *RAW* transforms each returned row into a generic XML element with the *<row/>>* element tag. *AUTO* transforms the results into a simple, nested XML tree. Finally, *EXPLICIT* transforms the resulting XML tree into an explicitly defined shape. However, the query must be written such that the desired nesting information is specified explicitly. You may optionally attach a few extra control features.

XMLDATA

Returns the schema appended to the XML document. *ELEMENTS* returns the columns as subelements instead of mapping them to XML attributes.

BINARY BASE64

Returns the binary data in base64-encoded format. (Binary is the default format for the *AUTO* mode but must be declared explicitly for *RAW* and *EXPLICIT*.) *FOR XML* cannot be used in any sort of subquery, in conjunction with a *COMPUTE* clause or a *BROWSE* clause, in the definition of a view, in the result set of a user-defined function, or in a cursor. Aggregations and *GROUP BY* are mutually exclusive of *FOR XML AUTO*.

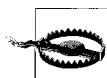
OPTION (hint[, ...])

Replaces elements of the default query plan with your own. Because the optimizer usually picks the best query plan for any query, you are strongly discouraged from placing optimizer hints into your queries. Refer to the SQL Server documentation for more information on hints.

Here's an example of SQL Server's *SELECT...INTO* capability. This example creates a table called **non_mgr_employee** using *SELECT...INTO*. The table contains the **emp_id**, first name, and last name of each non-manager from the **employee** table, joined with their job descriptions (taken from the **jobs** table):

```
-- Query
SELECT  e.emp_id, e.fname, e.lname,
        SUBSTRING(j.job_desc,1,30) AS job_desc
INTO    non_mgr_employee
FROM    employee e
JOIN    jobs AS j ON e.job_id = j.job_id
WHERE   j.job_desc NOT LIKE '%MANAG%'
ORDER BY 2,3,1
```

The newly created and loaded table **non_mgr_employee** now can be queried like any other table.



SELECT...INTO should be used only in development or non-production code because it is not logged or recoverable.

Many of SQL Server's extensions to the ANSI *SELECT* statement involve the *GROUP BY* clause. For example, the *GROUP BY ALL* clause causes the aggregation to include NULL valued results when they would normally be excluded. The following two queries are essentially the same except for the *ALL* keyword, yet they produce very different result sets:

```
-- Standard GROUP BY
SELECT type, AVG(price) AS price
FROM titles
WHERE royalty <= 10
GROUP BY type
ORDER BY type
-- Results
type      price
-----
business  17.3100
mod_cook   19.9900
popular_comp 20.0000
psychology  13.5040
trad_cook   17.9700

-- Using GROUP BY ALL
SELECT type, AVG(price) AS price
FROM titles
WHERE royalty = 10
GROUP BY ALL type
ORDER BY type
-- Results
type      price
-----
business  17.3100
mod_cook   19.9900
popular_comp 20.0000
psychology  13.5040
trad_cook   17.9700
UNDECIDED  NULL
```

COMPUTE has a number of permutations that can impact the result set retrieved by the query. The following example shows the sum of book prices broken out by type of book and sorted by type and then price:

```
-- Query
SELECT type, price
FROM titles
WHERE type IN ('business','psychology')
      AND price > 10
ORDER BY type, price
COMPUTE SUM(price) BY type
-- Results
```

```

type      price
-----
business  11.9500
business  19.9900
business  19.9900
      sum
=====
      51.9300
type      price
-----
psychology 10.9500
psychology 19.9900
psychology 21.5900
      sum
=====
      52.5300

```

The *COMPUTE* clause behaves differently if you do not include *BY*. The following query retrieves the grand total of prices and advances for books with prices over \$16.00:

```

-- Query
SELECT type, price, advance
FROM titles
WHERE price > $16
COMPUTE SUM(price), SUM(advance)
-- Result
type      price      advance
-----
business  19.9900   5000.0000
business  19.9900   5000.0000
mod_cook   19.9900   .0000
popular_comp 22.9500   7000.0000
popular_comp 20.0000   8000.0000
psychology  21.5900   7000.0000
psychology  19.9900   2000.0000
trad_cook   20.9500   7000.0000
      sum
=====
      165.4500
      sum
=====
      41000.0000

```

You can even use *COMPUTE BY* and *COMPUTE* in the same query to produce subtotals and grand totals. (For the sake of brevity, we'll show an example query, but not the result set.) In this example, we find the sum of prices and advances by type for business and psychology books that cost over \$16.00:

```

SELECT type, price, advance
FROM titles
WHERE price > $16
  AND type IN ('business','psychology')
ORDER BY type, price
COMPUTE SUM(price), SUM(advance) BY type
COMPUTE SUM(price), SUM(advance)

```

Don't forget that you must include the *ORDER BY* clause with a *COMPUTE BY* clause! (You do not need an *ORDER BY* clause with a simple *COMPUTE* clause without the *BY* keyword.) There are many permutations that you can perform in a single query—multiple *COMPUTE* and *COMPUTE BY* clauses, *GROUP BY* with a *COMPUTE* clause, and even *COMPUTE* with an *ORDER BY* statement. It's actually fun to tinker around with the different ways you can build queries using *COMPUTE* and *COMPUTE BY*. It's not theme park fun, but what'dya want? This is a programming book!

SQL Server also includes the *FOR XML* clause, which converts the standard result set output into an XML document. This is very useful for web database applications. You can execute queries with *FOR XML* directly against the database or within a stored procedure. For example, we can retrieve one of our earlier example queries as an XML document:

```
SELECT type, price, advance
  FROM titles
 WHERE price > $16
   AND type IN ('business','psychology')
 ORDER BY type, price
 FOR XML AUTO
```

The results aren't particularly pretty, but they're very usable:

```
XML_F52E2B61-18A1-11d1-B105-00805F49916B
-----
<titles type="business" price="19.9900" advance="5000.0000"/><titles type="business" price="19.9900" advance="5000.0000"/>
<titles type="psychology" price="19.9900" advance="2000.0000"/><titles type="psychology" price="21.5900" advance="7000.000
```

If you wanted the XML schema and/or XML elements fully tagged in the output, you could simply append the *XMLDATA* and *ELEMENTS* keywords to the *FOR XML* clause. The query would look like this:

```
SELECT type, price, advance
  FROM titles
 WHERE price > $16
   AND type IN ('business','psychology')
 ORDER BY type, price
 FOR XML AUTO, XMLDATA, ELEMENTS
```

SQL Server also implements a number of other enhancements to support XML. For example, the *OPENXML* rowset function can be used to insert an XML document into a SQL Server table. SQL Server also includes system stored procedures that can help you prepare and manipulate XML documents.

SQL Server (and Oracle, using a somewhat different technique) both support a non-ANSI, non-ISO query known as a *pivot query*. Although you should refer to the vendor documentation for details on exactly how to write a pivot (or unpivot) query, an example here will help you take advantage of this useful technique. A pivot query turns the result set on its side, enabling you to extract more value from the data. For example, the following query produces a two-column, four-row result set:

```
SELECT days_to_make, AVG(manufacturing_cost) AS Avg_Cost
  FROM manufacturing.products
 GROUP BY days_to_make;
```

where the result set is:

days_to_make	Avg_Cost
0	5
1	225
2	350
4	950

By using a pivot query, you can now turn the result “on its side” so that the `days_to_make` column values become the column headings and the query returns one row with five columns:

```
SELECT 'Avg_Cost' As Cost_by_Days, [0], [1], [2], [3], [4]
  FROM (SELECT days_to_make, manufacturing_cost FROM manufacturing.products)
       AS source
      PIVOT
        (AVG(manufacturing_cost) FOR days_to_make IN ([0], [1], [2], [3], [4])) )
     AS pivottable;
```

where the results are:

Cost_by_Days	0	1	2	3	4
Avg_Cost	5	225	350	NULL	950

See Also

JOIN
WHERE

SET Statement

The `SET` statement assigns a value to a runtime variable. The variables may be platform-specific system variables or user-defined variables.

Platform	Command
MySQL	Supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

`SET variable = value`

Keywords

variable

Denotes a system or user-defined variable.

value

Denotes a string or numeric value appropriate to the system or user-defined variable.

Rules at a Glance

Variable values are set for the duration of the session. The value assigned to a variable must match the datatype of the variable. For example, you cannot assign a string value

to a variable that is declared with a numeric datatype. The actual command to create a variable varies from platform to platform. For example, Oracle and SQL Server use the *DECLARE* statement to declare a variable name and datatype, but other platforms may use other means.

The *value* assigned to a variable does not have to be a literal value. It may be a dynamic value that is derived from a subquery. For example, we assign the maximum employee ID to the *emp_id_var* variable in the following example:

```
DECLARE emp_id_var CHAR(5)
SET emp_id_var = (SELECT MAX(emp_id)
                  FROM employees WHERE type = 'F')
```

In this example, an employee type of 'F' indicates that the employee is a full-time, salaried employee.

Programming Tips and Gotchas

The *SET* statement is easily transportable between the database platforms. Only Oracle uses a consistently different scheme for assigning values to variables. In the following example, we declare a variable on SQL Server called *emp_id_var* and assign a value to it:

```
DECLARE emp_id_var CHAR(5)
SET emp_id_var = '67888'
```

Now, we will perform the same action on an Oracle server:

```
DECLARE emp_id_var CHAR(5);
emp_id_var := '67888';
```

You'll find more on these differences in the sections that follow.

MySQL

SET, as a keyword in MySQL, has several uses. First, *SET* is a MySQL datatype that allows multiple comma-separated values. In addition, *SET* may assign a value to a user variable. That latter use is described here, and the syntax for it is:

```
SET @variable = value[, ...]
```

When setting multiple values in a single statement, set each value separately with a comma between them:

```
SET new_var.order_qty = 125, new_var.discount = 4;
```

In addition, MySQL allows the use of *SELECT* to assign values to variables in the same fashion as that described under the ANSI description of *SET*. However, there are some weaknesses when using the *SELECT* method for assigning values to variables. The primary problem is that values are not assigned immediately within the *SELECT* statement. Thus, in the following example:

```
SELECT (@new_var := row_id) AS a,
       (@new_var + 3) AS b
  FROM table_name;
```

the *@new_var* variable will not possess the newly selected value of *row_id* + 3; instead, it will retain the value from the beginning of the statement. For this reason, it is a best practice to assign only one value to a variable at a time.

Oracle

The *SET* clause is not supported as the method of assigning values to variables in Oracle. Instead, user-defined variables are simply assigned values using the assignment indicator `:=`. The basic syntax is shown here:

```
variable := value
```

PostgreSQL

The PostgreSQL command *SET* is used to set a runtime variable:

```
SET [SESSION | LOCAL] variable { TO | = } { value | DEFAULT }
```

The runtime variable may be set to a string literal designated by *value*, or to its default value using the *DEFAULT* keyword. The optional *SESSION* keyword specifies that the command takes effect only for the current session (the default when both *SESSION* and *LOCAL* are omitted). The optional *LOCAL* keyword indicates that the command takes effect only for the current transaction; behavior falls back to whatever the session-level settings are after the current transaction is committed or rolled back.

In addition, some client configuration settings may be configured using *SET* (though you can also use the function *SET_CONFIG* for equivalent functionality). PostgreSQL supports setting additional runtime values for:

[CLIENT_ENCODING] NAMES

Sets multibyte client encoding on PostgreSQL installations built with multibyte support.

DATESTYLE

Sets the style used to display dates and times. Supported styles include:

ISO

Depicts date and time as YYYY-MM-DD HH:MM:SS (the default ISO 8601 style).

SQL

Depicts the Oracle/Ingres style of date and time, not the date and time style mandated by ANSI SQL.

Postgresql

Depicts the long-time date and time format in PostgreSQL (this is no longer the default).

German

Depicts the date and time as DD.MM.YYYY.

You can further qualify the *SQL* and *Postgresql* styles using the keywords *European*, *US*, and *NonEuropean*, which give dates in the forms DD/MM/YYYY, MM/DD/YYYY, and MM/DD/YYYY, respectively (for example, *SET DATESTYLE = SQL, European;*).

SEED

Sets the internal random number generator seed. The value may be any floating-point number between 0 and 1 multiplied by $2^{31}-1$. This value can also be set via the PostgreSQL *setseed* function. For example:

```
SELECT setseed(value);
```

SERVER_ENCODING

Enables multibyte server encoding on servers built with multibyte support.

TIME ZONE {timezone | LOCAL | DEFAULT}

Sets the timezone to *timezone*, or to the local time zone (i.e., the default for the server operating system) using either *LOCAL* or *DEFAULT*. Refer to the vendor documentation for the full listing of allowable time zones.

SQL Server

SQL Server supports *SET* for variable assignments, provided the variables have previously been created with the *DECLARE* statement, and to define values for cursor variables in SQL Server. (SQL Server also uses the *SET* statement for a variety of other purposes, such as enabling or disabling session flags like *SET NOCOUNT ON*.) The platform-specific syntax is:

```
SET { { @variable = value }
  | { @cursor_variable = { @cursor_variable | cursor_name
  | { CURSOR [ FORWARD_ONLY | SCROLL ]
    [ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
    [ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
    [ TYPE_WARNING ]
  FOR select_statement
    [ FOR { READ ONLY | UPDATE [ OF column_name[, ...] ] }
    ] } } }
```

The command does not support the *DEFAULT* keyword, but otherwise supports all the syntax of the ANSI command. The value for *server_name* must reference a connection named in a previous *CONNECT* statement, either as a literal or as a variable.

SET CONNECTION Statement

The *SET CONNECTION* statement allows users to switch between several open connections on one or more database servers.

Platform	Command
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Supported, with limitations

SQL2003 Syntax

```
SET CONNECTION {DEFAULT | connection_name}
```

Keywords*connection_name*

Names the connection of the current session. If the *connection_name* is different from the current session connection name, the connection context is switched to the *connection_name*.

DEFAULT

Switches to the default connection from any other connection. This enables rapid switching to the default connection without knowing its name.

Description

This command does not end a connection. Instead, it switches from the current connection to the connection named (making it the current connection), or from the current connection to the default connection. When switching between connections, the old connection becomes dormant (without committing any changes), while the new connection becomes active.

Rules at a Glance

SET CONNECTION does not create a connection; it merely switches your connection context. Use the *CONNECT* command to create a new connection, and use *DISCONNECT* to terminate a connection.

Programming Tips and Gotchas

The *SET CONNECTION* command is not frequently used, since many users connect programmatically via ODBC, JDBC, or some other connectivity method. However, on those platforms that support *SET CONNECTION*, the command can be very useful for rapidly changing connection properties without terminating any existing connections.

MySQL

Not supported.

Oracle

Not supported.

PostgreSQL

Not supported.

SQL Server

SQL Server supports *SET CONNECTION*, but only in embedded SQL, not within its ad hoc querying tool, SQL Query Analyzer. Although SQL Server supports the full SQL2003 syntax within SQL embedded in other programs, such as a C++ program, it is not used very often. Most prefer the SQL Server-specific command *USE* instead. The platform-specific *SET CONNECTION* syntax is:

```
SET CONNECTION connection_name
```

This command does not support the *DEFAULT* keyword but is otherwise the same as the *ANSI* command. The value for *connection_name* must reference a connection named in a previous *CONNECT* statement, either as a literal or as a variable.

Here is a full T-SQL program in SQL Server that shows *CONNECT*, *DISCONNECT*, and *SET CONNECTION*:

```
EXEC SQL CONNECT TO chicago.pubs AS chicago1 USER sa;
EXEC SQL CONNECT TO new_york.pubs AS new_york1 USER read-only;
-- Opens connections to the servers named "chicago" and
-- "new_york"
EXEC SQL SET CONNECTION chicago1;
EXEC SQL SELECT name FROM employee INTO :name;
-- Sets the chicago1 connection as active and performs work
within that session
```

```

EXEC SQL SET CONNECTION new_york1;
EXEC SQL SELECT name FROM employee INTO :name;
-- sets the new_york1 connection as active and performs work
-- within that session
EXEC SQL DISCONNECT ALL;
-- Terminates all sessions. You could alternately use two
-- DISCONNECT commands, one for each named connection.

```

See Also

[CONNECT](#)
[DISCONNECT](#)

SET CONSTRAINT Statement

The *SET CONSTRAINT* statement defines, for the current transaction, whether a deferrable constraint is checked after each DML statement or when the transaction is finally committed. If the session is not currently in an open transaction, the setting applies to the next transaction.

Platform	Command
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Not supported

SQL2003 Syntax

```
SET CONSTRAINT {constraint_name[, ...] | ALL} {DEFERRED | IMMEDIATE}
```

Keywords

constraint_name[, ...] | *ALL*

Names one or more deferrable constraints where the setting will apply. The keyword *ALL* sets the constraint mode for all deferrable constraints of the current transaction.

DEFERRED

Checks the conditions specified by the deferrable constraint when the transaction is committed, rather than when the DML statements are issued.

IMMEDIATE

Checks the conditions specified by the deferrable constraint immediately after each DML statement is issued, rather than when the transaction is committed.

Rules at a Glance

SET CONSTRAINT defines a value for the constraint mode of all deferrable constraints of the current transaction. If the session is not currently in a transaction, the *SET CONSTRAINT* statement applies to the next transaction issued during the session.

The following example sets all deferrable constraints to be checked immediately following the issuance of each DML statement:

```
SET CONSTRAINT ALL IMMEDIATE;
```

The next example sets two constraints to defer their data modifications until the transaction is committed:

```
SET CONSTRAINT scott.hr_job_title, scott.emp_bonus DEFERRED;
```

Programming Tips and Gotchas

Constraints, when they are defined, may be specified as *DEFERRABLE* or *NOT DEFERRABLE*. The *SET CONSTRAINT* statement will fail if it is issued against a specific constraint that was defined as *NOT DEFERRABLE*.

MySQL

Not supported.

Oracle

Oracle supports the ANSI standard exactly as described, except that *SET CONSTRAINTS* is allowed, as well as *SET CONSTRAINT*.

PostgreSQL

PostgreSQL supports the ANSI-standard syntax exactly as described. Currently, however, PostgreSQL allows *SET CONSTRAINT* only against foreign key constraints, not check and unique constraints, which are always considered *IMMEDIATE*.

SQL Server

Not supported.

SET PATH Statement

The *SET PATH* statement changes the value of the *CURRENT PATH* setting to one or more schemas.

Platform	Command
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
SET PATH schema_name[, ...]
```

Keywords

schema_name[, ...]

Defines one or more schemas as the current path.

Rules at a Glance

SET PATH defines one or more schemas used to qualify an unqualified routine name (that is, functions, procedures, and methods).

The following example sets the current path (i.e., schema name) for unqualified objects to **scott**:

```
SET PATH scott;
```

Then, whenever a routine is referenced during the current session, it will assume the **scott** schema if no schema is identified.

Programming Tips and Gotchas

When referencing multiple schema names, all the schemas must belong to the current database. (The schemas cannot be on a remote database.)

SET PATH does not apply the schema to unqualified objects like tables or views; it only applies to routines.

SET PATH is not supported on any of the platforms discussed in this book.

See Also

[SET SCHEMA](#)

SET ROLE Statement

The *SET ROLE* statement enables and disables specific roles for the current session.

Platform	Command
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
SET ROLE {NONE | role_name}
```

Keywords

NONE

Assigns the *CURRENT ROLE* role to the current session.

role_name

Associates the set of privileges associated with the named role with the current session.

Rules at a Glance

When a user session is opened using the *CONNECT* statement, issuing the *SET ROLE* statement grants that session the privileges associated with a role. The *SET ROLE* command can be issued only outside of a transaction.

The value for *role_name* must reference a valid user role existing on the server. You may specify the role name either as a literal or through a variable.

Programming Tips and Gotchas

Sessions are created using the *CONNECT* statement, while roles are created using the *CREATE ROLE* statement.

Most database platforms offer some method of setting or changing the role used during a user session. *SET ROLE* is the ANSI-standard approach to setting the role used during a user session, but it is not widely supported by the different database platforms. The following sections detail the analogous commands supported by the platforms discussed in this book; check your platform's documentation for more information.

MySQL

SET ROLE is not supported. An analogous method to control connection settings in MySQL is controlled in the [client] section of the *.my.cnf* configuration file in the home directory. For example:

```
[client]
host=server_name
user=user_name
password=client_password
```

To rapidly change between user roles, you can reassign host, user, and password connection properties, respectively, by assigning new values to *MYSQL_HOST*, *USER* (for Windows only), and *MYSQL_PWD* (though *MYSQL_PWD* is insecure in that other users can view this file).

Oracle

When a user initiates a connection, Oracle explicitly assigns roles to that user. The role(s) under which the session is operating can be changed with the *SET ROLE* command, assuming the user is a member of the assigned role. Oracle uses the *MAX_ENABLED_ROLES* initialization parameter (in the *INIT.ORA* file) to control the maximum number of roles that can be enabled concurrently. The Oracle syntax is:

```
SET ROLE { role_name [ IDENTIFIED BY password] [, ...]
| [ALL [EXCEPT role_name[, ...]]
| NONE };
```

Options for the *SET ROLE* command include:

role_name

Declares a valid role name (or names) already created within Oracle, of which the user must already be a member. Any roles not specified here are unavailable throughout the current session. You can enable multiple roles by providing a list, placing a comma between each role listed.

IDENTIFIED BY *password*

If the *role_name* has a password, that password must be specified using this clause.

ALL

Enables all roles that are granted to the current user, including roles that are granted through other roles. Cannot be used with the *IDENTIFIED BY* clause.

EXCEPT

Specifies a list of roles to exclude from a *SET ROLE ALL* command.

NONE

Disables all roles, including the default role.

Roles with passwords may be accessed only through the statement *SET responsibility role_name IDENTIFIED BY password*. For example, we can use this statement to enable the specific roles *read_only* and *updater*—identified by the passwords *editor* and *red_marker*, respectively—for the current session:

```
SET ROLE
  read_only IDENTIFIED BY editor,
  updater   IDENTIFIED BY red_marker;
```

To enable all roles except the *read_write* role, we can issue this command:

```
SET ROLE ALL EXCEPT read_write;
```

PostgreSQL

SET ROLE is not supported in PostgreSQL. However, the ANSI SQL command *SET SESSION AUTHORIZATION* is supported and can achieve somewhat similar results.

SQL Server

Not supported.

See Also

CONNECT
CREATE ROLE
DISCONNECT
SET SESSION AUTHORIZATION

SET SCHEMA Statement

The *SET SCHEMA* statement changes the value of the *CURRENT SCHEMA* setting to a user-specified schema.

Platform	Command
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Not supported
SQL Server	Not supported

SQL2003 Syntax

```
SET SCHEMA schema_name[, ...]
```

Keywords

schema_name[, ...]

Defines one or more schemas as the current path.

Rules at a Glance

SET SCHEMA defines a user-defined schema to use to qualify an unqualified object, such as a table or view.

The following example sets the current schema for unqualified objects to **scott**:

```
SET SCHEMA scott;
```

Then, whenever an object is referenced during the current session, it will assume the `scott` schema if no schema is identified.

Programming Tips and Gotchas

The `SET SCHEMA` statement cannot assign a schema from a remote database as the `CURRENT SCHEMA`.

`SET SCHEMA` does not apply the schema to unqualified routines such as functions, procedures, and methods; it applies only to database objects such as tables and views. This command is not supported on any of the platforms discussed in this book.

See Also

`SET PATH`

SET SESSION AUTHORIZATION Statement

The `SET SESSION AUTHORIZATION` statement sets the user identifier for the current session.

Platform	Command
MySQL	Not supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Not supported

SQL2003 Syntax

`SET SESSION AUTHORIZATION username`

Keywords

username

Sets the session user and the current user of the SQL session to the context of *username*, where *username* may be a literal, a parameter, or a host variable.

Rules at a Glance

This command allows you to switch between users and to run under their permissions.

Programming Tips and Gotchas

Some platforms allow you to use special shortcut keywords like `SESSION USER` and `CURRENT USER`. These are usually the same thing: the username of the currently active session provided by the client. However, `SESSION USER` and `CURRENT USER` can diverge in a session when `SETUID` functions and other similar mechanisms are invoked.

Superuser permissions are required to invoke the `SET SESSION AUTHORIZATION` command, but you will still be able to switch back to the initial user session even if the current user session does not normally have permission to run `SET SESSION AUTHORIZATION`.

You might also wish to check the value of the `SESSION_USER` and `CURRENT_USER` functions with this SQL statement:

```
SELECT SESSION_USER, CURRENT_USER;
```

Normally, you should issue `SET SESSION AUTHORIZATION` before any transactions, to set the session and current user values for all transactions that follow. It must be issued as the only command in its transaction batch.

MySQL

Not supported. You must disconnect your session from MySQL and then reconnect to utilize another set of user privileges.

Oracle

Not supported. Similar functionality is achieved using the `CONNECT` statement or by logging off the server and reconnecting.

PostgreSQL

PostgreSQL supports the ANSI standard for this command. The only difference, and it is a minor one, is that the ANSI standard does not allow this command during a transaction, while PostgreSQL does not care one way or the other.

SQL Server

Not supported. Similar functionality is achieved with the `CONNECT` statement or by logging off the server and reconnecting.

See Also

[CONNECT](#)
[GRANT](#)
[SESSION_USER](#)

SET TIME ZONE Statement

The `SET TIME ZONE` statement changes the current session's time zone if it needs to be different from the default time zone.

Platform	Command
MySQL	Not supported
Oracle	Supported, with variations
PostgreSQL	Supported, with limitations
SQL Server	Not supported

SQL2003 Syntax

```
SET TIME ZONE {LOCAL | INTERVAL {+ | -}'00:00' [HOUR TO MINUTE]}
```

Keywords

LOCAL

Sets the current session's time zone to that of the local server.

INTERVAL

Specifies the time zone offset from Coordinated Universal Time (UTC) in terms of hours and minutes. The offset can be either an increase (with +) or decrease (with -) in relation to the default time.

HOUR TO MINUTE

Specifies the datatype of the *TIME ZONE* value.

Rules at a Glance

This is a relatively simple command that either sets the user session time zone to that of the server (*LOCAL*), or sets the time zone in relation to Coordinated Universal Time (formerly Greenwich Mean Time, or *GMT*). Thus, an *INTERVAL* of 2 would advance the time zone two hours greater than UTC, while an *INTERVAL* of -6 would reduce the time zone by six hours from UTC to the United States central time zone.

Programming Tips and Gotchas

Like most *SET* commands, *SET TIME ZONE* can be executed only outside of an explicit transaction. In other words, you do not need to encapsulate the command within a *START* or *BEGIN TRAN* and a *COMMIT TRAN* statement.

MySQL

Not supported.

Oracle

In Oracle9*i* and higher, you can use the following *ALTER SESSION* command to set the session time zone:

```
ALTER SESSION
  SET TIME_ZONE = {[+ | -] hh:mm'
                  | LOCAL
                  | DBTIMEZONE
                  | 'region'}
```

Use *LOCAL* to revert back to your session's original default time zone, Use *DBTIMEZONE* to set your session time zone to be the database time zone, and use '*region*' to specify a time zone region name such as '*EST*' or '*PST*'. Use an offset such as '-5:00' to specify your time zone in terms of an hour and minute displacement from UTC. A displacement of '-5:00' means that your time is five hours behind UTC time (e.g., 5:00 A.M. your time is 10:00 A.M. UTC time).

To retrieve a list of valid time zone region names, issue the following query:

```
SELECT tzname FROM v$timezone_names;
```

Both of the following commands set the time zone to Eastern Standard Time. The first command does this by specifying the appropriate displacement from UTC, while the second command specifies the time zone region name:

```
ALTER SESSION SET TIME_ZONE = '-5:00';
ALTER SESSION SET TIME_ZONE = 'EST';
```

Oracle's time zone support is complex. Steven Feuerstein's *Oracle PL/SQL Programming* (O'Reilly) contains a good explanation of it in the chapter on datetime datatypes.

PostgreSQL

PostgreSQL allows a session's time value to be set to the server default by using the *LOCAL* or the *DEFAULT* clause:

```
SET TIME ZONE {'timezone' | LOCAL | DEFAULT};
```

There are some variations from the ANSI standard:

'timezone'

Specifies a time zone by name. The possible values for time zone depend on the operating system. For example, the file */usr/share/zoneinfo* contains the database of time zones for Linux servers.

LOCAL / DEFAULT

Sets the current session's time values to those of the default time zone as determined by the local server.

For example, 'PST8PDT' is a valid time zone for California on Linux systems, while 'Europe/Rome' is a valid time zone for Italy on Linux and other systems. If you specify an invalid time zone, the command sets the time zone to UTC.

When increasing the time zone over UTC, the plus sign is optional.

The following example sets the PostgreSQL time zone to Pacific Standard Time:

```
SET TIME ZONE 'PST8PDT';
```

Next, the time for the current session is returned to the server's default time zone:

```
SET TIME ZONE LOCAL;
```

SQL Server

Not supported.

SET TRANSACTION Statement

The *SET TRANSACTION* statement controls many characteristics of data modification, primarily the read/write characteristics and isolation level of a transaction.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with limitations
PostgreSQL	Supported
SQL Server	Supported, with variations

SQL2003 Syntax

```
SET [LOCAL] TRANSACTION [READ ONLY | READ WRITE]
  [ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
    REPEATABLE READ | SERIALIZABLE}]
  [DIAGNOSTIC SIZE int]
```

Keywords

LOCAL

Changes transaction settings for the current session on the local server only. If this keyword is not specified, the transaction settings for the next transaction are changed, even if the transaction runs on a remote server.

READ ONLY

Sets the next upcoming transaction as a read-only transaction. Once the next transaction is complete, transaction behavior reverts to the default settings.

READ WRITE

Sets the next upcoming transaction so it may perform transactions that read and write data.

ISOLATION LEVEL

Sets the isolation level for the next transaction in the session.

READ COMMITTED

Allows a transaction to read rows written by other transactions only when they have been committed.

READ UNCOMMITTED

Allows a transaction to read rows that have been written, but not committed, by other transactions.

REPEATABLE READ

All sessions can see records that are committed before their first transactions were begun. Other open sessions can see or change only committed rows in the user's current session. Consequently, later transactions can add records that might then be visible to the transactions of earlier sessions, but the other sessions must requery to see those records.

SERIALIZABLE

All sessions can see records that are committed before their first transactions were begun. Before that point, open sessions can see records within other user sessions but cannot insert or update until those sessions' transactions are completed. This is the most restrictive isolation level and the default for SQL2003.

DIAGNOSTIC SIZE *int*

Designates the specific number of error messages (*int*) to capture for a transaction. The *GET DIAGNOSTICS* statement retrieves these error messages.

Rules at a Glance

When issued, *SET TRANSACTION* sets the properties of the next upcoming transaction. Because of this, *SET TRANSACTION* is an interim statement that should be issued after one transaction completes and before the next transaction starts. (To begin a transaction and set its characteristics at the same time, use *START TRANSACTION*.) More than one option may be applied with this command, but only one access mode, isolation level, and diagnostic size may be specified at a time.

The *isolation level* of a transaction specifies the degree of isolation a transaction has from other concurrently running sessions. The isolation level controls:

- Whether rows read and updated by your database session are available to other concurrently running database sessions.
- Whether the update, read, and write activity of other database sessions can affect your database session.

If you are unfamiliar with isolation levels, be sure to read your platform's vendor documentation.

Programming Tips and Gotchas

The *ISOLATION LEVEL* clause controls a number of behaviors and anomalies in a transaction concerning concurrent transactions, including the following:

Dirty reads

Occur when a transaction reads the altered records of another transaction before the other transaction has completed. This allows a data modification to occur on a record that might not be committed to the database.

Nonrepeatable reads

Occur when one transaction reads a record while another modifies it. If the first transaction then attempts to reread the record, it won't be able to find it.

Phantom records

Occur when transaction A reads a group of records, but transaction B adds or changes the data so that more records satisfy the query issued by transaction A. Thus, transaction A may read in the records of transaction B as if they were committed to the database, when in fact the records from transaction B may still be rolled back. Since transaction A is reading records that are not yet permanent, these are called *phantom records*.

Table 3-6 shows the impact of various isolation level settings on the anomalies just listed.

Table 3-6. Isolation level and anomaly impact

Isolation level	Dirty reads	Nonrepeatable reads	Phantom records
READ UNCOMMITTED	Allowed	Allowed	Allowed
READ COMMITTED	Not allowed	Allowed	Allowed
REPEATABLE READ	Not allowed	Not allowed	Allowed
SERIALIZABLE	Not allowed	Not allowed	Not allowed

MySQL

MySQL allows you to set the transaction isolation level for the next individual transaction, the whole session, or globally across the server, as follows:

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL
[READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ
 | SERIALIZABLE]
```

By default, MySQL sets the isolation level for the transaction that immediately follows the statement. The keywords are:

GLOBAL

Sets the transaction isolation level for all subsequent transactions across all user sessions or system threads.

SESSION

Sets the transaction isolation level for all subsequent transactions of the current session.

TRANSACTION ISOLATION LEVEL

Sets a specific transaction isolation level, as described earlier in the section "Keywords." When omitted, MySQL defaults to the *REPEATABLE READ* isolation level.

The *SUPER* privilege is required to set a *GLOBAL* transaction isolation level. You can also set the default isolation level via the *MySQL* command-line executable using the *-transaction-isolation=* switch. Following is an example that sets all subsequent threads (both user and system threads) to a serializable transaction isolation level:

```
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Oracle

Oracle allows you to set a transaction as read-only or read-write, set the transaction isolation level, and specify a specific rollback segment for your transactions:

```
SET TRANSACTION { [ READ ONLY | READ WRITE ]
| [ ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE } ]
| [ USE ROLLBACK SEGMENT segment_name ]
| NAME 'transaction_name' };
```

where:

READ ONLY

Sets the next transaction as read-only and serializable. This option is not available to the user SYS. The only statements permitted in read-only sessions are *SELECT*, *ALTER SESSION*, *ALTER SYSTEM*, *LOCK TABLE*, and *SET ROLE*.

READ WRITE

The default transaction style in Oracle. Allows transactions to read and write data.

READ COMMITTED

The default transaction isolation level in Oracle. The same as the ANSI standard.

SERIALIZABLE

Sets the transaction isolation level to the ANSI serializable level and requires the *COMPATIBLE init* parameter to be set to 7.3.0 or higher.

USE ROLLBACK SEGMENT *segment_name*

Sets the next read/write transaction to be written to a specific Oracle rollback segment identified by *segment_name*. Because it applies only to the current transaction, *USE ROLLBACK SEGMENT* should be the first statement in the transaction. This option is not compatible with the *READ ONLY* option. The rollback segment must already exist, or this statement will fail.

NAME

Assigns a name of 255 characters or less to the current transaction. This option is useful in distributed transaction processing environments for two-phase commits, because it lets you easily identify which local transactions belong to a single distributed transaction.

The *USE ROLLBACK SEGMENT* variant can be useful for performance tuning, as it allows you to direct long-running transactions to rollback segments large enough to hold them, while small transactions can be directed to rollback segments that might be small enough to be retained in the cache.

The *SET TRANSACTION* statement should be the first statement in any SQL batch, but Oracle treats it virtually the same as the *START TRANSACTION* statement, so one could be substituted for the other.

In the following example, the query reports from a bi-weekly process on the *chicago* server while avoiding any impact from other users who might be updating or inserting records:

```
SET TRANSACTION READ ONLY NAME 'chicago';
SELECT prod_id, ord_qty
  FROM sales
 WHERE stor_id = 5;
```

In another case, late-night batch processing might create a huge transaction that would overflow all but the rollback segment created to support that one transaction:

```
SET TRANSACTION USE ROLLBACK SEGMENT huge_tran_01;
```

PostgreSQL

SET TRANSACTION in PostgreSQL impacts only the new transaction you are beginning. Consequently, you may have to issue this statement before each new transaction. The syntax is:

```
SET TRANSACTION ISOLATION LEVEL {READ COMMITTED | SERIALIZABLE};
```

where:

READ COMMITTED

Sets the transaction isolation level to the ANSI level *READ COMMITTED*. This is the default.

SERIALIZABLE

Sets the transaction isolation level to the ANSI level *SERIALIZABLE*.

By default, PostgreSQL supports the *READ COMMITTED* transaction isolation level. You can set the default transaction isolation level for all transactions in the session by using either of the following commands:

```
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL
  { READ COMMITTED | SERIALIZABLE }
SET default_transaction_isolation =
  { 'read committed' | 'serializable' }
```

Of course, you can then override the isolation level of any subsequent transaction using the *SET TRANSACTION* statement.

For example, you can set the next transaction to the serializable transaction isolation level:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Alternately, you could set all the transactions in an entire session to serializable:

```
SET SESSION CHARACTERISTICS AS TRANSACTION
  ISOLATION LEVEL SERIALIZABLE;
```

SQL Server

SET TRANSACTION in SQL Server sets the isolation level for an entire session. All queries that follow a *SET TRANSACTION* statement run under the isolation level set by the statement until it is otherwise changed. The syntax is:

```
SET TRANSACTION ISOLATION LEVEL
  { READ COMMITTED
  | READ UNCOMMITTED
  | REPEATABLE READ
  | SERIALIZABLE}
```

where:

READ COMMITTED

Sets the transaction isolation level to the ANSI level *READ COMMITTED*. This is the default.

READ UNCOMMITTED

Sets the transaction isolation level to the ANSI level *READ UNCOMMITTED*. This has the same effect as the *NOLOCK* optimizer hint.

REPEATABLE READ

Sets the transaction isolation level to the ANSI level *REPEATABLE READ*.

SERIALIZABLE

Sets the transaction isolation level to the ANSI level *SERIALIZABLE*. Similar results can be achieved in SQL Server using the *HOLDLOCK* optimizer hint.

For example, the following command lowers the transaction isolation level for all *SELECT* statements during the session from *READ COMMITTED* to *REPEATABLE READ*:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
GO
```

See Also

[COMMIT](#)
[ROLLBACK](#)

START TRANSACTION Statement

The *START TRANSACTION* statement allows you to perform all the functions of *SET TRANSACTION* while also initiating a new transaction.

Platform	Command
MySQL	Supported, with limitations
Oracle	Not supported
PostgreSQL	Not supported; use <i>BEGIN TRANSACTION</i> instead
SQL Server	Not supported; use <i>BEGIN TRANSACTION</i> instead

SQL2003 Syntax

```
START TRANSACTION [READ ONLY | READ WRITE]
  [ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
    REPEATABLE READ | SERIALIZABLE}]
  [DIAGNOSTIC SIZE int]
```

Keywords**READ ONLY**

Sets the next upcoming transaction as a read-only transaction. Once the next transaction is complete, transaction behavior reverts to the default settings.

READ WRITE

Sets the next upcoming transaction so it may perform transactions that read and write data.

ISOLATION LEVEL

Sets the isolation level for the next transaction in the session.

READ COMMITTED

Allows a transaction to read rows written by other transactions only when they have been committed.

READ UNCOMMITTED

Allows a transaction to read rows that have been written, but not committed, by other transactions.

START TRANSACTION Statement

REPEATABLE READ

All sessions can see records that are committed before their first transactions were begun. Other open sessions can see or change only committed rows in the user's current session. Consequently, later transactions can add records that might then be visible to the transactions of earlier sessions, but the other sessions must requery to see those records.

SERIALIZABLE

All sessions can see records that are committed before their first transactions were begun. Before that point, open sessions can see records within other user sessions but cannot insert or update until those sessions' transactions are completed. This is the most restrictive isolation level and the default for SQL2003.

DIAGNOSTIC SIZE *int*

Designates the specific number of error messages (*int*) to capture for a transaction. The *GET DIAGNOSTICS* statement retrieves these error messages.

Rules at a Glance

According to the ANSI standard, the only difference between *SET* and *START* is that *SET* is considered outside of the current transaction, while *START* is considered the beginning of a new transaction. Thus, *SET TRANSACTION* settings apply to the next transaction, while *START TRANSACTION* settings apply to the current transaction.

While only MySQL supports the *START TRANSACTION* statement, three of the vendors (MySQL, PostgreSQL, and SQL Server) support a similar command, *BEGIN [TRAN|SACTION]* and its synonym *BEGIN [WORK]*. *BEGIN TRANSACTION* declares an explicit transaction, but it does not set isolation levels. The only significant rule of the *START TRANSACTION* statement is that it is used to control the access mode, isolation level, and/or diagnostic size of the current transaction only. Once a new transaction starts, you must either issue new values for the setting(s) or rely on the defaults.

Most database platforms allow you to implicitly control transactions, using what is commonly called *autocommit* mode. In autocommit mode, the database treats each statement as a transaction in and of itself, complete with implicit *BEGIN TRAN* and *COMMIT TRAN* statements.

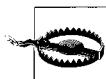
The alternative to autocommit mode is to manually control each transaction. Under explicit transaction control, you declare each new transaction with the *START TRANSACTION* statement. A new transaction may also start implicitly any time a transaction-initiating statement is issued, such as *INSERT*, *UPDATE*, *DELETE*, or *SELECT*. The transaction is not committed or rolled back until either a *COMMIT* or *ROLLBACK* statement is explicitly issued.

Oracle does not support the explicit declaration of a new transaction using *START TRANSACTION*, but it does support explicitly committing, savepointing, and rolling back a transaction. Other platforms, including MySQL, PostgreSQL, and SQL Server, allow you both to explicitly declare a transaction with *START TRANSACTION* and to explicitly commit, savepoint, and roll back the transaction.

Programming Tips and Gotchas

Many of the platforms discussed in this book run in autocommit mode by default. Therefore, it is a good rule of thumb to use explicitly declared transactions only if you intend to do so for all transactions in a session. In other words, do not mix implicitly declared transactions and explicitly declared transactions in a single session.

Each transaction that is explicitly declared can only be made permanent with the *COMMIT* statement. Similarly, any transaction that fails or needs to be discarded must be explicitly undone with the *ROLLBACK* statement.



Be sure to issue *START* in a pair with either *COMMIT* or *ROLLBACK*. Otherwise, the DBMS may not complete the transaction(s) until it encounters a *COMMIT* or *ROLLBACK* statement. Poor planning and omitting timely *COMMITs* (or *ROLLBACKs*) could potentially lead to huge transactions.

It is a good idea to issue explicit *COMMITs* or *ROLLBACKs* after one or a few statements, because long-running transactions can lock up resources, thus preventing other users from accessing those resources. Long-running or very large transaction batches can fill up the rollback segments or transaction logs of a database, even if those files are small.

MySQL

MySQL normally runs in autocommit mode, which means changes are automatically saved to disk when completed. If a change fails for any reason, it is automatically rolled back.

MySQL supports *START TRANSACTION* and a synonym, *BEGIN*. You can suspend autocommit for one or several statements using the *BEGIN* syntax:

```
START TRANSACTION [WITH CONSISTENT SNAPSHOT]
BEGIN [WORK]
```

where:

WITH CONSISTENT SNAPSHOT

Starts a consistent read of data on engines that support consistency of reads (currently, only InnoDB) when running under a transaction isolation level that supports consistent reads (i.e., *SERIALIZABLE* and *REPEATABLE READ*).

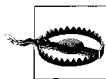
BEGIN [WORK]

Marks the beginning of one or more transactions. *WORK* is an optional keyword with no effect.

Issuing the following command can disable autocommit mode for all sessions and threads:

```
SET AUTOCOMMIT=0
```

Once you have disabled autocommit, the *COMMIT* statement is required to store any and every data modification to disk, and the *ROLLBACK* statement is required to undo changes made during a transaction. Disabling autocommit is only effective with “transaction-safe tables,” such as InnoDB or BDB tables. Disabling autocommit on non-transaction-safe tables has no effect—autocommit will still be enabled.



Earlier versions of MySQL use an update log. However, the update log does not support ANSI transactions unless the tables are defined as InnoDB or NDB Cluster tables.

START TRANSACTION Statement > Oracle

Transactions are stored in the binary log in a single write operation when the *COMMIT* statement is issued. Here's an example:

```
BEGIN;
  SELECT @A := SUM(salary)
  FROM employee
  WHERE type=1;
  UPDATE payhistory SET summary=@A WHERE type=1;
COMMIT;
```

Rollbacks issued against non-transactional tables will fail with the error *ER_WARNING_NOT_COMPLETE_ROLLBACK*, but transaction-safe tables will be restored as expected.

Oracle

Not supported. Transactions in Oracle are started implicitly. Refer to the "Oracle" section in the earlier discussion of *SET TRANSACTION* for more information about how Oracle controls individual transactions.

PostgreSQL

PostgreSQL does not support *START TRANSACTION*. Instead, the PostgreSQL syntax is:

```
BEGIN [ WORK | TRANSACTION ]
```

where:

BEGIN

Marks the beginning of one or more transactions.

WORK

Is an optional keyword with no effect.

TRANSACTION

Is an optional keyword with no effect.

PostgreSQL normally runs in autocommit mode, where each data-modification statement or query is its own transaction. PostgreSQL applies an implicit *COMMIT* for any transaction that completes without an error, and an implicit *ROLLBACK* for any statement that fails. The *BEGIN* statement allows explicit *COMMIT* or *ROLLBACK* of a transaction, which may then consist of multiple statements.

Manually coded transactions are much faster in PostgreSQL than autocommitted transactions. *SET TRANSACTION ISOLATION LEVEL* should be set to *SERIALIZABLE* just after the *BEGIN* statement to bolster the transaction isolation level. PostgreSQL allows many data-modification statements (*INSERT*, *UPDATE*, *DELETE*) within a *BEGIN...COMMIT* block. However, when the *COMMIT* command is issued, either all or none of the transaction is committed, depending on the success or failure of the command.



BEGIN has a separate usage on database platforms that support their own procedural languages (namely, Oracle and SQL Server). On these platforms, *BEGIN* (without the keyword *TRANSACTION*), is used to mark a new block of procedural code. For this reason, you are advised to include the *TRANSACTION* keyword for any transactions you write on PostgreSQL. Otherwise, you will face some complicated migration issues should you ever move your code to Oracle or SQL Server.

Following is an example of *BEGIN TRANSACTION* in PostgreSQL:

```
BEGIN TRANSACTION;
  INSERT INTO jobs(job_id, job_desc, min_lvl, max_lvl)
  VALUES(15, 'Chief Operating Officer', 185, 135)
COMMIT;
```

SQL Server

Microsoft SQL Server supports the *BEGIN TRANSACTION* statement rather than the ANSI *START TRANSACTION* statement. It also supports a couple of extensions that facilitate transaction backup and recovery. The Microsoft SQL Server syntax is:

```
BEGIN TRAN[SACTION] [transaction_descriptor
  [WITH MARK ['log_descriptor']]]
```

where:

TRAN[SACTION]

Marks the beginning of a transaction. SQL Server allows either *TRAN* or *TRANSACTION*.

transaction_descriptor

A name or variable string datatype (CHAR, NCHAR, VARCHAR, or NVARCHAR) variable, of up to 32 characters in size, used to identify a transaction. When working with nested transactions, only name the outermost transaction.

WITH MARK log_descriptor

Tells SQL Server to place a mark of name *log_descriptor* in the transaction log, allowing SQL Server to restore a transaction log up to that point. In a sense, this allows point-in-time recovery based on the name of the mark for databases set to *FULL* recovery mode. *WITH MARK* must be used in conjunction with a named transaction.

When nesting transactions, only the outermost *BEGIN...COMMIT* or *BEGIN...ROLLBACK* pair should reference the transaction name (if it has one). In general, we recommend avoiding nested transactions.

Here is a SQL Server set of *INSERT* statements, all performed as a single transaction:

```
BEGIN TRANSACTION
  INSERT INTO sales VALUES('7896','JR3435','Oct 28 2003',25,
    'Net 60','BU7832')
  INSERT INTO sales VALUES('7901','JR3435','Oct 28 2003',17,
    'Net 30','BU7832')
  INSERT INTO sales VALUES('7907','JR3435','Oct 28 2003',6,
    'Net 15','BU7832')
COMMIT
GO
```

If for some reason any one of these *INSERT* statements had to wait for completion, they would all have to wait, since they are treated as a single transaction.

See Also

COMMIT
ROLLBACK

SUBQUERY Substatement

A subquery is a nested query. Subqueries may appear in various places within a SQL statement.

Platform	Command
MySQL	Supported, with limitations
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL supports the following types of subquery:

Scalar subqueries

Subqueries that retrieve a single value. These are the most widely supported type of subquery among the various database platforms.

Table subqueries

Subqueries that retrieve more than one value or row of values.

Nested table subqueries

Subqueries that retrieve more than one column and more than one row.

Scalar and vector subqueries can, on some platforms, appear as part of the expression in a *SELECT* list of items, a *WHERE* clause, or a *HAVING* clause. Nested table subqueries tend to appear in the *FROM* clauses of *SELECT* statements.

A *correlated subquery* is a subquery that is dependent upon a value in an outer query. Consequently, the inner query is executed once for every record retrieved in the outer query. Since subqueries can be nested many layers deep, a correlated subquery may reference any level in the main query higher than that is than its own level.

Different rules govern the behavior of a subquery, depending on the clause in which it appears. The level of support amongst the database platforms also varies: some platforms support subqueries in all clauses mentioned earlier (*SELECT*, *FROM*, *WHERE*, and *HAVING*), while others support subqueries in only one or two of the clauses.

Subqueries are usually associated with the *SELECT* statement. Since subqueries may appear in the *WHERE* clause, they can be used in any SQL statement that supports a *WHERE* clause, including *SELECT*, *INSERT...SELECT*, *DELETE*, and *UPDATE* statements.

SQL2003 Syntax

Scalar, table, and nested table subqueries are represented in the following generalized syntax:

```
SELECT column1, column2, ... (scalar subquery)
  FROM table1, ... (nested table subquery)
    AS subquery_table_name]
  WHERE foo = (scalar subquery)
    OR foo IN (table subquery)
```

Correlated subqueries are more complex because the values of such subqueries are dependent on values retrieved in their main queries. For example:

```
SELECT column1
  FROM table1 AS t1
 WHERE foo IN
    (SELECT value1
      FROM table2 AS t2
     WHERE t2.pk_identifier = t1.fk_identifier)
```

Note that the *IN* clause is for example purposes only. Any comparison operator may be used.

Keywords

scalar subquery

Includes a scalar subquery in the *SELECT* item list or in the *WHERE* or *HAVING* clause of a query.

nested table subquery

Includes a nested table subquery only in the *FROM* clause in conjunction with the *AS* clause.

table subquery

Includes a table subquery only in the *WHERE* clause with operators such as *IN*, *ANY*, *SOME*, *EXISTS*, or *ALL* that act upon multiple values. Table subqueries return one or more rows containing a single value each.

Rules at a Glance

Subqueries allow you to return one or more values and nest them inside a *SELECT*, *INSERT*, *UPDATE*, or *DELETE* statement, or inside another subquery. Subqueries can be used wherever expressions are allowed. Subqueries also can often be replaced with a *JOIN* statement. Depending on the DBMS, subqueries may perform less quickly than joins.



Subqueries are always enclosed in parentheses.

Subqueries may appear in a *SELECT* clause with an item list containing at least one item, in a *FROM* clause for referencing one or more valid tables or views, and in *WHERE* and *HAVING* clauses.

Scalar subqueries can return only a single value. Certain operators in a *WHERE* clause, such as *=*, *<*, *>*, *>=*, *<=*, and *<>* (or *!=*), expect only one value. If a subquery returns more than one value against an operator that expects a single value, the entire query will fail. On the other hand, table subqueries may return multiple values, but they are usable only with multivalue expressions like *[NOT] IN*, *ANY*, *ALL*, *SOME*, or *[NOT] EXISTS*.

Nested table subqueries may appear only in the *FROM* clause and should be aliased by the *AS* clause. The result set returned by the nested table subquery, sometimes called a *derived table*, offers similar functionality to a view (see *CREATE VIEW*). Every column returned in the derived table need not be used in the query, though they can all be acted upon by the outer query.

Correlated subqueries typically appear as a component of a *WHERE* or *HAVING* clause in the outer query (and, less commonly, in the *SELECT* item list) and are correlated through the *WHERE* clause of the inner query (that is, the subquery).

SUBQUERY Substatement

(Correlated subqueries can also be used as nested table subqueries, though this is less common.) Be sure to include in such a subquery a *WHERE* clause that evaluates based on a correlating value from the outer query; the example for a correlated query in the earlier ANSI syntax diagram illustrates this requirement.

It is also important to specify a table alias, called a *correlation name*, using the *AS* clause or other alias shortcut for every table referenced in a correlated query, both in the outer and inner query. Correlation names avoid ambiguity and help the DBMS quickly resolve the tables involved in the query.

All ANSI-compliant subqueries comply with the following short list of rules:

- A subquery cannot include an *ORDER BY* clause.
- A subquery cannot be enclosed in an aggregate function. For example, the following query is invalid: `SELECT foo FROM table1 WHERE sales >= AVG(SELECT column1 FROM sales_table...)`. You can get around this limitation by performing the aggregation in the subquery rather than in the outer query.

Programming Tips and Gotchas

For most vendor platforms, subqueries should not reference large object datatypes (e.g., *CLOB* or *BLOB* on Oracle and *IMAGE* or *TEXT* on SQL Server) or array datatypes (such as *TABLE* or *CURSOR* on SQL Server).

The platforms all support subqueries, but not every vendor supports every type of subquery. Table 3-7 tells you whether your platform supports each type of subquery.

Table 3-7. Platform-specific subquery support

Platform	MySQL	Oracle	PostgreSQL	SQL Server
Scalar subquery in <i>SELECT</i> item list	✓	✓	✗	✓
Scalar subquery in <i>WHERE/HAVING</i> clause	✓	✓	✓	✓
Vector subquery in <i>WHERE/HAVING</i> clause	✓	✓	✓	✓
Nested table in <i>FROM</i> clause	✓	✓	✓	✓
Correlated subquery in <i>WHERE/HAVING</i> clause	✓	✓	✓	✓

Subqueries are not relegated to *SELECT* statements only. They may also be used in *INSERT*, *UPDATE*, and *DELETE* statements that include a *WHERE* clause. Subqueries are often used for the following purposes:

- To identify the rows inserted into the target table using an *INSERT...SELECT* statement, a *CREATE TABLE...SELECT* statement, or a *SELECT...INTO* statement
- To identify the rows of a view or materialized view in a *CREATE VIEW* statement
- To identify value(s) assigned to existing rows using an *UPDATE* statement
- To identify values for conditions in the *WHERE* and *HAVING* clauses of *SELECT*, *UPDATE*, and *DELETE* statements
- To build a view of a table(s) on the fly (i.e., nested table subqueries)

Examples

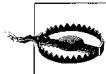
This section shows subquery examples that are equally valid on MySQL, Oracle, PostgreSQL, and SQL Server.

A simple scalar subquery is shown in the `SELECT` item list of the following query:

```
SELECT job, (SELECT AVG(salary) FROM employee) AS "Avg Sal"
  FROM employee
```

Nested table subqueries are functionally equivalent to querying a view. In the following, we query the education level and salary in a nested table subquery, and then perform aggregations on the values in the derived table in the outer query:

```
SELECT AVG(edlevel), AVG(salary)
  FROM (SELECT edlevel, salary
         FROM employee) AS emprand
 GROUP BY edlevel
```



Remember that this query may fail, depending on the platform, without the `AS` clause to associate a name with the derived table.

The following query shows a standard table subquery in the `WHERE` clause expression. In this case, we want all project numbers for employees in the department 'A00':

```
SELECT projno
  FROM emp_act
 WHERE empno IN
  (SELECT empno
    FROM employee
   WHERE workdept = 'A00')
```

The above subquery is executed only once for the outer query.

In the next example, we want to know the names of employees and their level of seniority. We get this result set through a correlated subquery:

```
SELECT firstname, lastname,
  (SELECT COUNT(*)
    FROM employee, senior
   WHERE employee.hiredate > senior.hiredate) as senioritytype
  FROM employee
```

Unlike the previous subquery, this subquery is executed one time for every row retrieved by the outer query. In a query like this, the total processing time could be very long, since the inner query may potentially execute many times for a single result set.

Correlated subqueries depend on values retrieved by the outer query before being able to complete the processing of the inner query. They are tricky to master, but they offer unique programmatic capabilities. The following example returns information about orders where the quantity sold in each order is less than the average quantity in other sales for that title:

```
SELECT s1.ord_num, s1.title_id, s1.qty
  FROM sales AS s1
 WHERE s1.qty <
  (SELECT AVG(s2.qty)
    FROM sales AS s2
   WHERE s2.title_id = s1.title_id)
```

For this example, you can accomplish the same functionality using a self-join. However, there are situations in which a correlated subquery may be the only easy way to do what you need.

SUBQUERY Substatement > MySQL

The next example shows how a correlated subquery might be used to update values in a table:

```
UPDATE course SET ends =
  (SELECT min(c.begins) FROM course AS c
  WHERE c.begins BETWEEN course.begins AND course.ends)
WHERE EXISTS
  (SELECT * FROM course AS c
  WHERE c.begins BETWEEN course.begins AND course.ends)
```

Similarly, you can use a subquery to determine which rows to delete. This example uses a correlated subquery to delete rows from one table based on related rows in another table:

```
DELETE FROM course
WHERE EXISTS
  (SELECT * FROM course AS c
  WHERE course.id > c.id
  AND (course.begins BETWEEN c.begins
  AND c.ends OR course.ends BETWEEN c.begins AND c.ends))
```

MySQL

MySQL supports nested table subqueries as select items and in the *WHERE* clause.

Oracle

Oracle supports ANSI-standard subqueries, though it uses a different nomenclature. In Oracle, a nested table subquery that appears in the *FROM* clause is called an *inline view*. That makes sense, because nested table subqueries are basically views built on the fly. Oracle calls a subquery that appears in the *WHERE* clause or the *HAVING* clause of a query a *nested subquery*. It allows correlated subqueries in the *SELECT* item list and in the *WHERE* and *HAVING* clauses.

PostgreSQL

PostgreSQL supports ANSI-standard subqueries in the *FROM*, *WHERE*, and *HAVING* clauses. However, subqueries appearing in a *HAVING* clause cannot include *ORDER BY*, *FOR UPDATE*, or *LIMIT* clauses. PostgreSQL does not currently support subqueries in the select item list.

SQL Server

SQL Server supports ANSI-standard subqueries. Scalar subqueries can be used almost anywhere a standard expression is allowed. Subqueries in SQL Server cannot include the *COMPUTE* or *FOR BROWSE* clauses. They can include the *ORDER BY* clause if the *TOP* clause is also used.

See Also

DELETE
INSERT
SELECT
UPDATE
WHERE

TRUNCATE TABLE Statement

The *TRUNCATE TABLE* statement, a non-ANSI statement, irrevocably removes all rows from a table without logging the individual row deletes. It quickly erases all the records in a table without altering the table structure, taking up little or no space in the redo logs or transaction logs. However, since a truncate operation is not logged, the *TRUNCATE TABLE* statement cannot be rolled back once it is issued.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

De Facto Standard Syntax

Officially, *TRUNCATE TABLE* is not an ANSI-standard command. However, it is a commonly supported statement that follows this standard format:

`TRUNCATE TABLE table_name`

Keywords

table_name

The name of any valid table within the current database or schema context.

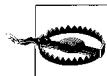
Rules at a Glance

The *TRUNCATE TABLE* statement has the same effect on a table as a *DELETE* statement with no *WHERE* clause; both erase all rows in a given table. However, there are some important differences: *TRUNCATE TABLE* is faster and it is non-logged, meaning it cannot be rolled back if issued in error, and *TRUNCATE TABLE* does not activate triggers, while the *DELETE* statement does.

This command should be issued manually. We strongly encourage you not to place it into automated scripts or production systems that contain irreplaceable data. It cannot be paired with transaction control statements such as *BEGIN TRAN* or *COMMIT*.

Programming Tips and Gotchas

Because the *TRUNCATE TABLE* statement is not logged, it is generally used only in development databases. Use it in production databases with caution!



We strongly advise that you do not write *TRUNCATE TABLE* statements into the stored procedures or functions of a production application, because most platforms do not log the operation and cannot recover an improperly issued *TRUNCATE* statement.

The *TRUNCATE TABLE* command will fail if another user has a lock on the table at the time the statement is issued. *TRUNCATE TABLE* does not activate triggers but will work when they are present. However, it won't work when foreign key constraints are in place on a given table.

MySQL

MySQL versions 3.23 and later support a basic format of the `TRUNCATE TABLE` statement:

```
TRUNCATE TABLE name
```

MySQL achieves the result of a `TRUNCATE TABLE` command by dropping and recreating the affected table. Since MySQL stores each table in a file called `<table_name>.frm`, the `<table_name>.frm` file must exist in the directory containing the database files for the command to work properly.

For example, to remove all data from the `publishers` table:

```
TRUNCATE TABLE publishers
```

Oracle

Oracle allows a table or an indexed cluster (but not a hash cluster) to be truncated. Oracle's syntax is:

```
TRUNCATE { CLUSTER [owner.]cluster
          | TABLE [owner.]table [{PRESERVE | PURGE} MATERIALIZED VIEW LOG] }
          [{DROP | REUSE} STORAGE]
```

`TRUNCATE TABLE` was first introduced by Oracle, and other platforms soon added support for the statement. Oracle has added more features to this statement than are commonly implemented by other vendors. The syntax elements are as follows:

CLUSTER | TABLE

Specifies whether an index cluster or a table will be truncated. An index cluster is a physical construct that stores related rows of two tables next to each other physically on disk to speed up joins by lowering I/O. The `MATERIALIZED VIEW LOG` and `STORAGE` options are not available when truncating an index cluster.

{PRESERVE | PURGE} MATERIALIZED VIEW LOG

Maintains any snapshot logs when a master table is truncated (`PRESERVE`), or clears out any snapshot logs (`PURGE`).

{DROP | REUSE} STORAGE

Causes the disk space freed by the deleted rows to be deallocated (`DROP`), or causes the space allocated to a table to remain allocated to that table, though empty (`REUSE`).

For example:

```
TRUNCATE TABLE scott.authors
  PRESERVE MATERIALIZED VIEW LOG REUSE STORAGE
```

This example command erases all records in the table `scott.authors`. It also maintains the existing snapshot log and allows the table to keep and reuse the storage space already allocated to it.

PostgreSQL

PostgreSQL varies little from the de facto industry standard, but it does add the optional `TABLE` keyword:

```
TRUNCATE [TABLE] name
```

The following command erases all the records in the `authors` table on a PostgreSQL database:

```
TRUNCATE authors
```

SQL Server

SQL Server supports the de facto industry standard.

See Also

DELETE

UNION Set Operator

The *UNION* set operator combines the result sets of two or more queries, showing all the rows returned by each of the queries as one single result set.

UNION is in a class of keyword known as *set operators*. Other set operators include *INTERSECT* and *EXCEPT/MINUS*. (*EXCEPT* and *MINUS* are functionally equivalent; *EXCEPT* is the ANSI standard.) All set operators are used to simultaneously manipulate the result sets of two or more queries; hence the term “set operators.”

Platform	Command
MySQL	Not supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL2003 Syntax

There are technically no limits to the number of queries that you may combine with the *UNION* statement. The general syntax is:

```
<SELECT statement1>
UNION [ALL | DISTINCT]
<SELECT statement2>
UNION [ALL | DISTINCT]
...
...
```

Keywords

UNION

Determines which result sets will be combined into a single result set. Duplicate rows are, by default, excluded.

ALL | DISTINCT

Combines duplicate rows from all result sets (*ALL*) or eliminates duplicate rows from the final result set (*DISTINCT*). Columns containing a NULL value are considered duplicates. If neither *ALL* nor *DISTINCT* is used, *DISTINCT* behavior is the default.

Rules at a Glance

There is only one significant rule to remember when using *UNION*: the order, number, and datatypes of the columns should be the same in all queries.

The datatypes do not have to be identical, but they should be compatible. For example, *CHAR* and *VARCHAR* are compatible datatypes. By default, the result set will default to the largest of two (or more) compatible datatypes, so a query that unions three *CHAR* columns—*CHAR(5)*, *CHAR(10)*, and *CHAR(12)*—will display the results in the *CHAR(12)* format with extra space padded onto the smaller column results.

Programming Tips and Gotchas

Even though the ANSI standard calls for *INTERSECT* to take precedence over other set operators in a single statement, many platforms evaluate all set operators with equal precedence. You can explicitly control the precedence of set operators using parentheses. Otherwise, the DBMS is likely to evaluate them in order from the left-most to the rightmost expression.

Depending on the platform, specifying *DISTINCT* can incur a significant performance cost, since it often involves a second pass through the results to winnow out duplicate records. *ALL* can be specified in any instance where no duplicate records are expected (or where duplicate records are OK) for faster results.

According to the ANSI standard, only one *ORDER BY* clause is allowed in the entire query. Include it at the end of the last *SELECT* statement. To avoid column and table ambiguity, be sure to alias matching columns in each table with the same respective aliases. However, for column-naming purposes, only the aliases in the first query are used for each column in the *SELECT...UNION* query. For example:

```
SELECT au_lname AS "lastname", au_fname AS "firstname"
  FROM authors
  UNION
SELECT emp_lname AS "lastname", emp_fname AS "firstname"
  FROM employees
 ORDER BY lastname, firstname
```

Also be aware that even if the queries in your *UNION* have compatibly datatyped columns, there may be some variation in behavior across the DBMS platforms, especially with regard to the length of the columns. For example, if the **au_lname** column in the first query is markedly longer than the **emp_lname** column in the second query, different platforms may apply different rules as to which length is used. In general, though, the platforms will choose the longer (and less restrictive) column size for use in the result set.

Each DBMS may apply its own rules as to which column name is used if the columns across the tables have different names. In general, the column names of the first query are used.

MySQL

Not supported.

Oracle

Oracle supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
UNION [ALL]
<SELECT statement2>
UNION [ALL]
...
```

Oracle does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent. Oracle does not support *UNION* or *UNION ALL* on the following types of queries:

- Queries containing columns with *LONG*, *BLOB*, *CLOB*, *BFILE*, or *VARRAY* datatypes.
- Queries containing a *FOR UPDATE* clause or a *TABLE* collection expression.

If the first query in the set operation contains any expressions in the select item list, include the *AS* statement to associate an alias with the column. Also, only the last query in the set operation may contain an *ORDER BY* clause. For example, you could find out all unique store IDs without duplicates using this query:

```
SELECT stor_id FROM stores
UNION
SELECT stor_id FROM sales;
```

PostgreSQL

PostgreSQL supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
UNION [ALL]
<SELECT statement2>
UNION [ALL]
...
...
```

PostgreSQL does not support *UNION* or *UNION ALL* on queries with a *FOR UPDATE* clause, and it does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent.

The first query in the set operation may not contain an *ORDER BY* clause or a *LIMIT* clause. Subsequent queries in the *UNION* or *UNION ALL* set operation may contain these clauses, but such queries must be enclosed in parentheses. Otherwise, the right-most occurrence of *ORDER BY* or *LIMIT* will be assumed to apply to the entire set operation.

For example, we could find all authors and all employees whose last names start with “P” with the following query:

```
SELECT a.au_lname
FROM authors AS a
WHERE a.au_lname LIKE 'P%'
UNION
SELECT e.lname
FROM employee AS e
WHERE e.lname LIKE 'W%';
```

SQL Server

SQL Server supports the *UNION* and *UNION ALL* set operators using the basic ANSI SQL syntax:

```
<SELECT statement1>
UNION [ALL]
<SELECT statement2>
UNION [ALL]
...
...
```

SQL Server does not support the *CORRESPONDING* clause. *UNION DISTINCT* is not supported, but *UNION* is the functional equivalent.

UPDATE Statement

You can use *SELECT...INTO* with *UNION* or *UNION ALL*, but *INTO* may appear only in the first query of the union. Special keywords, such as *SELECT TOP* and *GROUP BY... WITH CUBE*, are usable with all queries in a union, but if you use them in one query you must use them with all of the queries. If you use *SELECT TOP* or *GROUP BY... WITH CUBE* in only one query in a union, the operation will fail.

Each query in a union must contain the same number of columns. The datatypes of the columns do not have to be identical, but they must implicitly convert. For example, mixing *VARCHAR* and *CHAR* columns is acceptable. SQL Server uses the larger of the two columns when evaluating the size of the columns returned in the result set. Thus, if a *SELECT...UNION* statement has a *CHAR(5)* column and a *CHAR(10)* column, it will display the data of both columns as a *CHAR(10)* column. Numeric columns are converted to and displayed as the most precise datatype in the union.

For example, the following query unions the results of two independent queries that use *GROUP BY... WITH CUBE*:

```
SELECT ta.au_id, COUNT(ta.au_id)
  FROM pubs..titleauthor AS ta
  JOIN pubs..authors      AS a ON a.au_id = ta.au_id
 WHERE ta.au_id >= '722-51-5454'
  GROUP BY ta.au_id WITH CUBE
UNION
SELECT ta.au_id, COUNT(ta.au_id)
  FROM pubs..titleauthor AS ta
  JOIN pubs..authors      AS a ON a.au_id = ta.au_id
 WHERE ta.au_id < '722-51-5454'
  GROUP BY ta.au_id WITH CUBE
```

See Also

EXCEPT
INTERSECT
MINUS
SELECT

UPDATE Statement

The *UPDATE* statement changes existing data in a table. Use great caution when issuing an *UPDATE* statement without a *WHERE* clause, since the statement then affects every row in the entire table.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL2003 Syntax

```

UPDATE [ONLY] {table_name | view_name}
SET {{column_name = { ARRAY [array_val[, ...]] | DEFAULT |
    NULL | scalar_expression },
    column_name = { ARRAY | DEFAULT |
    NULL | scalar_expression }
    [, ...]}
    | ROW = row_expression}
    [ WHERE search_condition | WHERE CURRENT OF cursor_name ]

```

Keywords

ONLY

Restricts cascading of the updated values to any subtables of the target table or view. *ONLY* affects only typed (object-oriented) tables and views. If used with a nontyped table or view, it causes an error.

table_name | view_name

The target table or view of the *UPDATE* statement. You need appropriate permissions on the target, according to the rules of the platform. Updates against views often have special rules. Generally, it is advisable to perform an *UPDATE* against a view only if the view is representative of a single table.

SET

Assigns a specific value to a column or row.

column_name

Used in conjunction with *SET*, as in *SET column1 = 'foo'*. Allows you to set a new value for the specified column. You can update as many columns as you like in one statement, though you cannot update the same column more than once in a single statement.

ARRAY [array_val[, ...]]

Assigns the column to an array value or to an empty array (using *ARRAY[]*). This behavior is not widely supported by DBMS platforms.

NULL

Sets the column to *NULL*.

DEFAULT

Sets the column to its default value as defined by a *DEFAULT* specification.

scalar_expression

Sets the column to any single value expression, such as a literal string or numeric value, a scalar function, or a scalar subquery.

ROW

Used in conjunction with *SET*, is a mutually exclusive option to *SET column_name*, as in *SET ROW = ROW('foo','bar')*. This behavior is not widely supported by DBMS platforms.

row_expression

Used to set a value for each and every column in the table.

WHERE *search_condition*

Defines search criteria for the *UPDATE* statement using the *search_condition* to ensure that only the target rows are updated. Any legal *WHERE* clause is acceptable. Typically, these criteria are evaluated against each row of the table before the update is applied. If the search criterion is a subquery, the subquery is executed on each row, which is potentially a long-running process.

WHERE CURRENT OF *cursor_name*

Restricts the *UPDATE* statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

The *UPDATE* statement can be used to modify the value of one or more columns of a table or view at a time. Typically, you will update the values in one or more rows at a time. The new values must be scalar (except in row expressions and arrays). That is, a given field/column must have a single, constant value at the time the transaction is executed, though it can be literal or derived from a function call or subquery.

A basic *UPDATE* statement without a *WHERE* clause looks like this:

```
UPDATE authors
  SET contract = 0
```

This example sets the contract status of all authors in the **authors** table to zero (meaning they don't have contracts anymore). Similarly, values can be adjusted mathematically with an *UPDATE* statement:

```
UPDATE titles
  SET price = price * 1.1
```

This *UPDATE* statement would increase all book prices by 10%.

You can also set multiple values in one statement, as shown here, where we set all author last names and first names to uppercase letters:

```
UPDATE authors SET au_lname = UPPER(au_lname),
  au_fname = UPPER(au_fname);
```

Adding a *WHERE* clause to an *UPDATE* statement allows records in the table to be modified selectively:

```
UPDATE titles
  SET type = 'pers_comp',
  price = (price * 1.15)
 WHERE type = 'popular_com';
```

This query makes two changes to any record of the type 'popular_com': it increases the price by 15% and alters the type to 'pers_comp'.

You can also invoke functions or subqueries to derive the values used in an *UPDATE* statement. In some cases, you may wish to update the specific row that is being processed by a declared and open cursor. The following example shows both concepts:

```
UPDATE titles SET ytd_sales = (SELECT SUM(qty)
  FROM sales
  WHERE title_id = 'TC7777')
 WHERE CURRENT OF title_cursor;
```

This query assumes that you have declared and opened a cursor named `title_cursor` and that it is processing titles with IDs of 'TC7777'.

Sometimes you need to update values in a given table based on the values stored in another table. For example, if you need to update the publication date for all the titles written by a certain author, you might find the author and a list of her titles first through subqueries:

```
UPDATE titles
SET pubdate = 'Jan 01 2002'
WHERE title_id IN
  (SELECT title_id
   FROM titleauthor
   WHERE au_id IN
     (SELECT au_id
      FROM authors
      WHERE au_lname = 'White'))
```

Programming Tips and Gotchas

The `UPDATE` statement alters values in existing records of a table or view. If you update a view, the view should include all necessary (*NOT NULL*) columns, or else the statement may fail. Columns with a `DEFAULT` value can usually be safely omitted.

The `SET ROW` clause is not widely supported among DBMS platforms and should be avoided. The `WHERE CURRENT OF` clause is more commonly implemented on DBMS platforms, but it must be used in conjunction with a cursor. Make sure you understand the use and functionality of cursors before using this clause.

In an interactive user session, it is good practice to issue a `SELECT` statement using the same `WHERE` clause before issuing the actual `UPDATE` statement. This precaution enables you to check all rows in the result set before actually performing the `UPDATE`, helping ensure that you don't alter anything you don't mean to alter.

MySQL

MySQL supports the ANSI standard with a few variations, including the `LOW_PRIORITY` clause, the `IGNORE` clause, and the `LIMIT` clause:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_name
SET column_name = {scalar_expression}
[, ...]
WHERE search_conditions
[ORDER BY column_name1 [{ASC | DESC}][, ...]]
[LIMIT integer]
```

where:

LOW_PRIORITY

Tells MySQL to delay the execution of the `UPDATE` statement until no other client is reading from the table.

IGNORE

Tells MySQL to ignore any duplicate key errors generated by `PRIMARY KEY` and `UNIQUE` constraints. However, only records that don't generate such errors will be updated.

ORDER BY

Tells MySQL to update the rows in the specified order (ascending or descending) of the columns specified.

LIMIT integer

Restricts the *UPDATE* action to a specific number of rows, as designated by the *integer* value.

MySQL supports *DEFAULT* value assignment in the *INSERT* statement, but not currently in the *UPDATE* statement.

The following *UPDATE* increases all prices in the **titles** table by 1:

```
UPDATE titles SET price = price + 1;
```

The next example limits the *UPDATE* to the first 10 records encountered in the **titles** table, according to **title_id**. Also, the value of **price** is updated twice. Those two updates of **price** are assessed from left to right. First the price is doubled, and then it is increased by 1:

```
UPDATE titles
  SET price = price * 2,
      price = price + 1
 ORDER BY title_id
 LIMIT 10;
```

Oracle

The Oracle implementation of *UPDATE* allows updates against views, materialized views, subqueries, and tables in an allowable schema:

```
UPDATE [ONLY]
  { [schema.]{view_name | materialized_view_name | table_name}
    [@database_link] [alias]
    {[PARTITION (partition_name)] | [SUBPARTITION (subpartition_name)]}
    | subquery [WITH { [READ ONLY] | [CHECK OPTION [CONSTRAINT constraint_name]] }]
    | [TABLE (collection_expression_name) [ ( + ) ] ] }
  SET {column_name1[, ...]} = {expression[, ...] | subquery} |
    VALUE [(alias)] = {value | (subquery)},
    {column_name2[, ...]} = {expression[, ...] | subquery} |
    VALUE [(alias)] = {value | (subquery)},
    [, ...]
  {WHERE search_conditions | CURRENT OF cursor_name}
  [RETURNING expression[, ...] INTO variable[, ...]];
```

Syntax elements for Oracle's version of *UPDATE* are:

ONLY

Tells Oracle not to update rows from any subviews. Applies only to views that belong to a hierarchy.

materialized_view_name

Applies the *UPDATE* statement to a pre-existing snapshot.

@database_link

Applies the *UPDATE* statement to a view or table on a remote server made available through a pre-existing database link.

alias

Specifies an alias for the table being updated. Oracle allows table aliases, but it does not support the *AS* keyword with *UPDATE* statements. Aliases can also be supplied for the *VALUE* clause of an *UPDATE* statement.

PARTITION

Applies the update to a named partition, rather than to an entire table. You are not required to name a partition when updating a partitioned table, but doing so can, in many cases, help reduce the complexity of your *WHERE* clause.

SUBPARTITION

Applies the update to a named subpartition, rather than to an entire table.

WITH READ ONLY

Specifies that the *subquery* cannot be updated. In some cases, an *UPDATE* transaction may actually alter the records used in a subquery. The *WITH READ ONLY* clause prevents this from happening.

WITH CHECK OPTION

Tells Oracle to abort any changes to the updated table that would not appear in the result set of the subquery.

CONSTRAINT

Tells Oracle to further restrict changes based upon a specific constraint.

SET VALUE

Allows you to set the entire row value for any *TABLE* datatype. The *SET VALUE* clause is very similar to the ANSI *SET ROW* clause. You can also specify a subquery that retrieves all values needed by the *SET VALUE* clause.

RETURNING

Retrieves the rows affected by the command, whereas *UPDATE* normally only shows the number of rows updated. When used for a single-row update, the values of the row can be stored in PL/SQL variables and bind variables. When used for a multirow delete, the values of the rows are stored in bind arrays.

INTO

Indicates the variables into which the values brought back by the *RETURNING* clause should be stored.

Oracle has some important rules about issuing *UPDATE* statements:

- *UPDATE* statements are *not* allowed against tables or the base tables of views containing a domain index with a status of *FAILED* or *IN PROGRESS*. *UPDATE* statements are also not allowed against any table that has an index partition with a status of *UNUSABLE*. (You can avoid the index partition problem by setting the *SKIP_UNUSABLE_INDEXES* session parameter to *TRUE*.)
- Updates against views can be problematic if a view's defining query contains a join, an aggregate function, a set operator, the *DISTINCT* keyword, an *ORDER BY* clause, a *GROUP BY* clause, a *CONNECT BY* clause, a *START WITH* clause, an analytical function, a collection expression, or a subquery. (Under these circumstances, you can use an *INSTEAD OF* trigger to update the tables underlying a view.)

The following code snippets show some examples of the extensions that Oracle offers to the *UPDATE* statement. To begin with, assume that the **sales** table has grown into a large partitioned table. You can update a specific partition using the *PARTITION* clause. For example:

```
UPDATE sales PARTITION (sales_yr_2004) s
  SET s.payterms = 'Net 60'
 WHERE qty > 100;
```

You can also use the *SET VALUE* clause to supply a new value for each column in the row identified by the *WHERE* clause:

```
UPDATE big_sales bs
  SET VALUE(bs) = (SELECT VALUE(s) FROM sales s
                    WHERE bs.title_id = s.title_id
                      AND bs.stor_id = s.stor_id)
 WHERE bs.title_id = 'TC7777';
```

You can return values for review after an update by using the *RETURNING* clause. The following example updates one row, placing the updated values into PL/SQL variables:

```
UPDATE employee
  SET job_id = 13, job_level = 140
 WHERE last_name = 'Josephs'
 RETURNING last_name, job_id
   INTO :var1, :var2;
```

PostgreSQL

PostgreSQL supports the ANSI-standard *UPDATE* syntax with a couple of small variations: it supports the addition of a *FROM* clause and supports an array functionality, but it does not support the *ARRAY* keyword. PostgreSQL's *UPDATE* syntax is:

```
UPDATE [ONLY] {table_name | view_name}
  SET column_name1 = {DEFAULT | NULL | scalar_expression},
      column_name2 = {DEFAULT | NULL | scalar_expression}
      [, ...]
  [FROM {table1 [AS alias], table2 [AS alias][, ...]}]
  [ WHERE search_condition | WHERE CURRENT OF cursor_name ]
```

Most of the syntax elements are the same as in the ANSI standard. The only non-standard element is:

FROM

Provides the ability to build highly selective join-based criteria when determining which rows to update. *FROM* is not needed when only one table (the target table) is used to determine the rows to be updated.

In the following example, we want to update **job_lvl** for employees who have a **job_id** of 12 and a **min_lvl** of 25. We could do this with a large, complex subquery, or we could use the *FROM* clause to enable us to build a join:

```
UPDATE employee
  SET job_lvl = 80
  FROM employee AS e, jobs AS j
 WHERE e.job_id = j.job_id
   AND j.job_id = 12
   AND j.min_lvl = 25;
```

All other aspects of the ANSI standard are fully supported.

SQL Server

SQL Server supports most of the basic components of the ANSI *UPDATE* statement, but it does not support the *ONLY* and *ARRAY* keywords, nor does it support an array update functionality. However, SQL Server has extended the capabilities of *UPDATE* by adding table hints using the *WITH* clause, query hints using the *OPTION* clause, as well as more robust variable handling, as follows:

```
UPDATE {table_name | view_name | rowset}
[WITH (hint1, hint2[, ...])]
SET {column_name = {DEFAULT | NULL | scalar_expression}
  | variable_name = scalar_expression
  | variable_name = column_name = scalar_expression[, ...]}
[FROM {table1 | view1 | nested_table1 | rowset1[, ...]}
  [AS alias]
  [JOIN {table2[, ...]}]
WHERE {conditions | CURRENT OF [GLOBAL] cursor_name}
[OPTION (hint1, hint2[, ...])]
```

SQL Server's *UPDATE* syntax elements are as follows:

WITH *hint*

Allows the use of table hints to override the default behavior of the query optimizer. Since the query optimizer is quite good at choosing query plans, use hints only with a deep understanding of the tables, indexes, and data affected by the operation. Without this understanding, including hints could result in a decrease rather than an increase in performance.

variable_name

SQL Server variables must be declared prior to the *UPDATE* statement, in the form *DECLARE @variable*. The construct *SET @variable = column1 = expression1* sets a variable to the final value of an updated column, whereas *SET @variable = column1, column1 = expression* sets the variable to the value of the column before execution of the *UPDATE* statement.

FROM

Provides the ability to build highly selective join-based criteria when determining which rows to update. *FROM* is not needed when only one table (the target table) is used to determine the rows to be updated. SQL Server rowset functions are described later in the section.

AS *alias*

Allows you to assign an easy-to-use alias to the table, view, nested table subquery, or rowset function.

JOIN

Provides the ability to use ANSI-standard syntax for joined tables, in conjunction with the *FROM* clause.

GLOBAL

A slight variation on the ANSI-standard *WHERE CURRENT OF* clause. The clause *WHERE CURRENT OF cursor_name*, when used in combination with a cursor, tells SQL Server to update only the single record where the cursor is currently positioned. The cursor is assumed to be a local cursor, but it can be designated a global cursor by using the keyword *GLOBAL*.

OPTION hint

Allows the use of query hints to override the default behavior of the query optimizer. As with the *WITH* clause, which uses table hints, use query hints only if you have a deep understanding of the tables, indexes, and data affected by the operation. Without this understanding, including hints could result in a decrease rather than an increase in performance.

The primary extension to the ANSI standard that Microsoft SQL Server offers in an *UPDATE* statement is a *FROM* clause. This *FROM* clause allows the use of the *JOIN* statement to make it especially easy to update rows in the target table by correlating rows declared in the *FROM* clause with the rows updated by the *UPDATE table_name* component of the statement. The following example shows an update using the ANSI style and a rather cumbersome subquery, followed by an update using SQL Server's *FROM* clause extension to update the result of a table join. Both statements accomplish the same work, but in very different ways:

```
-- ANSI style
UPDATE titles
SET    pubdate = GETDATE()
WHERE  title_id IN
    (SELECT title_id
     FROM  titleauthor
     WHERE au_id IN
        (SELECT au_id
         FROM  authors
         WHERE au_lname = 'White'))
-- Microsoft Transact-SQL style
UPDATE titles
SET    pubdate = GETDATE()
FROM  authors AS a
JOIN  titleauthor AS t2 ON a.au_id = t2.au_id
WHERE  t2.title_id = titles.title_id
      AND a.au_lname  = 'White'
```

Performing this update using the Transact-SQL style is simply a matter of joining two tables—**authors** and **titleauthor**—to the **titles** table. To perform the same operation using ANSI-compliant code, the **au_id** in **author** must first be found and passed up to the **titleauthors** table, and then the **title_id** must be identified and passed up to the main *UPDATE* statement.

The following example updates the **state** column for the first 10 authors from the **authors** table:

```
UPDATE authors
SET state = 'ZZ'
FROM (SELECT TOP 10 * FROM authors ORDER BY au_lname) AS t1
WHERE authors.au_id = t1.au_id
```

The important thing to note about this example is that it is normally difficult to update the first n records in an *UPDATE* statement, unless there is some explicit row sequence you can identify in the *WHERE* clause. However, the nested table subquery in the *FROM* clause uses a *TOP* keyword to return the first 10 records, thereby saving a lot of added programming that would otherwise be required.

See Also

DECLARE
SELECT
WHERE

WHERE Clause

The *WHERE* clause sets the search criteria for an operation such as *SELECT*, *UPDATE*, or *DELETE*. Any records in the target table(s) that do not meet the search criteria are excluded from the operation. The search conditions may include many variations, such as calculations, Boolean operators, and SQL predicates (for example, *LIKE* or *BETWEEN*).

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL2003 Syntax

{ *WHERE search_criteria* | *WHERE CURRENT OF cursor_name* }

Keywords

WHERE search_criteria

Defines search criteria for the statement to ensure that only the target rows are affected.

WHERE CURRENT OF cursor_name

Restricts the operation of the statement to the current row of a defined and opened cursor called *cursor_name*.

Rules at a Glance

WHERE clauses are found in *SELECT* statements, *DELETE* statements, *INSERT...* *SELECT* statements, *UPDATE* statements, and any statement that might have a query or subquery (such as *DECLARE*, *CREATE TABLE*, *CREATE VIEW*, and so forth).

The search conditions, all of which are described in their own entries elsewhere in this book, can include:

All records (=ALL, >ALL, <= ALL, SOME/ANY)

For example, to see publishers who live in the same city as their authors:

```
SELECT pub_name
  FROM publishers
 WHERE city = SOME (SELECT city FROM authors);
```

Combinations (AND, OR, and NOT) and evaluation hierarchy

For example, to see all authors with sales in quantities greater than or equal to 75 units, or co-authors with a royalty of greater than or equal to 60:

```
SELECT a.au_id
  FROM authors AS a
 JOIN titleauthor AS ta ON a.au_id = ta.au_id
 WHERE ta.title_id IN (SELECT title_id FROM sales
                         WHERE qty >= 75)
   OR (a.au_id IN (SELECT au_id FROM titleauthor
                     WHERE royaltyper >= 60)
    AND a.au_id IN (SELECT au_id FROM titleauthor
                     WHERE au_ord = 2));
```

Comparison operators (such as =, <>, <, >, <=, and >=)

For example, to see the last and first names of authors who don't have a contract (i.e., authors with **contract** value of zero):

```
SELECT au_lname, au_fname
  FROM authors
 WHERE contract = 0;
```

Lists (IN and NOT IN)

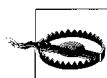
For example, to see all authors who do not yet have a title in the **titleauthor** table:

```
SELECT au_fname, au_lname
  FROM authors
 WHERE au_id NOT IN (SELECT au_id FROM titleauthor);
```

NULL comparisons (IS NULL and IS NOT NULL)

For example, to see titles that have NULL year-to-date sales:

```
SELECT title_id, SUBSTRING(title, 1, 25) AS title
  FROM titles
 WHERE ytd_sales IS NULL;
```



Be sure *not* to specify = NULL in a query. NULL is unknown and can never be equal to anything. Using = NULL is not the same as specifying the *IS NULL* operator.

Pattern matches (LIKE and NOT LIKE)

For example, to see authors whose last names start with a "C":

```
SELECT au_id
  FROM authors
 WHERE au_lname LIKE 'C%';
```

Range operations (BETWEEN and NOT BETWEEN)

For example, to see authors with last names that fall alphabetically between “Smith” and “White”:

```
SELECT au_lname, au_fname
FROM   authors
WHERE  au_lname BETWEEN 'smith' AND 'white';
```

Programming Tips and Gotchas

The *WHERE* clause may require special handling when dealing with certain datatypes, such as *LOBs*, or certain character sets, including *UNICODE*.

Parentheses are used to control evaluation hierarchy within a *WHERE* clause. Encapsulating a clause within parentheses tells the DBMS to evaluate that clause before others. Parentheses can be nested to create a hierarchy of evaluations. The innermost parenthetical clause will be evaluated first. You should watch parentheses very carefully, for two reasons:

- You must always have an equal number of opening and closing parentheses. Any imbalance in the number of opening and closing parentheses will cause an error.
- You should be careful where you place parentheses, since misplacing a parenthesis can dramatically change the result set of your query.

For example, consider again the following query, which returns six rows in the **pubs** database on the SQL Server platform:

```
SELECT DISTINCT a.au_id
FROM   authors AS a
JOIN   titleauthor AS ta ON a.au_id = ta.au_id
WHERE  ta.title_id IN (SELECT title_id FROM sales
                      WHERE qty >= 75)
      OR (a.au_id  IN (SELECT au_id FROM titleauthor
                        WHERE royaltyper >= 60)
       AND   a.au_id  IN (SELECT au_id FROM titleauthor
                        WHERE au_ord = 2))
```

The output from this query is as follows:

```
au_id
-----
213-46-8915
724-80-9391
899-46-2035
998-72-3567
```

Changing just one set of parentheses produces different results:

```
SELECT DISTINCT a.au_id
FROM   authors AS a
JOIN   titleauthor AS ta ON a.au_id = ta.au_id
WHERE  (ta.title_id IN (SELECT title_id FROM sales
                      WHERE qty >= 75)
      OR a.au_id  IN (SELECT au_id FROM titleauthor
                        WHERE royaltyper >= 60))
      AND   a.au_id  IN (SELECT au_id FROM titleauthor
                        WHERE au_ord = 2)
```

This time, the output will look like this:

```
au_id
-----
213-46-8915
724-80-9391
899-46-2035
```

All the platforms discussed in this book support the ANSI standard as described here.

See Also

ALL/ANY/SOME
BETWEEN
DECLARE CURSOR
DELETE
EXISTS
IN
LIKE
SELECT
UPDATE



4

SQL Functions

A function is a special type of command word in the SQL command set, and each SQL dialect varies in its implementation of that command set. In effect, functions are one-word commands that return single values. The value of a function can be determined by input parameters, as with a function that averages a list of database values. However, many functions do not use any type of input parameter. The function that returns the current system time, *CURRENT_TIME*, is an example of such a function.

The ANSI standard supports a number of useful functions. This chapter covers those functions, providing detailed descriptions and examples for each platform. In addition, each database maintains a long list of its own internal functions that are outside the scope of the standard SQL. This chapter provides parameters and descriptions for each database implementation's internal functions.



Most database platforms also support the ability to create user-defined functions (UDFs). For more information on UDFs, refer to Chapter 3.

Types of Functions

There are different ways to categorize functions into groups. The following subsections describe distinctions that are critical to understanding how functions work.

Deterministic and Nondeterministic Functions

Functions can be either deterministic or nondeterministic. A *deterministic function* always returns the same results if given the same input values. A *nondeterministic function* may return different results every time it is called, even when the same input values are provided.

Why is it important that a given input always returns the same output? It is important because of how functions may be used within views, in user-defined functions, and in stored procedures. Restrictions vary across implementations, but these objects sometimes allow only deterministic functions within their defining code. For example, SQL Server allows the creation of an index on a column expression, but only if the expression does not contain any nondeterministic functions. Rules and restrictions vary between the platforms, so check your platform's documentation when using functions.

Aggregate and Scalar Functions

Another way of categorizing functions is in terms of whether they operate on values from just one row at a time, on values from a collection, or on a set of rows. *Aggregate functions* operate against a collection of values and return a single summarizing value. *Scalar functions* return a single value based on scalar input arguments. Some scalar functions, such as *CURRENT_TIME*, do not require any arguments.

Window Functions

Window functions are similar to aggregate functions in that they operate over many rows at one time. The difference lies in how you define those rows. Aggregate functions operate over the sets of rows defined by a query's *GROUP BY* clause. With window functions, you specify the set of rows for each function call, so different invocations of a function within the same query can execute over different sets of rows.

ANSI SQL Aggregate Functions

Aggregate functions return a single value based upon a set of other values. If used among other expressions in the item list of a *SELECT* statement, the *SELECT* must have a *GROUP BY* or *HAVING* clause. No *GROUP BY* or *HAVING* clause is required if the aggregate function is the only value retrieved by the *SELECT* statement. The aggregate functions supported by the ANSI SQL standard and their syntax are listed in Table 4-1.

Table 4-1. ANSI SQL aggregate functions

Function	Usage
<code>AVG(expression)</code>	Computes the average value of a column given by <i>expression</i>
<code>CORR(dependent, independent)</code>	Computes a correlation coefficient
<code>COUNT(expression)</code>	Counts the rows defined by the <i>expression</i>
<code>COUNT(*)</code>	Counts all rows in the specified table or view
<code>COVAR_POP(dependent, independent)</code>	Computes population covariance
<code>COVAR_SAMP(dependent, independent)</code>	Computes sample covariance
<code>CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)</code>	Computes the relative rank of a hypothetical row within a group of rows, where the rank is equal to the number of rows less than or equal to the hypothetical row divided by the number of rows in the group

Table 4-1. ANSI SQL aggregate functions (continued)

Function	Usage
DENSE_RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a dense rank (no ranks are skipped) for a hypothetical row (<i>value_list</i>) in a group of rows generated by GROUP BY
MIN(<i>expression</i>)	Finds the minimum value in a column given by <i>expression</i>
MAX(<i>expression</i>)	Finds the maximum value in a column given by <i>expression</i>
PERCENT_RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group
PERCENTILE_CONT(<i>percentile</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates an interpolated value that, if added to the group, would correspond to the <i>percentile</i> given
PERCENTILE_DISC(<i>percentile</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Returns the value with the smallest cumulative distribution value greater than or equal to <i>percentile</i>
RANK(<i>value_list</i>) WITHIN GROUP (ORDER BY <i>sort_list</i>)	Generates a rank for a hypothetical row (<i>value_list</i>) in a group of rows generated by GROUP BY
REGR_AVGX(<i>dependent</i> , <i>independent</i>)	Computes the average of the independent variable
REGR_AVGY(<i>dependent</i> , <i>independent</i>)	Computes the average of the dependent variable
REGR_COUNT(<i>dependent</i> , <i>independent</i>)	Counts the number of pairs remaining in the group after any pair with one or more NULL values has been eliminated
REGR_INTERCEPT(<i>dependent</i> , <i>independent</i>)	Computes the y-intercept of the least-squares-fit linear equation
REGR_R2(<i>dependent</i> , <i>independent</i>)	Squares the correlation coefficient
REGR_SLOPE(<i>dependent</i> , <i>independent</i>)	Determines the slope of the least-squares-fit linear equation
REGR_SXX(<i>dependent</i> , <i>independent</i>)	Sums the squares of the independent variables
REGR_SXY(<i>dependent</i> , <i>independent</i>)	Sums the products of each pair of variables
REGR_SYy(<i>dependent</i> , <i>independent</i>)	Sums the squares of the dependent variables
STDDEV_POP(<i>expression</i>)	Computes the population standard deviation of all <i>expression</i> values in a group
STDDEV_SAMP(<i>expression</i>)	Computes the sample standard deviation of all <i>expression</i> values in a group
SUM(<i>expression</i>)	Computes the sum of the column values given by <i>expression</i>
VAR_POP(<i>expression</i>)	Computes the population variance of all <i>expression</i> values in a group
VAR_SAMP(<i>expression</i>)	Computes the sample standard deviation of all <i>expression</i> values in a group

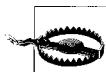
Technically speaking, *ALL*, *ANY*, and *SOME* are considered aggregate functions. However, they have been discussed as range search criteria since they are most often used that way. Refer to Chapter 3 for more information on these functions.

The number of values processed by an aggregate function varies depending on the number of rows queried from the table. This behavior differentiates aggregate functions from scalar functions, which can only operate on the values of a single row per invocation.

The general syntax of an aggregate function is:

aggregate_function_name([ALL | DISTINCT] expression)

The *aggregate_function_name* may be *AVG*, *COUNT*, *MAX*, *MIN*, or *SUM*, as listed in Table 4-1. The *ALL* keyword, which specifies the default behavior, evaluates all rows when aggregating the value of the function. The *DISTINCT* keyword uses only distinct values when evaluating the function.



All aggregate functions except *COUNT(*)* will ignore NULL values when computing their results.

AVG and SUM

The *AVG* function computes the average of values in a column or an expression, and *SUM* computes the sum. Both functions work with numeric values and ignore NULL values. Use the *DISTINCT* keyword to compute the average or sum of all distinct values in a column or expression.

ANSI SQL Standard Syntax

AVG([ALL | DISTINCT] expression)
SUM([ALL | DISTINCT] expression)

MySQL, PostgreSQL, and SQL Server

All of these platforms support the ANSI SQL syntax of *AVG* and *SUM*.

Oracle

Oracle supports the ANSI SQL syntax and the following analytic syntax:

AVG([ALL | DISTINCT] expression) OVER (window_clause)
SUM([ALL | DISTINCT] expression) OVER (window_clause)

For an explanation of the *window_clause*, see the “ANSI SQL Window Functions” section later in this chapter.

Examples

The following query computes average year-to-date sales for each type of book:

```
SELECT type, AVG( ytd_sales ) AS "average_ytd_sales" FROM titles GROUP BY type;
```

This query returns the sum of year-to-date sales for each type of book:

```
SELECT type, SUM( ytd_sales ) FROM titles GROUP BY type;
```

CORR

The CORR function returns the correlation coefficient between a set of dependent and independent variables.

ANSI SQL Standard Syntax

Call the function with two variables, one dependent and the other independent:

`CORR(dependent, independent)`

Any pair in which either the dependent variable, the independent variable, or both are NULL is ignored. The result of the function is NULL when none of the input pairs consists of two non-NULL values.

Oracle

Oracle supports the ANSI SQL syntax, and the following analytic syntax:

`CORR(dependent, independent) OVER (window_clause)`

For an explanation of the `window_clause`, see the “ANSI SQL Window Functions” section later in this chapter.

PostgreSQL

PostgreSQL supports the ANSI SQL syntax of the CORR function.

MySQL and SQL Server

These platforms do not support any form of the CORR function.

Example

The following CORR example uses the data retrieved by the first SELECT:

```
SELECT * FROM test2;
  Y      X
-----
  1      3
  2      2
  3      1

SELECT CORR(y,x) FROM test2;
CORR(Y,X)
-----
-1
```

COUNT

The COUNT function is used to compute the number of rows in an expression.

ANSI SQL Standard Syntax

`COUNT(*)`
`COUNT([ALL | DISTINCT] expression)`

COUNT(*)

Counts all the rows in the target table, regardless of whether they include NULLs.

COUNT([ALL | DISTINCT] expression)

Computes the number of rows with non-NULL values in a specific column or expression. When the keyword *DISTINCT* is used, duplicate values are ignored and a count of the distinct values is returned. *ALL* returns the number of non-NULL values in the expression and is implicit when *DISTINCT* is not used.

MySQL, PostgreSQL, and SQL Server

All of these platforms support the ANSI SQL syntax of COUNT.

Oracle

Oracle supports the ANSI SQL syntax and the following analytic syntax:

```
COUNT ({* | [DISTINCT] expression}) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

Examples

This query counts all the rows in a table:

```
SELECT COUNT(*) FROM publishers;
```

The following query finds the number of different countries where publishers are located:

```
SELECT COUNT(DISTINCT country) "Count of Countries"
  FROM publishers
```

COVAR_POP

The COVAR_POP function returns the population covariance of a set of dependent and independent variables.

ANSI SQL Standard Syntax

Call the function with two variables, one dependent and the other independent:

```
COVAR_POP(dependent, independent)
```

The function disregards any pair in which either the dependent variable, the independent variable, or both are NULL. If no rows remain in the group after NULL elimination, the result of the function is NULL.

Oracle

Oracle supports the ANSI SQL syntax and implements the following analytic syntax:

```
COVAR_POP(dependent, independent) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

PostgreSQL

PostgreSQL supports the ANSI SQL syntax of the COVAR_POP function.

MySQL and SQL Server

These platforms do not support any form of the COVAR_POP function.

Example

The following COVAR_POP example uses the data retrieved by the first *SELECT*:

```
SELECT * FROM test2;
  Y      X
  -----
  1      3
  2      2
  3      1

SELECT COVAR_POP(y,x) FROM test2;
COVAR_POP(Y,X)
-----
-.66666667
```

COVAR_SAMP

The COVAR_SAMP function returns the sample covariance of a set of dependent and independent variables.

ANSI SQL Standard Syntax

Call the function with two variables, one dependent and the other independent:

`COVAR_SAMP(dependent, independent)`

The function disregards any pair in which either the dependent variable, the independent variable, or both are NULL. The result of the function is NULL when none of the input pairs consists of two non-NULL values.

Oracle

Oracle supports the ANSI SQL syntax and implements the following analytic syntax:

`COVAR_SAMP(dependent, independent) OVER (window_clause)`

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

PostgreSQL

PostgreSQL supports the ANSI SQL syntax of the COVAR_SAMP function.

MySQL and SQL Server

These platforms do not support any form of the COVAR_SAMP function.

Example

The following COVAR_SAMP example uses the data retrieved by the first *SELECT*:

```
SELECT * FROM test2;
  Y      X
  -----
  1      3
  2      2
  3      1

SELECT COVAR_SAMP(y,x) FROM test2;
```

CUME_DIST > All Platforms

```
COVAR_SAMP(Y,X)
-----
-1
```

CUME_DIST

Computes the relative rank of a hypothetical row within a group of rows, using the following equation:

$$(rows_preceding_hypothetical + rows_peered_with_hypothetical) / rows_in_group$$

Bear in mind that the *rows_in_group* value includes the hypothetical row that you are proposing when you call the function.

ANSI SQL Standard Syntax

```
CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

Oracle

Oracle follows the ANSI SQL syntax and implements the following analytic syntax:

```
CUME_DIST() OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled “ANSI SQL Window Functions.”

MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *CUME_DIST* aggregate function.

Example

The following example determines the relative rank of the hypothetical new row (*num*=4, *odd*=1) within each group of rows from **test4**, where groups are distinguished by the values in the *odd* column:

```
SELECT * FROM test4;
  NUM      ODD
-----
  0        0
  1        1
  2        0
  3        1
  3        1
  4        0
  5        1
```

```
SELECT odd, CUME_DIST(4,1) WITHIN GROUP (ORDER BY num, odd)
  FROM test4 GROUP BY odd;
          ODD CUME_DIST(4,1)WITHINGROUP(ORDERBYNUM,ODD)
```

```
-----
  0                  1
  1                  .8
```

In the group `odd=0`, the new row comes after the three rows $(0,0)$, $(2,0)$, and $(4,0)$. It will peer with itself. The total number of rows in the group, including the hypothetical row, will be four. The relative rank, therefore, is computed as follows:

$$\begin{aligned} & (3 \text{ rows preceding} + 1 \text{ peering}) / (3 \text{ in group} + 1 \text{ hypothetical}) \\ & = 4 / 4 = 1 \end{aligned}$$

In the group `odd=1`, the new row follows the three rows $(1,1)$, $(3,1)$, and a duplicate $(3,1)$. Again, there is one peer: the hypothetical row itself. The number of rows in the group is five, which includes the hypothetical row. The relative rank is thus:

$$\begin{aligned} & (3 \text{ rows preceding} + 1 \text{ peering}) / (4 \text{ in group} + 1 \text{ hypothetical}) \\ & = 4 / 5 = .8 \end{aligned}$$

DENSE_RANK

Computes a rank in a group for a hypothetical row that you supply. This is a *dense rank*, which means rankings are never skipped, even when a group contains rows that rank identically.

ANSI SQL Standard Syntax

```
DENSE_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

Oracle and SQL Server

Oracle and SQL Server follow the ANSI SQL syntax and implement the following analytic syntax:

```
DENSE_RANK() OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled “ANSI SQL Window Functions.”

MySQL and PostgreSQL

These platforms do not implement the *DENSE_RANK* aggregate function.

Example

The following example determines the dense rank of the hypothetical new row (`num=4, odd=1`) within each group of rows from `test4`, where groups are distinguished by the values in the `odd` column:

```
SELECT * FROM test4;
  NUM      ODD
  ----- -----
    0        0
    1        1
    2        0
    3        1
    3        1
    4        0
    5        1
```

```

SELECT odd, DENSE_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4 GROUP BY odd;
  ODD DENSE_RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
-----
  0                      4
  1                      3

```

In the group `odd=0`, the new row comes after `(0,0)`, `(2,0)`, and `(4,0)`, and thus it is in position 4. In the group `odd=1`, the new row follows `(1,1)`, `(3,1)`, and a duplicate `(3,1)`. In that case, the duplicate occurrences of `(3,1)` both rank #2, so the new row is ranked #3. Compare this behavior with `RANK`, which gives a different result.

MIN and MAX

`MIN(expression)` and `MAX(expression)` find the minimum and maximum values of `expression` (string, datetime, or numeric) in a set of rows. Either `DISTINCT` or `ALL` may be used with these functions, but they do not affect the result.

ANSI SQL Standard Syntax

```

MIN( [ALL | DISTINCT] expression )
MAX( [ALL | DISTINCT] expression )

```

MySQL

MySQL supports the ANSI SQL syntax of `MIN` and `MAX`. MySQL also supports the functions `LEAST` and `GREATEST`, providing the same capabilities.

Oracle

Oracle supports the ANSI SQL syntax and implements the following analytic syntax:

```

MIN({ALL|[DISTINCT] expression}) OVER (window_clause)
MAX({ALL|[DISTINCT] expression}) OVER (window_clause)

```

For an explanation of the `window_clause`, see the section later in this chapter titled “ANSI SQL Window Functions.”

PostgreSQL and SQL Server

These platforms support the ANSI SQL syntax of `MIN` and `MAX`.

Examples

The following query finds the best and worst sales for any title on record:

```

SELECT MIN(ytd_sales), MAX(ytd_sales)
  FROM titles;

```

Aggregate functions are used often in the `HAVING` clauses of queries with `GROUP BY`. The following query selects all categories (types) of books that have an average price for all books in the category higher than \$15.00:

```

SELECT type 'Category', AVG( price ) 'Average Price'
  FROM titles
 GROUP BY type
 HAVING AVG(price) > 15

```

PERCENT_RANK

Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.

ANSI SQL Standard Syntax

```
PERCENT_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

Oracle

Oracle follows the ANSI SQL syntax and implements the following syntax:

```
PERCENT_RANK() OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled “ANSI SQL Window Functions.”

MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *PERCENT_RANK* aggregate function.

Example

The following example determines the percentage rank of the hypothetical new row (num=4, odd=1) within each group of rows from **test4**, where groups are distinguished by the values in the **odd** column:

```
SELECT * FROM test4;
  NUM      ODD
-----
```

0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, PERCENT_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
  FROM test4 GROUP BY odd;
      ODD PERCENT_RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
-----
```

0	1
1	.75

In the group *odd*=0, the new row comes after (0,0), (2,0), and (4,0), and thus it is in position 4. The rank computation is $(4\text{th rank} - 1)/3 \text{ rows} = 100\%$. In the group *odd*=1, the new row follows (1,1), (3,1), and a duplicate (3,1), and is again ranked at #4. The rank computation for *odd*=1 is $(4\text{th rank} - 1)/4 \text{ rows} = 3/4 = 75\%$.

PERCENTILE_CONT

Generates an interpolated value corresponding to a percentile that you specify.

ANSI SQL Standard Syntax

In the following syntax, *percentile* is a number between 0 and 1:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_list)
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_CONT(percentile) WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

See “ANSI SQL Window Functions,” later in this chapter, for a description of partitioning.

MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_CONT*.

Example

The following example groups the data in **test4** by the column named **odd** and invokes *PERCENTILE_CONT* to return a 50th-percentile value for each group:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
FROM test4 GROUP BY odd;
ODD PERCENTILE_CONT(0.50)WITHINGROUP(ORDERBYNUM)
```

0	2
1	3

PERCENTILE_DISC

Determines the value in a group with the smallest cumulative distribution greater than or equal to a percentile that you specify.

ANSI SQL Standard Syntax

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY sort_list)
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

percentile is a number between zero and one.

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_DISC(percentile) WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

See “ANSI SQL Window Functions,” later in this chapter, for a description of partitioning.

MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_DISC*.

Example

The following example is similar to that for *PERCENTILE_CONT*, except that it returns, for each group, the value closest to but not exceeding the 60th percentile:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```
SELECT odd, PERCENTILE_DISC(0.60) WITHIN GROUP (ORDER BY NUM)
FROM test4 GROUP BY odd;
PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
```

2

3

RANK

Computes a rank in a group for a hypothetical row that you supply. This is not a dense rank. If the group contains rows that rank identically, it’s possible for ranks to be skipped. If you want a dense rank, use the *DENSE_RANK* function.

ANSI SQL Standard Syntax

```

RANK(value_list) WITHIN GROUP (ORDER BY sort_list)
value_list ::= expression[, expression...]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]

```

Items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

Oracle and SQL Server

Oracle and SQL Server follow the ANSI SQL syntax and implement the following analytic syntax:

```
RANK() OVER ([partitioning] ordering)
```

For an explanation of the *partitioning* and *order* clauses, see the section later in this chapter titled “ANSI SQL Window Functions.”

MySQL and PostgreSQL

These platforms do not implement the *RANK* aggregate function.

Example

The following example determines the rank of the hypothetical new row (*num*=4, *odd*=1) within each group of rows from **test4**, where groups are distinguished by the values in the **odd** column:

```
SELECT * FROM test4;
```

NUM	ODD
0	0
1	1
2	0
3	1
3	1
4	0
5	1

```

SELECT odd, RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4 GROUP BY odd;
          ODD RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
-----
```

0	4
1	4

In both cases, the rank of the hypothetical new row is 4. In the group *odd*=0, the new row comes after (0,0), (2,0), and (4,0), and thus it is in position 4. In the group *odd*=1, the new row follows (1,1), (3,1), and a duplicate (3,1). In this case, even though two of the rows are duplicates and so have the same rank, the new row is still ranked #4 because it is preceded by three rows. Compare this behavior with that of *DENSE_RANK*.

The REGR Family of Functions

ANSI SQL defines a family of functions, having names beginning with *REGR_*, that relate to different aspects of linear regression. The functions work in the context of a least-squares regression line.

ANSI SQL Standard Syntax

Following is the syntax and a brief description of each *REGR_* function:

REGR_AVGX(dependent, independent)

Averages (as in *AVG(x)*) the *independent* variable values

REGR_AVGY(dependent, independent)

Averages (as in *AVG(y)*) the *dependent* variable values

REGR_COUNT(dependent, independent)

Counts the number of non-NULL number pairs

REGR_INTERCEPT(dependent, independent)

Computes the y-intercept of the regression line

REGR_R2(dependent, independent)

Computes the coefficient of determination

REGR_SLOPE(dependent, independent)

Computes the slope of the regression line

REGR_SXX(dependent, independent)

Sums the squares of the *independent* variable values

REGR_SXY(dependent, independent)

Sums the products of each pair of values

REGR_SYY(dependent, independent)

Sums the squares of the *dependent* variable values

The *REGR_* functions only work on number pairs containing two non-NULL values. Any number pair with one or more NULL values will be ignored.

Oracle and PostgreSQL

Oracle and PostgreSQL support the ANSI SQL syntax for all *REGR_* functions. Oracle also supports the following analytic syntax:

REGR_function(dependent, independent) OVER (window_clause)

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

MySQL and SQL Server

These platforms do not implement the *REGR_* family of functions.

Example

The following *REGR_COUNT* example demonstrates that any pair with one or more NULL values is ignored. The table **test3** contains three non-NULL number pairs, and three other pairs that have at least one NULL:

STDDEV_POP > All Platforms

```
SELECT * FROM test3;
Y      X
-----
1      3
2      2
3      1
4      NULL
NULL    4
NULL    NULL
```

The *REGR_COUNT* function ignores the pairs containing NULLs, counting only those pairs with two non-NULL values:

```
SELECT REGR_COUNT(y,x) FROM test3;
REGR_COUNT(Y,X)
-----
3
```

Likewise, all other *REGR_* functions filter out any pairs containing NULL values before performing their respective computations.

STDDEV_POP

Use *STDDEV_POP* to find the *population standard deviation* within a group of numeric values.

ANSI SQL Standard Syntax

```
STDDEV_POP(numeric_expression)
```

MySQL, Oracle, and PostgreSQL

MySQL, Oracle, and PostgreSQL support the ANSI SQL syntax. Oracle also supports the following analytic syntax:

```
STDDEV_POP(numeric_expression) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

SQL Server

Use the *STDEVP* function.

Example

The following example computes the population standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;
X
-----
1
2
3

SELECT STDDEV_POP(x) FROM test;
STDDEV_POP(X)
-----
.816496581
```

STDDEV_SAMP

Use *STDDEV_SAMP* to find the *sample standard deviation* within a group of numeric values.

ANSI SQL Standard Syntax

`STDDEV_SAMP(numeric_expression)`

MySQL and PostgreSQL

MySQL and PostgreSQL support the ANSI SQL syntax.

Oracle

Oracle supports the standard syntax. It also provides the *STDDEV* function, which operates similarly to *STDDEV_SAMP* except that it returns zero (instead of *NULL*) when there is only one value in the set.

Oracle also supports analytic syntax:

`STDDEV_SAMP(numeric_expression) OVER (window_clause)`

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

SQL Server

Use *STDEV* (with only one *D*!).

Example

The following example computes the sample standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;
  X
-----
  1
  2
  3

SELECT STDDEV_SAMP(x) FROM test;
STDDEV_SAMP(X)
-----
  1
```

VAR_POP

Use *VAR_POP* to compute the population variance of a set of values.

ANSI SQL Standard Syntax

`VAR_POP(numeric_expression)`

MySQL and PostgreSQL

MySQL and PostgreSQL support the ANSI SQL syntax.

Oracle

Oracle supports the standard syntax. It also supports the following analytic syntax:

VAR_POP(*numeric_expression*) OVER (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

SQL Server

Use the VARP function.

Example

The following example computes the population variance for the values 1, 2, and 3:

```
SELECT * FROM test;
X
-----
1
2
3

SELECT VAR_POP(x) FROM test;
VAR_POP(X)
-----
.666666667
```

VAR_SAMP

Use VAR_SAMP to compute the sample variance of a set of values.

ANSI SQL Standard Syntax

VAR_SAMP(*numeric_expression*)

MySQL and PostgreSQL

MySQL and PostgreSQL support the ANSI SQL syntax.

Oracle

Oracle supports the standard syntax. You may alternatively use the VARIANCE function, which differs from VAR_SAMP by returning zero (instead of NULL) for sets that contain only a single value.

Oracle also supports the following analytic syntax:

VAR_SAMP(*numeric_expression*) OVER (*window_clause*)

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

SQL Server

Use the VAR function.

Example

The following example computes the sample variance for the values 1, 2, and 3:

```
SELECT * FROM test;
X
-----
1
2
3

SELECT VAR_SAMP(x) FROM test;
VAR_SAMP(X)
-----
1
```

ANSI SQL Window Functions

ANSI SQL2003 allows for a *window_clause* in aggregate function calls, the addition of which makes those functions into window functions. Both Oracle and SQL Server support this window function syntax. This section describes how to use the *window_clause* within Oracle and SQL Server.



Oracle tends to refer to window functions as *analytic functions*.

Window (or analytic) functions are similar to standard aggregate functions in that they operate on multiple rows, or groups of rows, within the result set returned from a query. However, the groups of rows that a window function operates on are defined not by a *GROUP BY* clause, but by partitioning and windowing clauses. Furthermore, the order within these groups is defined by an ordering clause, but that order affects only function evaluation and has no effect on the order in which rows are returned by the query.



Window functions are the last items in a query to be evaluated, except for the *ORDER BY* clause. Because of this late evaluation, window functions *cannot* be used within the *WHERE*, *GROUP BY*, or *HAVING* clauses.

ANSI SQL2003's Window Syntax

SQL2003 specifies the following syntax for window functions:

```
FUNCTION_NAME(expr) OVER {window_name | (window_specification)}
window_specification ::= [window_name] [partitioning] [ordering] [framing]
partitioning ::= PARTITION BY value[, value...] [COLLATE collation_name]
ordering ::= ORDER [SIBLINGS] BY rule[, rule...]
rule ::= {value | position | alias} [ASC | DESC] [NULLS {FIRST | LAST}]
framing ::= {ROWS | RANGE} {start | between} [exclusion]
start ::= {UNBOUNDED PRECEDING | unsigned-integer PRECEDING | CURRENT ROW}
between ::= BETWEEN bound AND bound
bound ::= {start | UNBOUNDED FOLLOWING | unsigned-integer FOLLOWING}
exclusion ::= {EXCLUDE CURRENT ROW | EXCLUDE GROUP |
               EXCLUDE TIES | EXCLUDE NO OTHERS}
```

Oracle's Window Syntax

Oracle's window function syntax is as follows:

```
FUNCTION_NAME(expr) OVER (window_clause)
window_clause ::= [partitioning] [ordering [framing]]
partitioning ::= PARTITION BY value[, value...]
ordering ::= ORDER [SIBLINGS] BY rule[, rule...]
rule ::= {value | position | alias} [ASC | DESC]
    [NULLS {FIRST | LAST}]
framing ::= {ROWS | RANGE} {not_range | begin AND end}
not_range ::= {UNBOUNDED PRECEDING |
    CURRENT ROW |
    value PRECEDING}
begin ::= {UNBOUNDED PRECEDING |
    CURRENT ROW |
    value {PRECEDING | FOLLOWING}}
end ::= {UNBOUNDED FOLLOWING |
    CURRENT ROW |
    value {PRECEDING | FOLLOWING}}
```

SQL Server's Window Syntax

SQL Server's window function syntax is as follows:

```
FUNCTION_NAME(expr) OVER ([window_clause])
window_clause ::= [partitioning] [ordering]
partitioning ::= PARTITION BY value[, value...]
ordering ::= ORDER BY rule[, rule...]
rule ::= column [ASC | DESC]
```

Partitioning

Partitioning the rows operated on by the partitioning clause is similar to using the `GROUP BY` expression on a standard `SELECT` statement. The partitioning clause takes a list of expressions that will be used to divide the result set into groups. We'll use the following table as the basis for some examples:

```
SELECT * FROM odd_nums;
    NUM      ODD
    -----
    0        0
    1        1
    2        0
    3        1
```

The following results illustrate the effects of partitioning by `ODD`. The sum of the even numbers is 2 (0+2), and the sum of the odd numbers is 4 (1+3). The second column of the result set reports the sum of all values in the partition to which *that row* belongs, yet all the detail rows are returned. The query provides summary results in the context of detail rows:

```
SELECT NUM, SUM(NUM) OVER (PARTITION BY ODD) S FROM ODD_NUMS;
NUM          S
-----
0            2
2            2
1            4
3            4
```

Not using a partitioning clause at all will sum all of the numbers in the **NUM** column for each row returned by the query. In effect, the entire result set is treated as a single, large partition:

```
SELECT NUM, SUM(NUM) OVER () S FROM ODD_NUMS;
NUM          S
-----
0            6
1            6
2            6
3            6
```

Ordering

You specify the order of the rows on which an analytic function operates using the *ordering* clause. However, this analytic ordering clause does not define the ordering of the result set. To define the overall result set ordering, you must use the query's *ORDER BY* clause. The following use of Oracle's *FIRST_VALUE* function illustrates the effects of different orderings of the partitions:

```
SELECT NUM,
       SUM(NUM) OVER (PARTITION BY ODD) S,
       FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM ASC) first_asc,
       FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM DESC) first_desc
  FROM ODD_NUMS;
NUM          S  FIRST_ASC FIRST_DESC
-----
0            2      0        2
2            2      0        2
1            4      1        3
3            4      1        3
```

As you can see, the *ORDER BY* clauses in the window function invocations affect the ordering of the rows in the respective partitions when those functions are evaluated. *ORDER BY NUM ASC* orders partitions in ascending order, resulting in 0 for the first value in the even-number partition and 1 for the first value in the odd-number partition, while *ORDER BY NUM DESC* has the opposite effect.



The preceding query also illustrates an important point: using window functions, you can summarize and order results in many different ways in the same query.

Grouping or Windowing

Many analytic functions also allow you to specify a virtual, moving window surrounding a row within a partition, using the *framing* clause. Such moving windows are useful for calculations such as a running total.

The following Oracle-based example uses the framing clause on the analytic variant of *SUM* to calculate a running sum of the values in the first column. No partitioning clause is used, so each invocation of *SUM* operates over the entire result set. However, the *ORDER BY* clause sorts the rows for *SUM* in ascending order of **NUM**'s value, and the *BETWEEN* clause (which is the windowing clause) causes each invocation of *SUM* to include values for **NUM** only up through the current row. Each successive invocation of *SUM* includes yet another value for **NUM**, in order, from the lowest value of **NUM** to the greatest:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS
    BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS;

```

NUM	S
0	0
1	1
2	3
3	6

This example's a bit too easy, as the order of the final result set happens to match the order of the running total. That doesn't need to be the case. The following example generates the same results, but in a different order. You can see that the running total values are appropriate for each value of **NUM**, but the rows are presented in a different order than before. The result set ordering is completely independent of the ordering used for window function calculations:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS
    BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS
ORDER BY NUM DESC;

```

NUM	S
3	6
2	3
1	1
0	0

List of Window Functions

ANSI SQL specifies that any aggregate function may also be used as a window function. Oracle and SQL Server largely follow the standard in that respect, so you'll find that you can take just about any aggregate function (certainly the standard ones) and apply to it the window function syntax described in the preceding sections.

In addition to the aggregate functions, ANSI SQL defines the window functions described in the following sections. Only Oracle and SQL Server currently implement these functions. All examples use the following table and data, which is a variation on the **ODD_NUMS** table used earlier to illustrate the concepts of partitioning, ordering, and grouping:

```
SELECT * FROM test4;
  NUM      ODD
  -----
  0        0
  1        1
  2        0
  3        1
  3        1
  4        0
  5        1
```

Platform-specific window functions for Oracle (there are none for SQL Server) are included in the lists found in the section “Platform-Specific Extensions,” later in this chapter.

CUME_DIST

Calculates the cumulative distribution, or relative rank, of the current row with regard to other rows in the same partition. The calculation for a given row is as follows:

number of peer or preceding rows / number of rows in partition

Because the result for a given row depends on the number of rows preceding that row in the same partition, it’s important to always specify an *ORDER BY* clause when invoking this function.

ANSI SQL Standard Syntax

`CUME_DIST() OVER {window_name|(window_specification)}`

Oracle

Oracle does not allow the framing portion of the windowing syntax. It requires the *ordering* clause:

`CUME_DIST() OVER ([partitioning] ordering)`

SQL Server

SQL Server does not support *CUME_DIST*.

Example

The following Oracle-based example uses *CUME_DIST* to generate a relative rank for each row, ordering by *NUM*, after partitioning the data by *ODD*:

```
SELECT NUM, ODD, CUME_DIST() OVER
  (PARTITION BY ODD ORDER BY NUM) cumedist
  FROM test4;
  NUM      ODD      CUMEDIST
  -----
  0        0        .3333333333
  2        0        .6666666667
  4        0        1
  1        1        .25
  3        1        .75
  3        1        .75
  5        1        1
```

DENSE_RANK > All Platforms

Following is an explanation of the calculation behind the rank for the row in which **NUM=0**:

1. Because of the *ORDER BY* clause, the rows in the partition are ordered as follows:

NUM=0
NUM=2
NUM=4

2. There are no rows preceding NUM=0.
3. There is one row that is a peer of NUM=0, and that is the NUM=0 row itself. Thus, the divisor is 1.
4. There are three rows in the partition as a whole, making the dividend 3.
5. The result of 1/3 is .33 repeating, as shown in the example output.

DENSE_RANK

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same. Unlike with *RANK*, gaps in rank numbers will not result from two rows sharing the same rank.

ANSI SQL Standard Syntax

```
DENSE_RANK() OVER {window_name | (window_specification)}
```

Oracle and SQL Server

Oracle and SQL Server also require the *ordering* clause and do not allow the *framing* clause:

```
DENSE_RANK() OVER ([partitioning] ordering)
```

Example

Compare the results from the following Oracle-based example to those shown in the section on the *RANK* function:

```
SELECT NUM, DENSE_RANK() OVER (ORDER BY NUM) rank
  FROM test4;
  NUM      RANK
  -----  -----
    0        1
    1        2
    2        3
    3        4
    3        4
    4        5
    5        6
```

The two rows where **NUM=3** are both ranked at #3, and the next-higher row is ranked at #4. Rank numbers are not skipped, hence the term “dense.”

PERCENT_RANK

Computes the relative rank of a row by dividing that row's rank less 1 by the number of rows in the partition, also less 1:

$$(rank - 1) / (rows - 1)$$

Compare this calculation to that used for *CUME_DIST*.

ANSI SQL Standard Syntax

```
PERCENT_RANK( ) OVER ({window_name | (window_specification)})
```

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

```
PERCENT_RANK( ) OVER ([partitioning] ordering)
```

SQL Server

SQL Server does not support *PERCENT_RANK*.

Example

The following Oracle-based example assigns relative ranks to the values of *NUM*, partitioning the data on the *ODD* column:

```
SELECT NUM, ODD, PERCENT_RANK() OVER
      (PARTITION BY ODD ORDER BY NUM) cumedist
  FROM test4;
```

NUM	ODD	CUMEDIST
0	0	0
2	0	.5
4	0	1
1	1	0
3	1	.3333333333
3	1	.3333333333
5	1	1

Following is an explanation of the calculation behind the rank for the row in which *NUM*=2:

1. Row *NUM*=2 is the second row in its partition; thus, it ranks #2.
2. Subtract 1 from 2 to get a divisor of 1.
3. The dividend is the total number of rows in the partition, or 3.
4. Subtract 1 from 3 to get a dividend of 2.
5. The result of 1/3 is .33 repeating, as shown in the example output.

RANK

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same, leading to gaps in rank numbers.

ANSI SQL Standard Syntax

```
RANK( ) OVER {window_name | (window_specification)}
```

Oracle and SQL Server

Oracle and SQL Server also require the *ordering* clause and do not allow the *framing* clause:

```
RANK( ) OVER ([partitioning] ordering)
```

Example

The following Oracle-based example uses the **NUM** column to rank the rows in the **test4** table:

```
SELECT NUM, RANK( ) OVER (ORDER BY NUM) rank
FROM test4;

```

NUM	RANK
0	1
1	2
2	3
3	4
3	4
4	6
5	7

Because both rows where **NUM**=3 rank the same (at #4), the next-higher row will be ranked at #6. The #5 rank is skipped.

ROW_NUMBER

Assigns a unique number to each row in a partition.

ANSI SQL Standard Syntax

```
ROW_NUMBER( ) OVER {window_name | (window_specification)}
```

Oracle and SQL Server

Oracle and SQL Server also require the *ordering* clause and do not allow the *framing* clause:

```
ROW_NUMBER( ) OVER ([partitioning] ordering)
```

Example

```
SELECT NUM, ODD, ROW_NUMBER( ) OVER
(PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;
```

NUM	ODD	CUMEDIST
0	0	1
2	0	2
4	0	3
1	1	1
3	1	2
3	1	3
5	1	4

ANSI SQL Scalar Functions

The ANSI SQL scalar functions return a single value each time they are invoked. The SQL standard provides many scalar functions that can be used to manipulate date and time types, strings, and numbers, as well as to retrieve system information such as the current user or login name. Scalar functions fall into the categories listed in Table 4-2.

Table 4-2. Categories of scalar functions

Function category	Explanation
Built-in	Performs operations on values or settings built into the database. Oracle uses the term “built-in” to describe all the specialty functions that it provides and that are thus “built into” the DBMS. This is a distinct and separate usage from the “built-in” scalar functions described here.
<i>CASE</i> and <i>CAST</i>	While these two functions operate on scalar input values, they are in a category all their own. <i>CASE</i> supplies <i>IF/THEN</i> logic to SQL statements, and <i>CAST</i> can convert values from one datatype to another.
Date and time	Performs operations on <i>temporal</i> datatypes and returns values in a temporal datatype format. There is no SQL2003 function that operates on a temporal datatype and returns a temporal result. The closest function is <i>EXTRACT</i> (covered in “Numeric Scalar Functions” later in this chapter), which operates on temporal values and returns numeric values. Functions returning temporal values but operating on no arguments are covered in the later section “Built-in Scalar Functions.”
Numeric	Performs operations on numeric values and returns numeric values.
String	Performs operations on character values (e.g., <i>CHAR</i> , <i>VARCHAR</i> , <i>NCHAR</i> , <i>NVARCHAR</i> , and <i>CLOB</i>) and returns string or numeric values.

Built-in Scalar Functions

ANSI SQL built-in scalar functions identify both the current user session and its characteristics, such as the current session privileges. Built-in scalar functions are always nondeterministic. The *CURRENT_DATE*, *CURRENT_TIME*, and *CURRENT_TIMESTAMP* functions listed in Table 4-3 are built-in functions that fall into the date-and-time category of functions. Although the four platforms discussed in this book provide many additional functions beyond these SQL built-ins, the SQL standard defines only those listed in Table 4-3.

Table 4-3. ANSI SQL built-in scalar functions

Function	Usage
<i>CURRENT_DATE</i>	Returns the current date
<i>CURRENT_TIME</i>	Returns the current time
<i>CURRENT_TIMESTAMP</i>	Returns the current date and time
<i>CURRENT_USER</i> or <i>USER</i>	Returns the currently active user within the database server
<i>SESSION_USER</i>	Returns the currently active authorization ID, if it differs from the user's ID
<i>SYSTEM_USER</i>	Returns the currently active user within the host operating system

MySQL

MySQL supports all the ANSI SQL built-in scalar functions. In addition, MySQL supports *NOW* as a synonym of the function *CURRENT_TIMESTAMP*.

Oracle

Oracle supports *USER*, *CURRENT_DATE*, and *CURRENT_TIMESTAMP*.

PostgreSQL and SQL Server

PostgreSQL and SQL Server support all the built-in scalar functions. In addition, PostgreSQL supports *NOW* as a synonym of the function *CURRENT_TIMESTAMP*.

Examples

The following queries retrieve the values from built-in functions. Notice that the various platforms return dates in their native formats:

```
/* On MySQL */
SELECT CURRENT_TIMESTAMP;
'2001-12-15 23:50:26'

/* On Microsoft SQL Server */
SELECT CURRENT_TIMESTAMP
GO
'Dec 15,2001 23:50:26'

/* On Oracle */
SELECT USER FROM dual;
dylan
```

CASE and CAST Functions

ANSI SQL provides a function named *CASE* that can be used to create *IF/THEN* flow-control logic within queries or *UPDATE* statements. The *CAST* function is for converting between datatypes and is also in the ANSI standard. All of the databases covered in this book provide ANSI standard support for both the *CASE* and *CAST* functions.

CASE

The *CASE* function provides *IF/THEN/ELSE* functionality within a *SELECT* or *UPDATE* statement. It evaluates a list of conditions and returns one value out of several possible values.

CASE has two usages: *simple* and *searched*. Simple *CASE* expressions compare one value, the *input_value*, with a list of other values and return a result associated with the first matching value. Searched *CASE* expressions allow the analysis of several logical conditions and return a result associated with the first one that is true.

All vendors provide the ANSI SQL syntax for *CASE*.

ANSI SQL Standard Syntax and Description

```
-- Simple comparison operation
CASE input_value
  WHEN when_condition THEN resulting_value
  [... n]
  [ELSE else_result_value]
END
```

```
-- Boolean searched operation
CASE
  WHEN Boolean_condition THEN resulting_value
  [... n]
  [ELSE else_result_expression]
END
```

In the simple *CASE* function, the *input_value* is evaluated against each *WHEN* clause. The *resulting_value* is returned for the first *TRUE* instance of *input_value = when_condition*. If no *when_condition* evaluates as *TRUE*, the *else_result_value* is returned. If no *else_result_value* is specified, *NULL* is returned.

The structure is essentially the same in the more elaborate Boolean searched operation, except that each *WHEN* clause has its own Boolean comparison operation.

In either usage, multiple *WHEN* clauses are used, though only one *ELSE* clause is necessary.

Examples

Here is a simple comparison operation where the *CASE* function alters the display of the **contract** column to make it more understandable:

```
SELECT au_fname,
       au_lname,
       CASE contract
         WHEN 1 THEN 'Yes'
         ELSE 'No'
       END 'contract'
  FROM authors
 WHERE state = 'CA'
```

Here is an elaborate searched *CASE* function in a *SELECT* statement that will report how many titles have been sold in different year-to-date sales ranges:

```
SELECT CASE
      WHEN ytd_sales IS NULL THEN 'Unknown'
      WHEN ytd_sales <= 200 THEN 'Not more than 200'
      WHEN ytd_sales <= 1000 THEN 'Between 201 and 1000'
      WHEN ytd_sales <= 5000 THEN 'Between 1001 and 5000'
      WHEN ytd_sales <= 10000 THEN 'Between 5001 and 10000'
      ELSE 'Over 10000'
    END 'YTD Sales',
    COUNT(*) 'Number of Titles'
  FROM titles
 GROUP BY CASE
      WHEN ytd_sales IS NULL THEN 'Unknown'
      WHEN ytd_sales <= 200 THEN 'Not more than 200'
      WHEN ytd_sales <= 1000 THEN 'Between 201 and 1000'
      WHEN ytd_sales <= 5000 THEN 'Between 1001 and 5000'
      WHEN ytd_sales <= 10000 THEN 'Between 5001 and 10000'
      ELSE 'Over 10000'
    END
  ORDER BY MIN( ytd_sales )
```

The results are:

YTD Sales	Number of Titles
Unknown	2
Not more than 200	1
Between 201 and 1000	2
Between 1001 and 5000	9
Between 5001 and 10000	1
Over 10000	3

Next is an *UPDATE* statement that applies discounts to all of the titles. This more complicated command will discount all personal computer-related titles by 25% and all other titles by 10%, with the exception of titles with year-to-date sales exceeding 10,000 units, which will receive only a 5% discount. This query uses a searched *CASE* expression to perform the price adjustments:

```
UPDATE  titles
SET      price = price *
CASE
    WHEN ytd_sales > 10000      THEN 0.95  -- 5% discount
    WHEN type = 'popular_comp' THEN 0.75  -- 25% discount
    ELSE 0.9                         -- 10% discount
END
WHERE  pub_date IS NOT NULL
```

This example demonstrates completion of three separate *UPDATE* operations in a single statement.

CAST

The *CAST* command explicitly converts an expression of one datatype to another. All vendors provide the ANSI SQL syntax for *CAST*.

ANSI SQL Standard Syntax and Description

CAST(expression AS data_type [(length)])

The *CAST* function converts any *expression*, such as a column value or variable, into another defined datatype. The length of the datatype may optionally be supplied, for those datatypes (such as *CHAR* or *VARCHAR*) that support lengths.



Be aware that some conversions, such as *DECIMAL* values to *INTEGER*, will result in rounding operations. Also, some conversion operations may result in an error if the new datatype does not have sufficient space to display the converted value.

Examples

This example retrieves the year-to-date sales figure as a *CHAR* and concatenates it with a literal string and a portion of the title of the book. It converts *ytd_sales* to *CHAR(5)*, and it shortens the length of the *title* to make the results more readable:

```
SELECT CAST(ytd_sales AS CHAR(5)) + ' Copies sold of ' +
CAST(title AS VARCHAR(30))
FROM titles
WHERE ytd_sales IS NOT NULL
    AND ytd_sales > 10000
ORDER BY ytd_sales DESC
```

The results are:

```
22246 Copies sold of The Gourmet Microwave
18722 Copies sold of You Can Combat Computer Stress
15096 Copies sold of Fifty Years in Buckingham Pala
```

Numeric Scalar Functions

The list of official ANSI SQL numeric functions is rather small, and the different platforms provide supplementary mathematical and statistical functions. MySQL and PostgreSQL both support many of the ANSI standard functions. The other database platforms offer the same capabilities of numeric scalar functions through their own internally defined functions, but they do not share the same names as those declared by the SQL standard. The supported numeric functions and syntax are listed in Table 4-4.

Table 4-4. ANSI SQL numeric functions

Function	Usage
<i>ABS</i>	Returns the absolute value of a number.
<i>BIT_LENGTH</i>	Returns an integer value representing the number of bits in another value.
<i>CEIL</i> or <i>CEILING</i>	Rounds a noninteger value upward to the next greatest integer. Returns an integer value unchanged.
<i>CHAR_LENGTH</i>	Returns an integer value representing the number of characters in a string expression.
<i>EXP</i>	Raises a value to the power of the mathematical constant known as <i>e</i> .
<i>EXTRACT</i>	Allows the datepart to be extracted (<i>YEAR</i> , <i>MONTH</i> , <i>DAY</i> , <i>HOUR</i> , <i>MINUTE</i> , <i>SECOND</i> , <i>TIMEZONE_HOUR</i> , or <i>TIMEZONE_MINUTE</i>) from a temporal expression.
<i>FLOOR</i>	Rounds a noninteger value downward to the next least integer. Returns an integer value unchanged.
<i>LN</i>	Returns the natural logarithm of a number.
<i>MOD</i>	Returns the remainder of one number divided into another.
<i>OCTET_LENGTH</i>	Returns an integer value representing the number of octets in another value. This value is the same as <i>BIT_LENGTH</i> / 8.
<i>POSITION</i>	Returns an integer value representing the starting position of a string within the search string.
<i>POWER</i>	Raises a number to a specified power.
<i>SQRT</i>	Computes the square root of a number.
<i>WIDTH_BUCKET</i>	Deposits a value into the appropriate bucket from a set of buckets covering a given range.

ABS

All platforms support the ANSI standard ABS function.

ANSI SQL Standard Syntax

`ABS(expression)`

ABS returns the absolute value of the number in *expression*.

Example

The following example shows how to use the ABS function:

```
/* SQL2003 */
SELECT ABS(-1) FROM NUMBERS
1
```

BIT_LENGTH, CHAR_LENGTH, and OCTET_LENGTH

All platforms stray from the ANSI standard in their support for scalar functions for determining the length of expressions. While the platform support is nonstandard, the equivalent functionality exists under different names.

ANSI SQL Standard Syntax

The ANSI SQL scalar functions for getting the length of a value take an *expression* to calculate the value and return the length as an integer. The *BIT_LENGTH* function returns the number of bits contained within the value of *expression*, *CHAR_LENGTH* returns the number of characters in the string *expression*, and *OCTET_LENGTH* returns the number of octets in the string *expression*. All three of these functions will return NULL if *expression* is NULL:

```
BIT_LENGTH( expression )
CHAR_LENGTH( expression )
OCTET_LENGTH( expression )
```

MySQL and PostgreSQL

MySQL and PostgreSQL both support *BIT_LENGTH*, *CHAR_LENGTH*, *OCTET_LENGTH*, and the ANSI SQL synonym *CHARACTER_LENGTH*.

Oracle

Oracle supports the *LENGTHB* function, which returns an integer value representing the number of bytes in an expression. For the length of an expression in characters, Oracle provides a *LENGTH* function as a synonym for *CHAR_LENGTH*.

SQL Server

SQL Server provides *LEN*, which performs all three functions.

Example

The following example, shown for different databases, determines the length of a string and a value retrieved from a column:

```
/* On MySQL and PostgreSQL */
SELECT CHAR_LENGTH('hello');
SELECT OCTET_LENGTH(book_title) FROM titles;

/* On Microsoft SQL Server */
SELECT DATALENGTH(title) FROM titles
WHERE type = 'popular_comp'
GO

/* On Oracle */
SELECT LENGTH('HORATIO') "Length of characters"
FROM dual;
```

CEIL

The *CEIL* function returns the smallest integer greater than an input value that you specify.

ANSI SQL Standard Syntax

ANSI SQL supports the following two forms of the function:

```
CEIL( expression )
CEILING( expression )
```

MySQL and PostgreSQL

MySQL and PostgreSQL support both *CEIL* and *CEILING*.

Oracle

Oracle supports only *CEIL*.

SQL Server

SQL Server supports only *CEILING*.

Examples

When you pass a positive, non-integer number, the effect of *CEIL* is to round up to the next highest integer:

```
SELECT CEIL(100.1) FROM dual;
CEIL(100.1)
-----
101
```

Remember, though, that with negative numbers, rounding “up” results in a lower absolute value:

```
SELECT CEIL(-100.1) FROM dual;
CEIL(-100.1)
-----
-100
```

Use *FLOOR* to get behavior opposite to that of *CEIL*.

EXP

The *EXP* function returns the value of the mathematical constant *e* (approximately 2.718281) raised to the power of a specified number.

ANSI SQL Standard Syntax

All platforms support the ANSI SQL standard syntax:

```
EXP( expression )
```

Example

The following example uses *EXP* to return an approximation of *e*:

EXTRACT > MySQL

```
SELECT EXP(1) FROM dual;  
EXP(1)  
-----  
2.71828183
```

Use *LN* to go in the opposite direction.

EXTRACT

The ANSI SQL scalar function for extracting parts from a date is *EXTRACT*.

ANSI SQL Standard Syntax

The ANSI SQL *EXTRACT* function takes a *date_part* and an *expression* that evaluates to a datetime value. MySQL, Oracle, and PostgreSQL support the ANSI SQL standard syntax:

```
EXTRACT( date_part FROM expression )
```

MySQL

MySQL's implementation extends somewhat beyond the ANSI standard. The ANSI standard does not have a provision for returning multiple fields from the same call to *EXTRACT* (e.g., *DAY_HOUR*). The MySQL extensions try to accomplish what the combination *DATE_TRUNC* and *DATE_PART* do in PostgreSQL. MySQL supports the *dateparts* listed in Table 4-5.

Table 4-5. MySQL *dateparts*

Type value	Meaning
<i>MICROSECOND</i>	Microseconds
<i>SECOND</i>	Seconds
<i>MINUTE</i>	Minutes
<i>HOUR</i>	Hours
<i>DAY</i>	Days
<i>WEEK</i>	Weeks
<i>MONTH</i>	Months
<i>QUARTER</i>	Quarter
<i>YEAR</i>	Years
<i>SECOND_MICROSECOND</i>	Seconds and microseconds
<i>MINUTE_MICROSECOND</i>	Minutes and microseconds
<i>MINUTE_SECOND</i>	Minutes and seconds
<i>HOUR_MICROSECOND</i>	Hours and microseconds
<i>HOUR_SECOND</i>	Hours, minutes, and seconds
<i>HOUR_MINUTE</i>	Hours and minutes
<i>DAY_MICROSECOND</i>	Days and microseconds
<i>DAY_SECOND</i>	Days, hours, minutes, and seconds
<i>DAY_MINUTE</i>	Days, hours, and minutes
<i>DAY_HOUR</i>	Days and hours
<i>YEAR_MONTH</i>	Years and months

Oracle

Oracle supports the ANSI SQL syntax with the dateparts listed in Table 4-6.

Table 4-6. Oracle dateparts

Type value	Meaning
DAY	The day of the month field (1–31)
HOUR	The hour field (0–23)
MINUTE	The minutes field (0–59)
MONTH	The month field (1–12)
SECOND	The seconds field (0–59)
TIMEZONE_HOUR	The hour component of the time zone offset
TIMEZONE_MINUTE	The minute component of the time zone offset
TIMEZONE_REGION	The current time zone name
TIMEZONE_ABBR	The abbreviation of the current time zone
YEAR	The year field

PostgreSQL

PostgreSQL supports the ANSI SQL syntax with a few extra dateparts. The dateparts it supports are listed in Table 4-7.

Table 4-7. PostgreSQL dateparts

Type value	Meaning
CENTURY	Before version 8.0, returned the year field divided by 100. After version 8.0, returns the correct numbering used by Gregorian calendar countries, where the first century starts in 0001 and there is no century numbered 0 (zero).
DAY	The day of the month field (1–31).
DECADE	The year field divided by 10.
DOW	The day of the week field (0–6, where Sunday is 0). This type only works for <i>TIMESTAMP</i> values.
DOY	The day of the year field (1–366). The maximum returned value is only 365 for years that are not leap years. This type can only be used on <i>TIMESTAMP</i> values.
EPOCH	The number of seconds between the epoch (1970-01-01 00:00:00-00) and the supplied value. The result can be negative for values before the epoch.
HOUR	The hour field (0–23).
MICROSECONDS	The seconds field (including fractional parts) multiplied by 1,000,000.
MILLENNIUM	Before version 8.0, returned the year field divided by 1000. After version 8.0, returns the correct numbering used by Gregorian calendar countries, where the first millennium starts in 0001 and there is no century numbered 0 (zero). The year 1001 is the start of the second millennium, while the year 2001 is the start of the third millennium.
MILLISECONDS	The seconds field (including fractional parts) multiplied by 1,000.
MINUTE	The minutes field (0–59).
MONTH	The month field (1–12).
QUARTER	The quarter of the year (1–4) in which the value occurs. This type can only be used with <i>TIMESTAMP</i> values.
SECOND	The seconds field (0–59).

Table 4-7. PostgreSQL dateparts (continued)

Type value	Meaning
<code>TIMEZONE</code>	The time zone offset in seconds.
<code>TIMEZONE_HOUR</code>	The hour component of the time zone offset.
<code>TIMEZONE_MINUTE</code>	The minute component of the time zone offset.
<code>WEEK</code>	The number of the week within the year in which the value falls.
<code>YEAR</code>	The year field.

SQL Server

SQL Server provides the function `DATEPART(date_part, expression)` as a synonym for the ANSI SQL function `EXTRACT(date_part FROM expression)`. SQL Server supports the dateparts listed in Table 4-8.

Table 4-8. SQL Server dateparts

Date part value	Meaning
<code>day</code>	The day of the month for the datetime expression. The abbreviations <code>d</code> and <code>dd</code> can also be used.
<code>dayofyear</code>	The day of the year for the datetime expression. The abbreviations <code>y</code> and <code>dy</code> can also be used.
<code>hour</code>	The hour of the day for the datetime expression. The abbreviation <code>hh</code> can also be used.
<code>ISO_WEEK</code>	The ISO 8601 week number (1–53). The abbreviations <code>isowk</code> and <code>isoww</code> can also be used.
<code>microsecond</code>	The microseconds (0–999999) for the datetime expression. The abbreviation <code>mcs</code> can also be used.
<code>millisecond</code>	The milliseconds for the datetime expression. The abbreviation <code>ms</code> can also be used.
<code>minute</code>	The minute of the hour for the datetime expression. The abbreviations <code>n</code> and <code>mi</code> can also be used.
<code>month</code>	The month (1–12). The abbreviations <code>m</code> and <code>mm</code> can also be used.
<code>nanosecond</code>	The number of nanoseconds (0–999999999). The abbreviation <code>ns</code> can also be used.
<code>quarter</code>	The quarter of the year in which the datetime expression falls. The abbreviations <code>q</code> and <code>qq</code> can also be used.
<code>second</code>	The second of the minute for the datetime expression. The abbreviations <code>s</code> and <code>ss</code> can also be used.
<code>TZoffset</code>	The time zone. The abbreviation <code>tz</code> can also be used.
<code>week</code>	The week of the year for the datetime expression. The abbreviations <code>wk</code> and <code>ww</code> can also be used.
<code>weekday</code>	The day of the week for the datetime expression. The abbreviation <code>dw</code> can also be used.
<code>year</code>	The year field of the datetime expression. The abbreviations <code>yy</code> and <code>yyyy</code> can also be used for two-digit and four-digit years, respectively.

Example

This example extracts dateparts from several datetime values:

```
/* On MySQL */
SELECT EXTRACT(YEAR FROM "2013-07-02");
2013
SELECT EXTRACT(YEAR_MONTH FROM "2013-07-02 01:02:03");
```

```

201307
SELECT EXTRACT(DAY_MINUTE FROM "2013-07-02 01:02:03");
20102

/* On PostgreSQL */
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
20

```

FLOOR

The *FLOOR* function returns the largest integer less than an input value that you specify.

ANSI SQL Standard Syntax

All platforms support the ANSI SQL syntax:

```
FLOOR( expression )
```

Examples

When you pass a positive number, the effect of *FLOOR* is to eliminate anything after the decimal point:

```

SELECT FLOOR(100.1) FROM dual;
FLOOR(100.1)
-----
100

```

Remember, though, that with negative numbers going in the “less-than” direction corresponds to increasingly larger absolute values:

```

SELECT FLOOR(-100.1) FROM dual;
FLOOR(-100.1)
-----
-101

```

Use *CEIL* to get behavior opposite to *FLOOR*.

LN

The *LN* function returns the natural logarithm of a number, which is the power to which you would need to raise the mathematical constant *e* (approximately 2.718281) in order to get the number in question as the result.

ANSI SQL Standard Syntax

```
LN( expression )
```

MySQL, Oracle, and PostgreSQL

MySQL, Oracle, and PostgreSQL support the ANSI SQL syntax for the *LN* function. MySQL and PostgreSQL also support the use of *LOG* as a synonym for *LN*.

SQL Server

SQL Server calls its natural logarithm function *LOG*:

```
LOG( expression )
```

Example

The following Oracle-based example shows the natural logarithm of a number closely approximating the mathematical constant known as e :

```
SELECT LN(2.718281) FROM dual;
LN(2.718281)
-----
.999999695
```

Use the *EXP* function to go in the other direction.

MOD

The *MOD* function returns the remainder of a dividend divided by a divider. All platforms support the ANSI SQL syntax for the *MOD* function.

ANSI SQL Standard Syntax

```
MOD( dividend, divider )
```

The standard syntax for the *MOD* function is to return the remainder of the dividend divided by the divider; it returns the dividend if the divider is 0.

Example

The following example shows how to use the *MOD* function from within a *SELECT* statement:

```
SELECT MOD(12, 5) FROM NUMBERS
2
```

POSITION

The *POSITION* function returns an integer that indicates the starting position of a string within the search string.

ANSI SQL Standard Syntax

```
POSITION( string1 IN string2 )
```

The standard syntax for the *POSITION* function is to return the first location of *string1* within *string2*. *POSITION* returns 0 if *string1* does not occur within *string2* and NULL if either argument is NULL.

MySQL and PostgreSQL

MySQL and PostgreSQL support the ANSI SQL syntax for the *POSITION* function.

Oracle

Oracle's equivalent function is called *INSTR*.

SQL Server

Instead of *POSITION*, SQL Server supports *CHARINDEX* and *PATINDEX* functions. *CHARINDEX* and *PATINDEX* are very similar, except that *PATINDEX* allows the use of wildcard characters in the search criteria.

Examples

```
/* On MySQL */
SELECT LOCATE('bar', 'foobar');
4

/* On MySQL and PostgreSQL */
SELECT POSITION('fu' IN 'snafhu');
0

/* On Microsoft SQL Server */
SELECT CHARINDEX( 'de', 'abcdefg' )
GO
4
SELECT PATINDEX( '%fg', 'abcdefg' )
GO
6
```

POWER

Use *POWER* to raise a number to a specific value.

ANSI SQL Standard Syntax

All platforms support the ANSI SQL syntax:

```
POWER( base, exponent )
```

The result of the *POWER* function is *base* raised to the *exponent* power, or *base^{exponent}*. If *base* is negative, *exponent* must be an integer.

Examples

Raising a positive number to an exponent is straightforward:

```
SELECT POWER(10,3) FROM dual;
POWER(10,3)
-----
1000
```

Anything raised to the 0th power evaluates to 1:

```
SELECT POWER(0,0) FROM dual;
POWER(0,0)
-----
1
```

Negative exponents move the decimal point to the left:

```
SELECT POWER(10,-3) FROM dual;
POWER(10,-3)
-----
.001
```

SQRT

The *SQRT* function returns the square root of a number.

ANSI SQL Standard Syntax

All platforms support the ANSI SQL syntax:

```
SQRT( expression )
```

Example

```
SELECT SQRT(100) FROM dual;
```

```
SQRT(100)
```

```
-----
```

```
10
```

WIDTH_BUCKET

The `WIDTH_BUCKET` function assigns values to buckets (individual segments) in an equiwidth histogram.

ANSI SQL Standard Syntax

In the following syntax, *expression* represents a value to be assigned to a bucket. You would typically base *expression* on one or more columns returned by a query:

```
WIDTH_BUCKET( expression, min, max, buckets )
```

The *buckets* argument specifies the number of buckets to create over the range defined by *min* through *max*. *min* is inclusive, whereas *max* is not. The value from *expression* is assigned to one of those buckets, and the function then returns the corresponding bucket number. When *expression* falls outside the range of buckets, the function returns either 0 or *max* + 1, depending on whether *expression* is lower than *min* or greater than or equal to *max*.

MySQL and SQL Server

MySQL and SQL Server do not support `WIDTH_BUCKET`.

Oracle and PostgreSQL

Oracle and PostgreSQL support the ANSI SQL syntax for `WIDTH_BUCKET`.

Examples

The following example divides the integer values 1 through 10 into two buckets:

```
SELECT x, WIDTH_BUCKET(x,1,10,2)
  FROM pivot;
-----
```

1	1
2	1
3	1
4	1
5	1
6	2
7	2
8	2
9	2
10	3

This next example is more interesting. It divides 11 values (from 1 through 10) into three buckets and illustrates the distinction between *min* being inclusive and *max* being noninclusive:

```
SELECT x, WIDTH_BUCKET(x,1,10,3)
  FROM pivot;
      X  WIDTH_BUCKET(X,1,10,3)
-----
 1      1
 2      1
 3      1
 4      2
 5      2
 6      2
 7      3
 8      3
 9      3
 9.9    3
 10     4
```

Pay particular attention to the results for $X=1$, $X=9.9$, and $X=10$. An input value of *min* (1, in this example) falls into the first bucket, proving that the lower end of the range for bucket #1 is defined as $x \geq \text{min}$. An input value of *max*, however, falls *outside* the highest bucket. In this example, 10 falls into the overflow bucket numbered $\text{max} + 1$. The value 9.9, on the other hand, falls into bucket #3, illustrating that the upper end of the range for the highest bucket is defined as $x < \text{max}$.

String Functions and Operators

Basic string functions and operators offer a number of capabilities and return string values as their results. Some string functions are *dyadic*, indicating that they operate on two strings at once. ANSI SQL supports the string functions listed in Table 4-9.

Table 4-9. SQL string functions and operators

Function or operator	Usage
Concatenation operator	Appends two or more literal string expressions, column values, or variables together into one string
<i>CONVERT</i>	Converts a string to a different representation within the same character set
<i>LOWER</i>	Converts a string to all lowercase characters
<i>OVERLAY</i>	Returns the result of replacing a substring of one string with another
<i>SUBSTRING</i>	Returns a portion of a string
<i>TRANSLATE</i>	Converts a string from one character set to another
<i>TRIM</i>	Removes leading characters, trailing characters, or both from a character string
<i>UPPER</i>	Converts a string to all uppercase characters

Concatenation Operator

ANSI SQL defines a concatenation operator (`||`), which joins two distinct strings into one string value.

MySQL

MySQL supports `CONCAT` as a synonym for the ANSI SQL concatenation operator and uses the `||` operator for logical OR.

Oracle and PostgreSQL

Oracle and PostgreSQL support the ANSI SQL double vertical bar (`||`) concatenation operator. Oracle also supports `CONCAT` as a synonym for the ANSI SQL operator.

SQL Server

SQL Server uses the plus sign (+) as a synonym for the ANSI SQL concatenation operator. SQL Server has the system setting `CONCAT_NULL_YIELDS_NULL`, which can be set to alter the behavior when NULL values are used in the concatenation of string values.

Examples

```
/* ANSI SQL Syntax */
'string1' || 'string2' || 'string3'
'string1string2string3'

/* On MySQL */
CONCAT('string1', 'string2')
'string1string2'
```

If any of the concatenation values are NULL, the entire returned string is NULL. Also, if a numeric value is concatenated, it is implicitly converted to a character string.

```
SELECT CONCAT('My ', 'bologna ', 'has ', 'a ', 'first ', 'name...');
'My bologna has a first name...'
SELECT CONCAT('My ', NULL, 'has ', 'first ', 'name...');
NULL
```

CONVERT and TRANSLATE

The `CONVERT` function alters the representation of a character string within its character set and collation. For example, `CONVERT` might be used to alter the number of bits per character.

`TRANSLATE` alters the character set of a string value from one base character set to another. Thus, `TRANSLATE` might be used to translate a value from the English character set to a Kanji (Japanese) or Cyrillic (Russian) character set. The translation must already exist, either by default or by virtue of having been created using the `CREATE TRANSLATION` command.

ANSI SQL Standard Syntax

```
CONVERT(char_value USING conversion_char_name)
```

```
TRANSLATE(char_value USING translation_name)
```

`CONVERT` converts *char_value* to the character set with the name supplied in *conversion_char_name*. `TRANSLATE` converts *char_value* to the character set provided in *translation_name*.

MySQL

MySQL supports the ANSI SQL syntax for *CONVERT* but does not support *TRANSLATE*.

Oracle

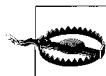
Oracle supports *CONVERT* and *TRANSLATE* with the same meaning as ANSI SQL. The Oracle syntax follows:

```
CONVERT(char_value, target_char_set, source_char_set)
```

```
TRANSLATE(char_value USING {CHAR_CS | NCHAR_CS})
```

Under Oracle's implementation, the *CONVERT* function returns the text of *char_value* in the target character set. *char_value* is the string to convert, *target_char_set* is the name of the character set into which the string is to be converted, and *source_char_set* is the name of the character set in which *char_value* was originally stored.

Oracle's *TRANSLATE* function follows the ANSI syntax, but it supports only two arguments for the character set: you can choose between the database character set (CHAR_CS) and the national character set (NCHAR_CS).



Oracle also supports a different function named *TRANSLATE*, which omits the *USING* keyword. That version of *TRANSLATE* has nothing to do with character set translation.

Both the target and source character set names can be passed either as literal strings, in variables, or in columns from a table. Note that replacement characters might be substituted when converting from or to a character set that does not support a representation of all the characters used in the conversion.

Oracle supports several common character sets, including *US7ASCII*, *WE8DECDEC*, *WE8HP*, *F7DEC*, *WE8EBCDIC500*, *WE8PC850*, and *WE8ISO8859P1*. For example:

```
SELECT CONVERT('Gro2', 'US7ASCII', 'WE8HP')FROM DUAL;
Gross
```

PostgreSQL

PostgreSQL supports the ANSI standard *CONVERT*, and conversions can be defined by using *CREATE CONVERSION*. PostgreSQL's implementation of the *TRANSLATE* function offers a large superset of functions that can convert any occurrence of one text string to another within a specified string:

```
TRANSLATE(character_string, from_text, to_text)
```

Here are some examples:

```
SELECT TRANSLATE('12345abcde', '5a', 'XX');
'1234XXbcde'
```

```
SELECT TRANSLATE(title, 'Computer', 'PC')
FROM titles
WHERE type = 'Personal_computer'
```

```
SELECT CONVERT('PostgreSQL' USING iso_8859_1_to_utf_8)
'PostgreSQL'
```

SQL Server

SQL Server does not support `TRANSLATE`. SQL Server's implementation of `CONVERT` is a very rich utility that alters the base datatype of an expression but is otherwise dissimilar to the ANSI SQL `CONVERT` function. It is functionally equivalent to the `CAST` function:

```
CONVERT (data_type[(length) | (precision, scale)], expression[, style])
```

The `style` clause is used to define the format of a date conversion. Refer to the SQL Server documentation for more information. Following is an example:

```
SELECT title, CONVERT(char(7), ytd_sales)
FROM titles
ORDER BY title
GO
```

LOWER and UPPER

The functions `LOWER` and `UPPER` allow the case of a string to be altered quickly and easily, so that all the characters are lower- or uppercase, respectively. These functions are supported in all the database implementations covered in this book. The different database platforms also support a variety of other text formatting functions that are specific to their implementations.

ANSI SQL Standard Syntax

```
LOWER(string)
UPPER(string)
```

`LOWER` converts `string` into a lowercase string. `UPPER` is the uppercase counterpart of `LOWER`.

MySQL

MySQL supports the ANSI SQL `UPPER` and `LOWER` scalar functions, as well as the synonyms `UCASE` and `LCASE`.

Oracle, PostgreSQL, and SQL Server

These platforms all support the ANSI SQL `UPPER` and `LOWER` scalar functions.

Example

```
SELECT LOWER('You Talkin To ME?'), UPPER('you talking to me?!');
you talkin to me?, YOU TALKING TO ME?!
```

OVERLAY

The `OVERLAY` function embeds one string into another and returns the result.

ANSI SQL Standard Syntax

```
OVERLAY(string PLACING embedded_string FROM start
[FOR length])
```

If any of the inputs are `NULL`, the `OVERLAY` function returns `NULL`. The `embedded_string` replaces the `length` characters in `string`, starting at the character position `start`.

If the *length* is not specified, the *embedded_string* will replace all the characters in *string* after *start*.

MySQL, Oracle, and SQL Server

These platforms do not support the *OVERLAY* function. You can simulate the *OVERLAY* function on these platforms by using a combination of *SUBSTRING* and the concatenation operator.

PostgreSQL

PostgreSQL supports the ANSI standard for *OVERLAY*.

Examples

This is an example of how to use the *OVERLAY* function:

```
/* ANSI SQL and PostgreSQL */
SELECT OVERLAY('DONALD DUCK' PLACING 'TRUMP' FROM 8) FROM NAMES;
'DONALD TRUMP'
```

SUBSTRING

The *SUBSTRING* function allows one character string to be returned from another.

ANSI SQL Standard Syntax

```
SUBSTRING(extraction_string FROM starting_position [FOR length]
[COLLATE collation_name])
```

If any of the inputs are *NULL*, the *SUBSTRING* function returns *NULL*. The *extraction_string* is the source from which the character value is to be extracted. It may be a literal string, a column in a table with a character datatype, or a variable with a character datatype. The *starting_position* is an integer value telling the function at which position to begin performing the extraction. The optional *length* is an integer value that tells the function how many characters to extract, starting at the *starting_position*. If the optional *FOR* keyword is omitted, the substring starting at *starting_position* and continuing to the end of the *extraction_string* is returned.

MySQL

MySQL largely supports the ANSI standard, but it does not accept the *COLLATE* clause. MySQL's implementation assumes that the characters are to be extracted from the starting position and will continue to the end of the character string. The syntax is as follows:

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

Oracle

Oracle's implementation, *SUBSTR*, largely functions the same way as ANSI SQL's *SUBSTRING*, but Oracle does not support the *COLLATE* clause. When the *starting_position* is a negative number, Oracle counts from the end of the *extraction_string*. If *length* is omitted, the remainder of the string (starting at *starting_position*) is returned. The syntax is:

```
SUBSTR(extraction_string, starting_position[, length])
```

PostgreSQL

PostgreSQL largely supports the ANSI standard, but it does not accept the `COLLATE` clause. The PostgreSQL syntax is:

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

SQL Server

SQL Server's implementation is similar to the ANSI standard, except that it does not support the `COLLATE` clause. SQL Server allows this command to be applied to text, image, and binary datatypes; however, the *starting_position* and *length* represent the number of bytes rather than the number of characters to count. The SQL Server syntax follows:

```
SUBSTRING(extraction_string [FROM starting_position] [FOR length])
```

Examples

These examples generally work on any one of the five database platforms profiled in this book. Only the second Oracle example, with a negative starting position, fails on the others (assuming, of course, that Oracle's `SUBSTR` is translated into `SUBSTRING`):

```
/* On Oracle, counting from the left */
SELECT SUBSTR('ABCDEFG',3,4) FROM DUAL;
'CDEF'

/* On Oracle, counting from the right */
SELECT SUBSTR('ABCDEFG',-5,4) FROM DUAL;
'CDEF'

/* On MySQL */
SELECT SUBSTRING('Be vewy, vewy quiet' FROM 5);
'wy, vewy quiet'

/* On PostgreSQL or SQL Server */
SELECT au_lname, SUBSTRING(au_fname, 1, 1)
FROM authors
WHERE au_lname = 'Carson'
Carson      C
```

TRIM

The `TRIM` function removes leading characters, trailing characters, or both from a specified character string or *BLOB* value. This function also removes other types of characters from a specified character string. The default behavior is to trim the specified character from both sides of the character string. If no removal character is specified, `TRIM` removes spaces by default.

ANSI SQL Standard Syntax

```
TRIM( [ [{LEADING | TRAILING | BOTH}] [removal_char] FROM ]
      target_string
      [COLLATE collation_name] )
```

The *removal_char* is the character to be stripped out, and the *target_string* is the character string from which characters are to be stripped. If no *removal_char* is specified, *TRIM* strips out spaces. The *COLLATE* clause forces the result set of the function into another pre-existing collation set.

MySQL, Oracle, and PostgreSQL

These platforms support the ANSI SQL syntax of *TRIM*.

SQL Server

SQL Server provides the functions *LTRIM* and *RTRIM* to trim off leading spaces or trailing spaces, respectively. On SQL Server, *LTRIM* and *RTRIM* cannot be used to trim other types of characters.

Examples

```
SELECT TRIM('  wamalamadingdong  ');
'wamalamadingdong'

SELECT LTRIM( RTRIM('  wamalamadingdong  ') );
'wamalamadingdong'

SELECT TRIM(LEADING '19' FROM '1976 AMC GREMLIN');
'76 AMC GREMLIN'

SELECT TRIM(BOTH 'x' FROM 'xxxWHISKEYxxx');
'WHISKEY'

SELECT TRIM(TRAILING 'snack' FROM 'scooby snack');
'scooby '
```

Platform-Specific Extensions

The following sections provide a full listing and description of each vendor-supported function. The functions are platform-specific. Thus, a MySQL function, for example, is not guaranteed to be supported by any other vendor.

MySQL-Supported Functions

This section provides an alphabetical listing of MySQL-supported functions, with examples and corresponding results.

ACOS(number)

Returns the arccosine of *number*, ranging from -1 to 1 . The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

ADDDATE(date, days)

Returns *date* with *days* added to it. For example:

```
SELECT ADDDATE('2008-05-14', 1) -> '2008-05-15'
```

ADDDATE(*date*, *INTERVAL expr unit*)

Returns *date* with *expr* units of *unit* added. For example:

```
SELECT ADDDATE('2008-05-14', INTERVAL 1 DAY) -> '2008-05-15'
```

AES_DECRYPT(*crypt_str*, *key_str*)

Returns *crypt_str* decrypted using AES and the key *key_str*.

AES_ENCRYPT(*str*, *key_str*)

Returns *str* encrypted with AES using the key *key_str*. For example:

```
SELECT AES_DECRYPT(AES_ENCRYPT('secret sauce', 'password'), 'password')  
-> 'secret sauce'
```

ASCII(*text*)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(*number*)

Returns the arcsine of *number*, ranging from -1 to 1 . The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(*number*)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATAN2(*x*, *y*)

Returns the arctangent of the two variables *x* and *y*. *ATAN2(x, y)* is similar to *ATAN(y/x)*, with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

```
ATAN2(3.1415, 1) -> 1.262619
```

BENCHMARK(*count*, *expr*)

Executes the expression *expr* *count* times. The result value is always 0 . For example:

```
BENCHMARK(1000000,ATAN2(3.1415, 1)) -> 0
```

BIN(*number*)

Returns a string containing the binary value of *number*, where *number* is a *BIGINT*.

BINARY(*string*)

Casts *string* to a binary string.

BIT_AND(*expr*)

Aggregate function that returns the bitwise AND of all bits in *expr*. The calculation is performed with 64-bit (*BIGINT*) precision. The value -1 is returned when no matching rows are found. For example:

```
BIT_AND(mycolumn) -> 0
```

BIT_COUNT(*number*)

Returns the number of bits that are set in *number*. For example:

```
BIT_COUNT(5) -> 2
```

BIT_OR(expr)

Aggregate function that returns the bitwise OR of all bits in *expr*. The calculation is performed with 64-bit (BIGINT) precision. The value 0 is returned when no matching rows are found. For example:

```
BIT_OR(mycolumn) -> 1
```

BIT_XOR(expr)

Aggregate function that returns the bitwise XOR of all bits in *expr*. The calculation is performed with 64-bit (BIGINT) precision. The value 0 is returned when no matching rows are found. For example:

```
BIT_XOR(mycolumn) -> 1
```

CHAR(number[, ...])

Returns a string consisting of the characters given by the ASCII code values in the arguments. Any NULL values are ignored. For example:

```
CHAR(120,121,122) -> 'xyz'
```

CHARSET(str)

Returns the character set of the string argument *str*. For example:

```
CHARSET('oolong') -> 'latin1'
```

COALESCE(list)

Returns the first non-NULL element in the list. For example:

```
COALESCE( NULL, 1, 2 ) -> 1
```

COERCIBILITY(expr)

Returns the collation coercibility value of *expr*, which is an integer value between 0 and 5; the lowest values have the highest precedence. For example:

```
COERCIBILITY( 'darjeeling' ) -> 4
```

COLLATION(str)

Returns the collation of *str*. For example:

```
COLLATION( _utf8'assam' ) -> 'utf8_general_ci'
```

COMPRESS(string)

Returns a compressed version of *string*.

CONCAT_WS(separator, str1, str2[, ...])

A special form of *CONCAT* that inserts *separator* between every pair of string arguments concatenated. If *separator* is NULL, the result is NULL. For example:

```
CONCAT_WS( ' ', au_lname, au_fname ) -> 'Jefferson, Thomas'
```

CONNECTION_ID()

Returns the connection ID for the connection. Every connection has its own unique ID. For example:

```
CONNECTION_ID() -> 305102
```

CONV(number, from_base, to_base)

Returns a string representation of the number *number*, converted from base *from_base* to base *to_base*. If any argument is NULL, the result is NULL. For example:

```
CONV(12,10,2) -> 1100
```

COS(*number*)

Returns the cosine of *number* as an angle expressed in radians. For example:

```
SELECT COS(0) -> 1.000000
```

COT(*number*)

Returns the cotangent of *number* as an angle expressed in radians. For example:

```
SELECT COT(0) -> NULL
```

CRC32(*expr*)

Returns the CRC32 checksum of *expr*. For example:

```
SELECT CRC32('mysql') -> 2501908538
```

CURDATE()

Returns today's date as a value in YYYY-MM-DD or YYYYMMDD format, depending on whether the function is used in a string or numeric context. For example:

```
CURDATE() -> '2003-06-24'
```

CURTIME()

Returns the current time as a value in HH:MM:SS or HHMMSS format, depending on whether the function is used in a string or numeric context. For example:

```
CURTIME() -> '20:40:20'
```

DATABASE()

Returns the current database name. For example:

```
DATABASE() -> 'PUBS'
```

DATE_ADD(*date*, INTERVAL *expr* *type*)

DATE_SUB(*date*, INTERVAL *expr* *type*)

ADDDATE(*date*, INTERVAL *expr* *type*)

SUBDATE(*date*, INTERVAL *expr* *type*)

These functions perform date arithmetic calculations. *ADDDATE* and *SUBDATE* are synonyms for *DATE_ADD* and *DATE_SUB*. *DATE_ADD* returns the result of adding the *INTERVAL* to the *date* expression. *DATE_SUB* returns the result of subtracting the *INTERVAL* from the *date* expression. For example:

```
DATE_ADD('1999-04-15', INTERVAL 1 DAY) -> '1999-04-16'
```

```
DATE_SUB('1999-04-15', INTERVAL 1 DAY) -> '1999-04-14'
```

DATE_FORMAT(*date*, *format*)

Formats the *date* value according to the *format* string. For example:

```
DATE_FORMAT('2008-04-15', '%M-%D-%Y') -> 'April-15th-2008'
```

Table 4-10 lists the available specifiers for *format* and their meanings.

Table 4-10. MySQL format specifiers

Format specifier	Meaning
<code>%a</code>	Abbreviation of the day (Sun–Sat)
<code>%b</code>	Abbreviation of the month (Jan–Dec)
<code>%c</code>	Month number (1–12)
<code>%D</code>	Day of month with a suffix (1 st , 2 nd , 3 rd , ...)
<code>%d</code>	Two-digit day of month (01, 02, ...)
<code>%e</code>	Day of month (1, 2, 3, ...)
<code>%H</code>	Hour (00–23)
<code>%h</code>	Hour (01–12)
<code>%l</code>	Hour (01–12)
<code>%i</code>	Minute (00–59)
<code>%j</code>	Day of year (001–366)
<code>%k</code>	Hour (0–23)
<code>%l</code>	Hour (1–12)
<code>%M</code>	Full month name (January–December)
<code>%m</code>	Month (01–12)
<code>%p</code>	AM or PM
<code>%r</code>	Twelve-hour time (hh:mm:ss AM or PM)
<code>%S</code> or <code>%s</code>	Seconds (00–59)
<code>%T</code>	Twenty-four-hour time (hh:mm:ss)
<code>%U</code>	Week number (00–53, Sunday being the first day of the week)
<code>%u</code>	Week number (00–53, Monday being the first day of the week)
<code>%V</code>	Week number (01–53, Sunday being the first day of the week)
<code>%v</code>	Week number (01–53, Monday being the first day of the week)
<code>%W</code>	Name of the day (Sunday–Saturday)
<code>%w</code>	Day of the week (0–6, 0 being Sunday and 6 being Saturday)
<code>%X</code>	Four-digit year with Sunday being the first day of the week
<code>%x</code>	Four-digit year with Monday being the first day of the week
<code>%Y</code>	Four-digit year
<code>%y</code>	Two-digit year
<code>%%</code>	Literal “%”

DAYNAME(*date*)

Returns the name of the weekday for *date*. For example:

```
DAYNAME('1999-04-15') -> 'Thursday'
```

DAYOFMONTH(*date*)

Returns the day of the month for *date*, in the range 1 to 31. For example:

```
DAYOFMONTH('1999-04-15') -> 15
```

DAYOFWEEK(*date*)

Returns the weekday index for *date* (1 = Sunday, 2 = Monday, ... 7 = Saturday). For example:

DAYOFWEEK('1999-04-15') -> 5

DAYOFYEAR(*date*)

Returns the day of the year for *date*, in the range 1 to 366. For example:

DAYOFYEAR('1999-04-15') -> 105

DECODE(*crypt_str*, *pass_str*)

Decrypts the encrypted string *crypt_str* using *pass_str* as the password; *crypt_str* should be a string returned from *ENCODE*. For example:

DECODE(ENCODE('foo', 'bar'), 'bar') -> 'foo'

DEFAULT(*column_name*)

Returns the default value of the column *column_name*. For example:

SELECT DEFAULT(a_column) FROM a_table -> 0

DEGREES(*number*)

Returns the argument *number* converted from radians to degrees. For example:

DEGREES(3.1415926) -> 179.99999692953

DES_DECRYPT(*crypt_str*, *key_str*)

Returns *crypt_str* decrypted using DES and the key *key_str*.

DES_ENCRYPT(*str*, *key_str*)

Returns *str* encrypted with DES using the key *key_str*. For example:

**SELECT DES_DECRYPT(DES_ENCRYPT('secret sauce', 'password'), 'password')
-> 'secret sauce'**

ELT(*n*, *str1*, *str2*, *str3*[, ... *n*])

Returns *str1* if *n* = 1, *str2* if *n* = 2, and so on. If *n* is less than 1 or greater than the number of arguments, this function returns NULL. *ELT* is the complement of *FIELD*. For example:

ELT(1, 'Hi', 'There') -> 'Hi' ELT(2, 'Hi', 'There') -> 'There'

ENCODE(*str*, *pass_str*)

Encrypts *str* using *pass_str* as the password. To decrypt the result, use *DECODE*. The result is a binary string the same length as the string. For example:

DECODE(ENCODE('foo', 'bar'), 'bar') -> 'foo'

ENCRYPT(*str*[, *salt*])

Encrypts *str* using the Unix *crypt* system call. The *salt* argument should be a string with two characters. For example:

ENCRYPT('password') -> 'ZB7yqPUHvNnmo'

EXPORT_SET(*bits*, *on*, *off*, [*separator*, [*number_of_bits*]])

Returns a string where every bit in *bits* that is set gets an *on* string and every unset bit gets an *off* string. Each string is separated with *separator*; the default is a comma (,). *number_of_bits* is optional; when omitted, the default is 64. For example:

EXTRACTVALUE(xml, xpath)

Extracts text from the XML fragment given by `xml` corresponding to the matching XPath in `xpath`. For example:

```
EXPORT_SET('<catalog><tea>oolong</tea><tea>darjeeling</tea></catalog>','/catalog/tea')
'oolong darjeeling'
```

FIELD(str, str1, str2, str3[, ...])

Returns the index of *str* in the given string arguments, or 0 if *str* is not found. *FIELD* is the complement of *ELT*. For example:

```
FIELD('GOOSE','DUCK','DUCK','GOOSE','DUCK') -> 3
```

FIND_IN_SET(str, strlist)

Returns the index of *str* within the *strlist*, where *strlist* is a list of strings separated by commas. This function is equivalent to calling *FIELD(str, CONCAT_WS(',', str1, str2, str3[, ...]))*. For example:

`FIND_IN_SET('b','a,b,c,d') -> 2`

FORMAT(number, decimals)

Formats the number *number* to a format like #,###,###.##, rounded to *decimals* decimals. If *decimals* is 0, the result has no decimal point or fractional part. For example:

FORMAT(12345.2132,2) -> 12,345.21
FORMAT(12345.2132,0) -> 12,345

FOUND_ROWS()

Returns the number of rows that would have been returned by a query that was previously executed with the *LIMIT* clause. The query of *FOUND_ROWS* must be done immediately after the limited query is executed. For example:

```
SELECT FOUND_ROWS() -> 31415926
```

FROM_DAYS(*number*)

Given a day *number*, returns a *DATE* value. This function should not be used for values that precede the advent of the Gregorian calendar (1582), due to the days lost when the calendar was changed. For example:

FROM_DAYS(88888) -> 2433-09-10

FROM_UNIXTIME(*unix_timestamp*)

Returns a representation of the `unix_timestamp` argument as a value in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. For example:

FROM_UNIXTIME(888123892) -> 1998-02-21 21:04:52

FROM_UNIXTIME(*unix_timestamp*, *format*)

Returns a string representation of the *unix_timestamp*, formatted according to the *format* string, *format* may contain the same specifiers as those listed in the entry for the *DATE_FORMAT* function. For example:

FROM_UNIXTIME(888123892, '%Y %D %M') -> '1998 21st February'

`GET_LOCK(str, timeout)`

Tries to obtain a lock with a name given by the string `str`, with a time-out of `timeout` seconds. Returns 1 if the lock is obtained successfully, or NULL if an error occurs or the attempt to acquire the lock times out. For example:

```
GET_LOCK('lochness',10) -> 1
```

`GREATEST(x, y[, ...])`

Returns the largest argument. For example:

```
GREATEST(8,2,4) -> 8
```

`GROUP_CONCAT([DISTINCT] expr [ORDER BY order [ASC | DESC]] [SEPARATOR sep])`

Returns a concatenation of non-NULL values from a grouping where `expr` is the expression to use in the concatenation, `order` is the expression to use in the ordering, and `sep` is the string to insert between concatenated values. For example:

```
SELECT estate, GROUP_CONCAT(tea SEPARATOR ',') FROM catalog GROUP BY estate
```

`HEX(number)`

Returns a string representation of the hexadecimal value of `number`. This is equivalent to `CONV(number, 10, 16)`. For example:

```
HEX(255) -> FF
```

`HOUR(time)`

Returns the hour for `time`, in the range 0 to 23. For example:

```
HOUR('08:20:15') -> 8
```

`IF(expr1, expr2, expr3)`

Returns `expr2` if `expr1` is TRUE; otherwise, returns `expr3`. For example:

```
IF(1,'yes','no') -> 'yes'  
IF(0,'yes','no') -> 'no'
```

`IFNULL(expr1, expr2)`

Returns `expr1` if `expr1` is not NULL; otherwise, returns `expr2`. For example:

```
IFNULL(0,'NULL') -> 0  
IFNULL(NULL,'NULL') -> 'NULL'
```

`INET_ATON(expr)`

Returns a numeric representation of a network IP address found in `expr`. For example:

```
INET_ATON('127.0.0.1') -> 2130706433
```

`INET_NTOA(num)`

Returns the network IP address as a string decoded from the numeric value `num`. For example:

```
INET_NTOA(2130706433) -> '127.0.0.1'
```

`INSERT(str, pos, len, newstr)`

Returns the string `str` with `newstr` inserted at character position `pos` for length `len`. For example:

```
INSERT('paper',2,3,'ea') -> 'pear'
```

INSTR(*str*, *substr*)

Returns the position of the first occurrence of the substring *substr* in the string *str*. For example:

```
INSTR('ducks', 'c') -> 3
```

INTERVAL(*num1*, *num2*, *num3*, *num4*[, ... *n*])

Returns 0 if *num1* < *num2*, 1 if *num1* < *num3*, and so on. It is required that *num2* < *num3* < *num4* < ... < *numN*. For example:

```
INTERVAL(5,1,6) -> 1
```

```
INTERVAL(5,2,3,7,9) -> 2
```

IS_FREE_LOCK(*lock*)

Returns 1 if *lock* is free and 0 if the lock is currently in use. The function may return NULL on error conditions. For example:

```
IS_FREE_LOCK('lochness') -> 0
```

IS_USED_LOCK(*lock*)

Returns the connection identifier if the lock with the ID *lock* is taken, and NULL otherwise. For example:

```
IS_USED_LOCK('lochness') -> 0
```

ISNULL(*expr*)

If *expr* is NULL, *IFNULL* returns 1; otherwise, it returns 0. For example:

```
ISNULL(1) -> 0
```

```
ISNULL(NULL) -> 1
```

LAST_DAY(*expr*)

Returns the last day in the month for the date found in *expr*. For example:

```
LAST_DAY('2012-01-01') -> '2012-01-31'
```

LAST_INSERT_ID([*expr*])

Returns the last automatically generated value that was inserted into an *AUTO_INCREMENT* column. For example:

```
LAST_INSERT_ID() -> 0
```

LCASE(*str*)

Synonym for *lower(str)*. For example:

```
LCASE('DUCK') -> 'duck'
```

LEAST(*X*, *Y*[, ... *n*])

With two or more arguments, returns the smallest (minimum-valued) argument. For example:

```
LEAST(10,5,3,7) -> 3
```

LEFT(*str*, *len*)

Returns the leftmost *len* characters from the string *str*. For example:

```
LEFT('Ducks', 4) -> 'Duck'
```

LENGTH(*str*)

Returns the length of the string *str*. For example:

```
LENGTH('DUCK') -> 4
```

`LOAD_FILE(file_name)`

Reads the file identified by *file_name* and returns the file contents as a string. The file must be on the server, and the user must specify the full pathname to the file and have permission to access the file.

`LOCATE(substr, str), POSITION(substr IN str)`

Returns the position of the first occurrence of the substring *substr* in the string *str*; returns 0 if *substr* is not in *str*. *LOCATE* is a synonym for the standard `POSITION(substr IN str)`. For example:

```
LOCATE('al','Donald') -> 4
POSITION('al' IN 'Donald') -> 4
```

`LOCATE(substr, str, pos)`

Returns the position of the first occurrence of the substring *substr* in the string *str*, starting at position *pos*; returns 0 if *substr* is not in *str*. For example:

```
LOCATE('World', 'Hello, World!') -> 8
```

`LOG(X)`

Returns the natural logarithm of *X*. For example:

```
LOG(50) -> 3.912023
```

`LOG2(X)`

Returns the base-2 logarithm of *X*. For example:

```
LOG2(50) -> 5.64386
```

`LOG10(X)`

Returns the base-10 logarithm of *X*. For example:

```
LOG10(50) -> 1.698970
```

`LPAD(str, len, padstr)`

Returns the string *str*, left-padded with the string *padstr* until *str* is *len* characters long. For example:

```
LPAD('ucks',6,'d') -> 'dducks'
```

`LTRIM(str)`

Returns the string *str* with leading-space characters removed. For example:

```
LTRIM(' Howdy! ') -> 'Howdy! '
```

`MAKE_SET(bits, str1, str2[, ...n])`

Returns a *set* (a string containing substrings separated by commas) consisting of the string arguments that have the corresponding bit in *bits* set; *str1* corresponds to bit 0, *str2* to bit 1, etc. NULL strings in *str1*, *str2*, ... are not appended to the result. For example:

```
MAKE_SET(1 | 4,'hello','nice','world') -> 'hello,world'
```

`MAKEDATE(y, n)`

Returns a date corresponding to the year *y* and the day number *n*. For example:

```
MAKEDATE(2008, 1) -> '2008-01-01'
```

MAKETIME(*hour, minute, second*)

Returns a time matching *hour:minute:second*. For example:

MAKETIME(2, 30, 0) -> '02:30:00'

MATCH(*col1, col2, ...*) AGAINST (*expr [search_modifier]*)

Performs a full-text search looking for *expr* in the supplied columns. See the MySQL documentation for more information on full-text searching.

MD5(*string*)

Calculates an MD5 checksum for the *string*. The value is returned as a 32-digit-long hex number. For example:

MD5('somestring') -> 1f129c42de5e4f043cbd88ff6360486f

MICROSECOND(*time*)

Returns the microseconds for *time*, in the range 0 to 999999. For example:

MICROSECOND('08:20:15.000050') -> 50

MID(*str, pos, len*)

Synonym for *SUBSTRING(str, pos, len)*.

MINUTE(*time*)

Returns the minute for *time*, in the range 0 to 59. For example:

MINUTE('08:20:15') -> 20

MONTH(*date*)

Returns the month for *date*, in the range 1 to 12. For example:

MONTH('1999-04-15') -> 4

MONTHNAME(*date*)

Returns the name of the month for *date*. For example:

MONTHNAME('1999-04-15') -> 'April'

NOW(), SYSDATE()

Returns the current date and time as a value in YYYY-MM-DD HH:MM:SS or YYYYMMDDHHMMSS format, depending on whether the function is used in a string or numeric context. For example:

NOW() -> 2003-06-24 20:40:24

SYSDATE() -> 2003-06-24 20:40:24

CURRENT_TIMESTAMP -> 2003-06-24 20:40:24

NULLIF(*expr1, expr2*)

Returns NULL if *expr1* is equal to *expr2*; otherwise, returns *expr1*. For example:

NULLIF(2,29) -> 2**NULLIF(29,29)** -> NULL

OCT(*n*)

Returns an octal value equivalent of *n*, where *n* is a number. This is equivalent to *CONV(N,10,8)*. Returns NULL if *n* is NULL. For example:

OCT(255) -> 377

`OLD_PASSWORD(str)`

Calculates a password string from the plain-text password *str*. This is the function that is used for encrypting MySQL passwords. The “OLD_” prefix was added in version 4.1, when the password hashing changed to improve security. For example:

```
OLD_PASSWORD('password') -> '5d2e19393cc5ef67'
```

`ORD(str)`

Returns the character ordinal of the multibyte character string *str*. The value is calculated using the following formula: ((first byte ASCII code) * 256 + (second byte ASCII code) * 256 * 256) (third byte ASCII code) * 256 * 256 * 256[, ...]. If *str* isn’t a multibyte character, this function returns the same value as the *ASCII* function. For example:

```
ORD('29') -> 50
```

`PASSWORD(str)`

Calculates a password string from the plain-text password *str*. This is the function that is used for encrypting MySQL passwords. For example:

```
PASSWORD('password') -> '5d2e19393cc5ef67'
```

`PERIOD_ADD(period, months)`

Adds the number of months found in *months* to the period in *period* (in the format YYMM or YYYYMM). Returns a value in the format YYYYMM. For example:

```
PERIOD_ADD(9902,3) -> 199905
```

`PERIOD_DIFF(period1, period2)`

Returns the number of months between *period1* and *period2*. *period1* and *period2* should be in the format YYMM or YYYYMM. For example:

```
PERIOD_DIFF(9902,9905) -> -3
```

`PI()`

Returns the value of π . For example:

```
PI() -> 3.141593
```

`POW(X, Y), POWER(X, Y)`

Returns the value of *X* raised to the power of *Y*. For example:

```
POW(2, 8) -> 256.000000
```

`QUARTER(date)`

Returns the quarter of the year for *date*, in the range 1 to 4. For example:

```
QUARTER('1999-04-15') -> 2
```

`QUOTE(str)`

Returns *str* with special characters properly escaped for usage within a SQL statement. For example:

```
QUOTE('\'start and end with quote\'') -> '\'start and end with quote\''
```

`RADIANS(X)`

Returns the argument *X*, converted from degrees to radians. For example:

```
RADIANS(180) -> 3.1415926535898
```

RAND(), RAND(*N*)

Returns a random floating-point value in the range 0 to 1.0. If an integer argument *N* is specified, it is used as the seed value. For example:

```
RAND() -> 0.29588872501244
```

expr* REGEXP *pat*, *expr* RLIKE *pat

Returns 1 if *expr* matches the regular expression pattern in *pat*; otherwise, returns 0. For example:

```
SELECT 'oolong' REGEXP '^[a-z]' -> 1
```

RELEASE_LOCK(*str*)

Releases the lock named by the string *str* that was obtained with *GET_LOCK*. Returns 1 if the lock is released, or NULL if the named lock doesn't exist or isn't locked by this thread (in which case the lock is not released). For example:

```
RELEASE_LOCK('lochness') -> 1
```

REPEAT(*str*, *count*)

Returns a string consisting of the string *str* repeated *count* times. For example:

```
REPEAT('Duck', 3) -> 'DuckDuckDuck'
```

REPLACE(*str*, *from_str*, *to_str*)

Returns the string *str* with all occurrences of the string *from_str* replaced by the string *to_str*. For example:

```
REPLACE('change', 'e', 'ing') -> 'changing'
```

REVERSE(*str*)

Returns the string *str* reversed. For example:

```
REVERSE('STOP') -> 'POTS'
```

RIGHT(*str*, *int*)

Returns the rightmost 10 characters of the string *str*. For example:

```
RIGHT('Hello, World!', 6) -> 'World!'
```

ROUND(*X*, *D*)

Returns the argument *X*, rounded to a number with *D* decimals. If *D* is 0, the result has no decimal point or fractional part. For example:

```
ROUND(12345.6789, 2) -> 12345.68
```

ROW_COUNT()

Returns the number of rows updated in the previous statement. For example:

```
SELECT ROW_COUNT() -> 4
```

RPAD(*str*, *len*, *padstr*)

Returns the string *str*, right-padded with the string *padstr* until *str* is *len* characters long. For example:

```
RPAD('duck',6,'s') -> 'duckss'
```

RTRIM(*str*)

Returns the string *str* with trailing space characters removed. For example:

```
RTRIM(' welcome ') -> 'welcome '
```

SCHEMA()

Synonym for *DATABASE*.

SEC_TO_TIME(seconds)

Returns the *seconds* argument, converted to hours, minutes, and seconds, as a value in HH:MM:SS or HHMMSS format, depending on whether the function is used in a string or numeric context. For example:

SEC_TO_TIME(256) -> 00:04:16

SECOND(*time*)

Returns the seconds for *time*, in the range 0 to 59. For example:

SECOND('08:20:15') -> 15

SESSION_USER()

Synonym for *USER*.

SHA(*X*) or SHA1(*X*)

Returns a SHA1 160-bit checksum for *X*. For example:

SHA('abc') -> 'a9993e364706816aba3e25717850c26c9cd0d89d'

SIGN(*X*)

Returns the sign of the argument as -1, 0, or 1, depending on whether *X* is negative, zero, or positive. For example:

SIGN(-3.1415926) -> -1

SIN(*number*)

Returns the sine of *number*, where *number* is in radians. For example:

SELECT SIN(0) -> 0.000000

SLEEP(*s*)

Sleeps for *s* seconds. For example:

SLEEP(60) -> 0

SOUNDEX(*str*)

Returns a soundex string from *str*. For example:

SOUNDEX('thimble') -> 'T514'

expr1* SOUNDS LIKE *expr2

Synonymous with the expression:

SOUNDEX(*expr1*) = SOUNDEX(*expr2*)

SPACE(*n*)

Returns a string consisting of *n* space characters. For example:

SPACE(5) -> ' '

STD(*expr*), STDDEV(*expr*)

Returns the standard deviation of *expr*. The *STDDEV* form of this function is provided for Oracle compatibility. For example:

STD(5) -> NULL

STR_TO_DATE(*str, format*)

Returns a date parsed from *str* using the format specifiers found in the *format* argument. This is the reverse of *DATE_FORMAT*; please see the *DATE_FORMAT* coverage for a list of the supported specifiers. For example:

```
STR_TO_DATE('28/08/1976', '%d/%m/%Y') -> '1976-08-28'
```

STRCMP(*expr1, expr2*)

STRCMP returns 0 if the strings are the same, -1 if the first argument is smaller than the second according to the current sort order, and 1 otherwise. For example:

```
STRCMP('DUCKY', 'DUCK') -> 1
STRCMP('DUCK', 'DUCK') -> 0
```

SUBSTRING(*str, pos*), SUBSTRING(*str FROM pos*)

Returns a substring from the string *str* starting at the position *pos*. For example:

```
SUBSTRING('Hello, World!', 8) -> 'World!'
SUBSTRING('Hello, World!' FROM 8) -> 'World!'
```

SUBSTRING(*str, pos, len*), MID(*str, pos, len*)

Returns a substring *len* characters long from the string *str*, starting at the position *pos*. These functions are synonyms for the ANSI SQL92 function *SUBSTRING(str FROM pos FOR len)*. For example:

```
SUBSTRING('Hello, World!', 8, 10) -> 'World!'
SUBSTRING('Hello, World!' FROM 8 FOR 10) -> 'World!'
```

SUBSTRING_INDEX(*str, delim, count*)

Returns the substring *str* after *count* occurrences of the delimiter *delim*. For example:

```
SUBSTRING_INDEX('www.mysql.com', '.', 2) -> 'www.mysql'
```

SUBTIME(*expr1, expr2*)

Returns the result of subtracting *expr2* from *expr1*. For example:

```
SUBTIME('2008-01-31 16:30:00.999999', '0 0:30:0.999996') -> '2008-01-31
16:00:00.000003'
```

TAN(*number*)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TIME(*expr*)

Returns the time portion of the value found in *expr*. For example:

```
TIME('1976-08-28 08:20:15') -> '08:20:15'
```

TIME_FORMAT(*time, format*)

Used like *DATE_FORMAT*, but the *format* string may contain only those format specifiers that handle hours, minutes, and seconds. Other specifiers produce a NULL value or 0. See *DATE_FORMAT* for a list of the available format specifiers. For example:

```
TIME_FORMAT('2003-04-15 08:20:15', '%r') -> 08:20:15 AM
```

TIME_TO_SEC(*time*)

Returns the *time* argument, converted to seconds. For example:

```
TIME_TO_SEC('08:20:15') -> 30015
```

TIMEDIFF(*expr1*, *expr2*)

Returns the difference between *expr1* and *expr2*. For example:

```
TIMEDIFF('1976-08-28 08:20:15', '1976-08-28 08:20:16') ->  
'00:00:01.000000'
```

TIMESTAMP(*expr1*[, *expr2*])

Returns a timestamp value where the date comes from *expr1* and the time component comes from *expr2*. For example:

```
TIMESTAMP('1976-08-28', '08:20:16') -> '1976-08-28 08:20:16'
```

TIMESTAMPADD(*unit*, *interval*, *expr*)

Returns a timestamp constructed by adding *interval* to the date expression *expr*. The units for *interval* are specified by *unit*. For example:

```
TIMESTAMPADD(DAY, 2, '1976-08-28') -> '1976-08-30 00:00:00'
```

TIMESTAMPDIFF(*unit*, *expr1*, *expr2*)

Returns an integer that is the result of subtracting *expr1* from *expr2*. The units of the result are specified by *unit*. For example:

```
TIMESTAMPADD(DAY, '1976-08-30', '1976-08-28') -> -2
```

TO_DAYS(*date*)

Given a *date*, returns a day number (the number of days since the year 0). For example:

```
TO_DAYS('1999-04-15') -> 730224
```

TRUNCATE(*X*, *D*)

Returns the number *X*, truncated to *D* decimal places. If *D* is 0, the result has no decimal point or fractional part. For example:

```
TRUNCATE('123.456', 2) -> '123.45'  
TRUNCATE('123.456', 0) -> '123'  
TRUNCATE('123.456', -1) -> '120'
```

UCASE(*str*)

Synonym for *UPPER(str)*. For example:

```
UCASE('duck') -> 'DUCK'
```

UNCOMPRESS(*string*)

Returns an uncompressed version of *string*.

UNCOMPRESS_LENGTH(*string*)

Returns the length of *string* in its uncompressed form.

UNHEX(*str*)

Returns a binary string constructed from hex characters in *str*.

UNIX_TIMESTAMP(), UNIX_TIMESTAMP(*date*)

If called with no argument, returns a Unix timestamp (seconds since 1970-01-01 00:00:00 GMT). If called with a *date* argument, returns the value of the argument as seconds since 1970-01-01 00:00:00 GMT. For example:

```
UNIX_TIMESTAMP() -> 1056512427
UNIX_TIMESTAMP('1999-04-15') -> 924159600
```

UPDATEXML(*xml_target*, *xpath_expr*, *new_xml*)

Returns the XML fragment in *xml_target* with *new_xml* at locations specified by the XPath expression *xpath_expr*. For example:

```
UPDATEXML('<c><v>unknown</v></c>', '//v', '<v>Acme</v>') -> '<c><v>
Acme</v></c>'
```

USER(), SYSTEM_USER(), SESSION_USER()

These functions return the current MySQL username. For example:

```
USER() -> 'login@machine.com'
SYSTEM_USER() -> 'login@machine.com'
SESSION_USER() -> 'login@machine.com'
```

UTC_DATE()

Returns the current UTC date. For example:

```
UTC_DATE() -> '2008-05-15'
```

UTC_TIME()

Returns the current UTC time. For example:

```
UTC_TIME() -> '01:01:00'
```

UTC_TIMESTAMP()

Returns the current UTC date and time. For example:

```
UTC_TIMESTAMP() -> '2008-05-15 01:01:00'
```

UUID()

Returns a Universal Unique Identifier. For example:

```
UUID() -> '1ae84bc9-5e4d-8f22-1f2e-123456789abc'
```

VARIANCE(*expr*)

Synonym for *VAR_POP*.

VERSION()

Returns a string indicating the MySQL server version. For example:

```
VERSION() -> '4.0.12-standard'
```

WEEK(*date*), WEEK(*date*, *first*)

With a single argument, returns the week for *date*, in the range 1 to 53. (The beginning of a week 53 is possible during some years.) The two-argument form of *WEEK* allows the user to specify whether the week starts on Sunday (0) or Monday (1). For example:

```
WEEK('1999-04-15') -> 15
```

WEEKDAY(*date*)

Returns the weekday index for *date* (0 = Monday, 1 = Tuesday,...6 = Sunday). For example:

```
WEEKDAY('1999-04-15') -> 3
```

WEEKOFYEAR(*date*)

Returns the calendar week for *date*, where the calendar week is an integer between 1 and 53, inclusive. For example:

```
WEEKOFYEAR('2008-01-01') -> 1
```

op1 XOR op2

Returns the logical XOR of *op1* and *op2*. For example:

```
SELECT 1 XOR 1, 1 XOR 0 -> 0, 1
```

YEAR(date)

Returns the year for *date*, in the range 1,000 to 9,999. For example:

```
YEAR('1999-04-15') -> 1999
```

YEARWEEK(date), *YEARWEEK(date, first)*

Returns the year and week for *date*. The second argument works exactly like the second argument to *WEEK*. Note that the year may be different from the year in the *date* argument for the first and the last week of the year. For example:

```
YEARWEEK('1999-04-15') -> 199915
```

Oracle-Supported Functions

This section provides an alphabetical listing of the SQL functions specific to Oracle, with examples and corresponding results.

ACOS(number)

Returns the arccosine of *number* ranging from -1 to 1 . The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) FROM DUAL -> 1.570796
```

ADD_MONTHS(date, int)

Returns the *date* plus *int* months. For example:

```
SELECT ADD_MONTHS('15-APR-1999', 3) FROM DUAL -> 15-JUL-99
```

APPENDCHILDXML(xml_fragment, xpath, value[, namespace])

Injects *value* into the XML fragment *xml_fragment* at the location specified by the XPath expression *xpath* and returns the result. The optional *namespace* argument provides the namespace information for the XPath expression. For example:

```
SELECT APPENDCHILDXML('<a><b>B</b></a>', '/a', '<c>C</c>') FROM DUAL
'<a><b>B</b><c>C</c></a>'
```

ASCII(text)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') FROM DUAL -> 120
```

ASCIIISTR(text)

Converts *text* from any character set into an ASCII equivalent. Characters in *text* that have no equivalent in ASCII will be replaced with the string `\XXXX`, where `XXXX` represents the UTF-16 code unit. For example:

```
SELECT ASCIIISTR('ÄBC') FROM DUAL -> '\00C4BC'
```

ASIN(number)

Returns the arcsine of *number* ranging from -1 to 1 . The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) FROM DUAL -> 0.000000
```

ATAN(*number*)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) FROM DUAL -> 1.262619
```

ATAN2(*number*, *nbr*)

Returns the arctangent of *number* and *nbr*. ATAN2(*x*, *y*) is similar to ATAN(*y/x*), with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

```
SELECT ATAN2(3.1415, 1) FROM DUAL -> 1.26261873
```

BFILENAME(*directory*, *filename*)

Returns a *BFILE* locator associated with a physical *LOB* binary file on the server's filesystem in *directory* with the name *filename*.

BIN_TO_NUM(*expr*[, ... *n*])

Returns a decimal number equivalent of the binary bit vector contained in the *expr* arguments. For example:

```
SELECT BIN_TO_NUM(1,0,1) FROM DUAL -> 5
```

BITAND(*integer1*, *integer2*)

Returns the bitwise AND of the two integer arguments. For example:

```
SELECT BITAND(101, 2) FROM DUAL -> 0
```

```
SELECT BITAND(column1, 1) FROM DUAL -> 1
```

CARDINALITY(*nested_table*)

Returns the number of elements (cardinality) of the *nested_table*. If the *nested_table* is empty, returns NULL. For example:

```
SELECT CARDINALITY(mytable) FROM DUAL -> 6
```

CHARTOROWID(*char*)

Converts a value from a character datatype (CHAR or VARCHAR2 datatype) to a ROWID datatype.

CHR(*number* [USING NCHAR_CS])

Returns the character having the binary equivalent to *number* in either the database character set (if USING NCHAR_CS is not included) or the national character set (if USING NCHAR_CS is included).

CLUSTER_ID(), CLUSTER_PROBABILITY(), CLUSTER_SET()

Support data mining features. See the documentation for the Oracle Data Mining Java API or the DBMS_DATA_MINING package for more details on these functions.

COALESCE(*list*)

Returns the first non-NULL element in the list. For example:

```
SELECT COALESCE( NULL, 1, 2 ) FROM DUAL -> 1
```

COLLECT(*column*)

Creates for each group a nested table consisting of all the values in the *column*. This is an aggregate function.

COMPOSE(*string*)

Returns *string* as a fully normalized UNICODE string.

CONCAT(*string1, string2*)

Returns *string1* concatenated with *string2*. **CONCAT** is equivalent to the concatenation operator (||). For example:

```
SELECT CONCAT( au_lname, au_fname ) FROM AUTHORS -> 'JeffersonThomas'
```

CONVERT(*char_value, target_char_set, source_char_set*)

Converts a character string from one character set to another; returns *char_value* in the *target_char_set* after converting *char_value* from the *source_char_set*.

CORR_K(*expr1, expr2[, return_type]*)

CORR_S(*expr1, expr2[, return_type]*)

CORR_K returns Kendall's tau-b correlation coefficient, and **CORR_S** returns Spearman's rho correlation coefficient for a set of numbered pairs (*expr1* and *expr2*). The *return_type* argument, a *VARCHAR2*, can be omitted or can one of the following values: 'COEFFICIENT', 'ONE_SIDED_SIG', or 'TWO_SIDED_SIG'. The value 'COEFFICIENT' (the default if this argument is omitted) returns the coefficient of the correlation. The values 'ONE_SIDED_SIG' and 'TWO_SIDED_SIG' return the one- and two-tailed significance of the correlation, respectively.

COS(*number*)

Returns the cosine of *number* as an angle expressed in radians. For example:

```
SELECT COS(0) FROM DUAL -> 1.000000
```

COSH(*number*)

Returns the hyperbolic cosine of *number*. For example:

```
SELECT COSH(180) FROM DUAL -> 7.4469E+77
```

COT(*number*)

Returns the cotangent of *number*. For example:

```
SELECT COT( 3.1415 ) -> -10792.88993953
```

CUBE_TABLE('expr')

Extracts a two-dimensional relational view from an OLAP cube or dimension. See the Oracle OLAP documentation for more information.

CV([*dimension_column*])

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns the current value of the *dimension_column*. **CV** can only be used in the righthand side of a rule, since it returns the value of the *dimension_column* from the lefthand side of the same rule.

DATAOBJ_TO_PARTITION(*table, partition_id*)

Returns the partition identifier for the system-partitioned table specified by the arguments. See the Oracle documentation for more information on this function.

DBTIMEZONE

Returns the time zone offset from UTC time for the database server. For example:

```
SELECT DBTIMEZONE FROM DUAL -> +00:00
```

DECODE(expr, search, result[, search, result[, ... n]][, default])

Compares *expr* to the *search* value; if *expr* is equal to *search*, it returns the result. For example:

```
DECODE ('B','A',1,'B',2,...'Z',26,'?') -> 2
```

Without a match, *DECODE* returns *default*, or NULL if *default* is omitted. Refer to the Oracle documentation for more details. Consider using *CASE* instead, as *CASE* is part of the ANSI SQL standard.

DECOMPOSE(string [{CANONICAL | COMPATIBILITY}])

Returns *string* decomposed into UNICODE code-points. The second argument specifies the type of decomposition performed. *CANONICAL*, which specifies the default behavior, allows the original UNICODE string to be recomposed.

DELETEXML(xml_fragment, xpath[, namespace])

Deletes nodes within *xml_fragment* that match the XPath expression *xpath* and returns the result. The optional *namespace* parameter specifies the namespace for the XPath expression. For example:

```
SELECT DELETEXML('<a><b>Sifl</b><c>0lly</c></a>', '/a/c') FROM DUAL
'<a><b>Sifl</b></a>'
```

DEPTH(number)

Returns the depth of the path specified by the *UNDER_PATH* condition in an XML query. See the Oracle SQL Reference for more information.

DEREF(expression)

Returns the object referenced by *expression*, where *expression* must return a *REF* to an object.

DUMP(expression[, return_format[, starting_at[, length]]])

Returns a VARCHAR2 value containing a datatype code, *length* in bytes, and internal representation of *expression*. The resulting value is returned in the format of *return_format*. For example:

```
SELECT DUMP('abc', 1016) FROM DUAL
Typ=96 Len=3 CharacterSet=AL32UTF8: 61,62,63
```

EMPTY_BLOB(), EMPTY_CLOB()

Returns an empty *LOB* locator that can be used to initialize a *LOB* variable. It can also be used to initialize a *LOB* column or attribute to empty in an *INSERT* or *UPDATE* statement.

EXISTSNODE(instance, xpath[, namespace])

Returns 1 if applying the XPath query in *xpath* would return any nodes from *instance*; otherwise, returns 0. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference.

EXTRACT(instance, xpath[, namespace])

Returns the XML nodes from *instance* returned by running the XPath query contained in the *xpath* parameter. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries,

refer to the Oracle SQL Reference. (Oracle also supports an *EXTRACT* function for date values, which was covered earlier in this chapter.) For example:

```
SELECT EXTRACT( XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),
  '/foo/bar' ) from DUAL
```

```
<bar>Hello, World!</bar>
```

EXTRACTVALUE(instance, xpath[, namespace])

Returns the value from an XML node returned by running the XPath query contained in the *xpath* parameter. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

```
SELECT EXTRACTVALUE( XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),
  '/foo/bar' ) from DUAL
```

```
Hello, World!
```

FEATURE_ID(), *FEATURE_SET()*, *FEATURE_VALUE()*

Support data mining features. See the documentation for the Oracle Data Mining Java API or the *DBMS_DATA_MINING* package for more details on these functions.

FIRST

Aggregate function that returns a specified value from the row that ranks first, given the order specified in the *ORDER BY* clause. The syntax is:

```
aggregate(aexpr) KEEP (DENSE_RANK FIRST ORDER BY expr[, ... n])
```

where the syntax of *expr* is:

```
expr := [ASC | DESC] [NULLS {FIRST | LAST}]
```

The first ranking row following the order specified by *expr* will be used in the aggregate function *aggregate*. *aexpr* is the expression passed to the aggregate function. For example:

```
SELECT MAX(c1) KEEP (DENSE_RANK FIRST ORDER BY c2) FROM FIVE_NUMS
  1
```

FIRST_VALUE(expression IGNORE NULLS) OVER (window_clause)

Returns the first value in an ordered set of values. *FIRST_VALUE* is an analytic function. See the “ANSI SQL Window Functions” section, earlier in this chapter, for a detailed explanation of the *window_clause*. For example:

```
SELECT FIRST_VALUE(col1) OVER () FROM NUMS
  1
  1
  1
  1
```

FROM_TZ(timestamp, timezone)

Returns *timestamp* converted to a *TIMESTAMP WITH TIME ZONE* value, where *timestamp* is a *TIMESTAMP* value and *timezone* is a string in the TZH:TZM format. For example:

```
SELECT FROM_TZ(TIMESTAMP '2004-04-15 23:59:59', '8:00') FROM DUAL
  '15-APR-04 11.59.59 PM +08:00'
```

GREATEST(*expression1*, ... *n*)

Returns the greatest of the list of *expressions*. All *expressions* after the first are implicitly converted to the datatype of the first expression before the comparison. For example:

```
SELECT GREATEST(8,2,4) FROM DUAL -> 8
```

GROUP_ID()

Returns a positive value for each duplicate group returned by a query containing a *GROUP BY* clause. This function is useful in filtering out duplicate groups created when using *CUBE*, *ROLLUP*, or another *GROUP BY* extension (see *GROUPING*).

GROUPING(*column_name*)

Returns 1 when a row is added by *CUBE*, *ROLLUP*, or another *GROUP BY* extension; otherwise, returns 0. For example:

```
SELECT royalty, SUM(advance) 'total advance',
GROUPING(royalty) 'grp'
FROM titles
GROUP BY royalty WITH ROLLUP
royalty      total advance      grp
-----      -----      --
NULL        NULL            0
10          57000.0000        0
12          2275.0000        0
14          4000.0000        0
16          7000.0000        0
24          25125.0000       0
NULL        95400.0000       1
```

GROUPING_ID(*column_name1*[, *column_name2*, ...])

Returns the base-10 number that is equal to the binary value constructed by concatenating the *GROUPING* values on each of the parameters. *GROUPING_ID* is useful when returning a query containing multiple levels of aggregation created by *GROUP BY* expressions. Consider using the *GROUPING_ID* function instead of multiple *GROUPING* functions within one query. This function is a shorthand equivalent of:

```
BIN_TO_NUM( GROUPING(column_name1)[, GROUPING(column_name2), ...] )
```

HEXTORAW(*string*)

Converts a *string* containing hexadecimal digits into a raw value. For example:

```
SELECT HEXTORAW('0FE') FROM DUAL -> '00FE'
```

INITCAP(*string*)

Returns *string*, with the first letter of each word in uppercase and all other letters in lowercase. For example:

```
SELECT INITCAP('thomas jefferson') FROM DUAL -> 'Thomas Jefferson'
```

INSERTCHILDXML(*xml_fragment*, *xpath*, *child_expr*, *value_expr*[, *namespace*])

Injects the nodes specified by *child_expr* within *value_expr* into *xml_fragment* at the location given by the XPath query *xpath*, and returns the result. The optional *namespace* argument provides the namespace for the XPath query. For example:

```
SELECT INSERTCHILDXML('<a></a>', '/a', 'b', '<b>B1</b><b>B2</b>')
FROM DUAL
'<a><b>B1</b><b>B2</b></a>'
```

INSERTXMLBEFORE(*xml_fragment*, *xpath*, *value_expr*[, *namespace*])

Injects *value_expr* into *xml_fragment* at the location given by the XPath query *xpath* and returns the result. The optional *namespace* argument provides the namespace for the XPath query. For example:

```
SELECT INSERTXMLBEFORE('<a><b>B2</b></a>', '/a/b', '<b>B1</b>') FROM DUAL
'<a><b>B1</b><b>B2</b></a>'
```

INSTR(*string1*, *string2*[, *start_at*[, *occurrence*]])

Returns the position of *string2* within *string1*. *INSTR* searches *string1* from a starting position of *start_at* (an integer), looking for the specified *occurrence* of *string2*. For example:

```
SELECT INSTR('foobar', 'o', 1, 1) FROM DUAL -> 2
```

Use *INSTRB* for bytes, *INSTRC* for UNICODE complete characters, *INSTR2* for UNICODE UCS2 code points, and *INSTR4* for UNICODE UCS4 code points.

ITERATION_NUMBER

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns the number of times the rules within the *MODEL* clause have been executed while processing the query.

LAG(*expression*[, *offset*][, *default*]) OVER (*window_clause*)

Analytic function that provides access to more than one row of a table at the same time without a self-join. *LAG* provides a “lagging” value in the result set that lags *offset* rows behind the current row. The *default* value is used for the first *offset* rows in the result set, since the “lagging” value is undefined for these rows. See “ANSI SQL Window Functions,” earlier in this chapter, for a detailed explanation of the *window_clause*. For example:

```
SELECT c1, LAG(c1, 2, 0) OVER (ORDER BY c1) FROM FIVE_NUMS
1      0
2      0
3      1
4      2
5      3
```

LAST

Returns the row that ranks last given the order specified in the *ORDER BY* clause. The syntax is:

```
aggregate(aexpr) KEEP (DENSE_RANK LAST ORDER BY expr[, ... n])
```

where the syntax of *expr* is:

```
expr := [ASC | DESC] [NULLS {FIRST | LAST}]
```

The last ranking row following the order specified by *expr* will be used in the aggregate function *aggregate*. The *aexpr* is the expression passed to the aggregate function. For example:

```
SELECT MIN(c1) KEEP (DENSE_RANK LAST ORDER BY c1) FROM FIVE_NUMS
```

LAST_DAY(*date*)

Returns the date of the last day of the month that contains *date*. For example:

```
SELECT LAST_DAY('15-APR-1999') FROM DUAL -> 30-APR-99
```

LAST_VALUE(*expression* [IGNORE NULLS]) OVER (*window_clause*)

Returns the last value in an ordered set of values. See “ANSI SQL Window Functions,” earlier in this chapter, for a detailed explanation of the *window_clause*. For example:

```
SELECT c1, LAST_VALUE(c1) OVER (ORDER BY c1) FROM FIVE_NUMS
1      5
2      5
3      5
4      5
5      5
```

LEAD(*expression*[, *offset*][, *default*]) OVER (*window_clause*)

Analytic function that provides access to more than one row of a table at the same time without a self-join. *LEAD* provides a “leading” value in the result set that is *offset* rows ahead of the current row. The *default* value is used for the last *offset* rows in the result set, since the “leading” value is undefined for these rows. See “ANSI SQL Window Functions,” earlier in this chapter, for a detailed explanation of the *window_clause*. For example:

```
SELECT c1, LEAD(c1, 2) OVER (ORDER BY c1) FROM FIVE_NUMS
1      3
2      4
3      5
4
5
```

LEAST(*expression*[, ... *n*])

Returns the least of the list of expressions. For example:

```
SELECT LEAST(10,5,3,7) FROM DUAL -> 3
```

LENGTH(*string*)

Returns the integer length of *string*, or NULL if *string* is NULL. For example:

```
SELECT LENGTH('DUCK') FROM DUAL -> 4
```

LENGTHB(*string*)

Returns the length of *char* in bytes; otherwise, the same as LENGTH. For example:

```
SELECT LENGTHB('DUCK') FROM DUAL -> 4
```

Use LENGTHB for bytes, LENGTHC for UNICODE complete characters, LENGTH2 for UNICODE UCS2 code points, and LENGTH4 for UNICODE UCS4 code points.

LNNVL(*condition*)

Returns TRUE if *condition* is false or if one of the operands in *condition* is NULL; otherwise, returns FALSE. For example:

```
SELECT COUNT(*) FROM authors WHERE LNNVL( contract <> 1 ) -> 4
```

LOCALTIMESTAMP[*precision*]

Returns a *TIMESTAMP* value for the current date and time. This function is similar to *CURRENT_TIMESTAMP*, with the exception that this function does not return a *TIMEZONE* value with the *TIMESTAMP*. For example:

```
SELECT LOCALTIMESTAMP FROM DUAL -> '15-APR-05 03.15.00 PM'
```

LOG(*base_number*, *number*)

Returns the logarithm of any *base_number* of *number*. For example:

```
SELECT LOG(50,10) FROM DUAL -> .58859191
```

LPAD(*string1*, *number*[, *string2*])

Returns *string1*, left-padded to length *number* using characters in *string2*; *string2* defaults to a single blank. For example:

```
SELECT LPAD('ucks',5,'d') FROM DUAL -> 'ducks'
```

LTRIM(*string*[, *set*])

Removes all characters in *set* from the left of *string*. *set* defaults to a single blank. For example:

```
SELECT LTRIM(' Howdy! ',' ') FROM DUAL -> 'Howdy! '
```

MAKE_REF({*table_name* | *view_name*}, *key*[, ... *n*])

Creates a reference (REF) to a row of an object view or a row in an object table whose object identifier is primary key-based.

MEDIAN(*expression*) OVER (*partitioning*)

Returns the median value in an ordered set of numeric or datetime values. See “ANSI SQL Window Functions,” earlier in this chapter for a detailed explanation of the *partitioning* clause. For example:

```
SELECT MEDIAN(c1) FROM FIVE_NUMS -> 3
```

MONTHS_BETWEEN(*date1*, *date2*)

Returns the number of months between the dates *date1* and *date2*. When *date1* is later than *date2*, the result is positive. When it is earlier, the result is negative. For example:

```
SELECT MONTHS_BETWEEN('15-APR-2000', '15-JUL-1999') FROM DUAL -> 9
```

NANVL(*a*, *b*)

Returns *b* when *a* is not a number (*NaN*); returns *a* otherwise. The expression *a* must evaluate to a *BINARY_FLOAT* or *BINARY_DOUBLE* number, which are the only number types that permit storing *NaN*. For example:

```
SELECT c1, NANVL(c1, 0) FROM NUMS
```

1.0E+000	1.0E+000
2.0E+000	2.0E+000
Nan	0

NCHAR(*number*)

Synonym for *CHR*(*number*) USING *NCHAR_CS*.

NEW_TIME(*date*, *time_zone1*, *time_zone2*)

Returns the date and time in *time_zone2*, using *date* as the input date/time and using *time_zone1* as the originating time zone. For example:

```

ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH12:MI:SS'
SELECT NEW_TIME(TO_DATE('04-15-99 08:22:31', 'MM-DD-YY HH12:MI:SS'),
  'AST', 'PST')
FROM DUAL
15-APR-2099 04:22:31

```

time_zone1 and *time_zone2* may be any of these text strings:

‘AST’, ‘ADT’

Atlantic Standard or Daylight Time

‘BST’, ‘BDT’

Bering Standard or Daylight Time

‘CST’, ‘CDT’

Central Standard or Daylight Time

‘EST’, ‘EDT’

Eastern Standard or Daylight Time

‘GMT’

Greenwich Mean Time

‘HST’, ‘HDT’

Alaska-Hawaii Standard or Daylight Time

‘MST’, ‘MDT’

Mountain Standard or Daylight Time

‘NST’

Newfoundland Standard Time

‘PST’, ‘PDT’

Pacific Standard or Daylight Time

‘YST’, ‘YDT’

Yukon Standard or Daylight Time

NEXT_DAY(date, string)

Returns the date of the first weekday named by *string* that is later than *date*. The argument *string* must be either the full name or the abbreviation of a day of the week in the date language of the session. For example:

```

ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY'
SELECT NEXT_DAY('15-APR-1999', 'SUNDAY') FROM DUAL
18-APR-1999

```

NLS_CHARSET_DECL_LEN(bytecnt, csid)

Returns the declaration width (*bytecnt*) of an *NCHAR* column using the character set ID (*csid*) of the column.

NLS_CHARSET_ID(text)

Returns the NLS character set ID number corresponding to *text*.

NLS_CHARSET_NAME(number)

Returns the *VARCHAR2* name for the NLS character set corresponding to the ID *number*.

NLS_INITCAP(*string*[, *nlsparameter*])

Returns *string* with the first letter of each word in uppercase and all other letters in lowercase. The *nlsparameter* offers special linguistic sorting features.

NLS_LOWER(*string*[, *nlsparameter*])

Returns *string* with all letters in lowercase. The *nlsparameter* offers special linguistic sorting features.

NLS_UPPER(*string*[, *nlsparameter*])

Returns *string* with all letters in uppercase. The *nlsparameter* offers special linguistic sorting features.

NLS_SORT(*string*[, *nlsparameter*])

Returns the string of bytes used to sort *string*. The *nlsparameter* offers special linguistic sorting features.

NTILE(*expression*) OVER ([*partitioning*] *ordering*)

Divides an ordered data set into a number of groups numbered from 1 to *expression* and assigns the appropriate group number to each row. Rows are allocated to each group so that the number of rows per group varies by no more than 1. See “ANSI SQL Window Functions,” earlier in this chapter, for details on the *partitioning* and *ordering* clauses. For example:

```
SELECT c1, NTILE(4) OVER (ORDER BY c1) FROM FIVE_NUMS
 1      1
 2      1
 3      2
 4      3
 5      4
```

NULLIF(*expr1*, *expr2*)

Returns NULL if *expr1* is equal to *expr2*; otherwise, returns *expr1*. When passing a NULL value as one of the expressions, it can only be specified as the second expression (*expr2*). For example:

```
SELECT c1, c2, NULLIF(c1, c2) FROM NUMS
 1      2      1
 2      2
 3      2      3
```

NUMTODSINTERVAL(*number*, *string*)

Converts *number* to an INTERVAL DAY TO SECOND literal, where *number* is a number or an expression resolving to a number, such as a numeric datatype column. The second argument, *string*, specifies how to interpret *number*: it can be ‘DAY’, ‘HOUR’, ‘MINUTE’, or ‘SECOND’. For example:

```
SELECT NUMTODSINTERVAL(100, 'DAY') FROM DUAL
+000000100 00:00:00.000000000
```

NUMTOYMINTERVAL(*number*, *string*)

Converts *number* to an INTERVAL YEAR TO MONTH literal, where *number* is a number or an expression resolving to a number, such as a numeric datatype column. The second argument, *string*, specifies how to interpret *number*: it can be ‘YEAR’ or ‘MONTH’. For example:

```
SELECT NUMTOYMINTERVAL(100, 'YEAR') FROM DUAL
+000000100-00
```

NVL(*expression1*, *expression2*)

If *expression1* is NULL, *expression2* is returned in place of that NULL value. Otherwise, *expression1* is returned. *expression1* and *expression2* may be any datatype. For example:

```
SELECT NVL(2,29) FROM DUAL -> 2
```

NVL2(*expression1*, *expression2*, *expression3*)

Similar to NVL, except that if *expression1* is not NULL, *expression2* is returned, and if *expression1* is NULL, *expression3* is returned. The expressions may be any datatype except LONG. For example:

```
SELECT NVL2(1,3,5) FROM DUAL -> 3
```

ORA_HASH(*expression*[, *buckets*[, *seed*]])

Computes a hash value from *expression* and returns a bucket number based on the computed hash value. The optional *buckets* argument is the maximum bucket number to use, which is one less than the total number of buckets, since the bucket numbering starts at 0. The default for *buckets* is 4,294,967,295. The optional *seed* value is used to seed the hashing function so that multiple results can be produced from the same data by changing only the *seed* value. The *seed* value defaults to 0. This example pseudorandomly assigns all numbers to one of two buckets and returns those assigned to the first bucket, which will be a sample of roughly half of the values:

```
SELECT C1 FROM FIVE_NUMS WHERE
  ORA_HASH( C1, 1, TO_CHAR( SYSTIMESTAMP, 'SSSS.FF' ) ) = 0
  1
  5
```

PATH(*number*)

Returns the path specified by the *UNDER_PATH* condition with the correlation variable *number* in an XML query. See the Oracle SQL Reference for more information.

POWERMULTISET(*nested_table*)

POWERMULTISET_BY_CARDINALITY(*nested_table*, *cardinality*)

Return a nested table of nested tables of all nonempty subsets of the input nested table in the *nested_table* parameter. *POWERMULTISET_BY_CARDINALITY* has an additional parameter that can be used to limit the subsets returned to a specified minimum cardinality. For more information, see the Oracle SQL Reference.

**PREDICTION(), PREDICTION_BOUNDS(), PREDICTION_COST(),
PREDICTION_DETAILS(), PREDICTION_PROBABILITY(),
PREDICTION_SET()**

Support Oracle's data mining features. See the documentation for the Oracle Data Mining Java API or the *DBMS_DATA_MINING* package for more details on these functions.

PRESENTNNV(cell_reference, expr1, expr2)

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns *expr1* when *cell_reference* exists and is not NULL; otherwise, it returns *expr2*.

PRESENTV(cell_reference, expr1, expr2)

Relevant only in the inter-row calculations performed within the *MODEL* clause of a *SELECT* statement, this function returns *expr1* when *cell_reference* exists; otherwise, it returns *expr2*.

PREVIOUS(cell_reference)

Relevant only in the inter-row calculations performed within the *ITERATE...* [*UNTIL*] section of a *SELECT*'s *MODEL* clause, this function returns the value held by *cell_reference* at the beginning of the iteration.

RATIO_TO_REPORT(value_exprs) OVER (partitioning)

Computes the ratio of a value in *value_exprs* to the sum of all *value_exprs* with each partition. If *value_exprs* is NULL, the ratio-to-report value is also NULL. See “ANSI SQL Window Functions,” earlier in this chapter, for details on the *partitioning* clause. For example:

```
SELECT c1, RATIO_TO_REPORT(c1) OVER () FROM FIVE_NUMS
1           .066666667
2           .133333333
3           .2
4           .266666667
5           .333333333
```

RAWTOHEX(raw)

Converts a *raw* value to a string (character datatype) of its hexadecimal equivalent. For example:

```
SELECT RAWTOHEX('Hi') FROM DUAL -> 4869
```

RAWTONHEX(raw)

Converts a *raw* value to an NVARCHAR2 (character datatype) of its hexadecimal equivalent.

REF(table_alias)

Takes a table alias associated with a row from an object table or an object view. A special reference value is returned for the object instance that is bound to the variable or row.

REFTOHEX(expression)

Converts *expression* to a character value containing its hexadecimal equivalent.

REGEXP_INSTR(string, pattern[, start_at[, occurrence[, roption[, mparam]]]])

Searches *string* from a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of the regular expression *pattern*, and returns the character position within *string* matching *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *roption* parameter can be 0 or 1 and specifies whether the position returned is the first character matching the pattern or the character after, respectively. The default for *roption* is 0. The *mparam* argument can be used to modify the matching

behavior of the function and can be set to one or more of the following characters:

- 'i' Matching is case insensitive.
- 'c' Matching is case sensitive.
- 'n' The “.” character matches newline characters.
- 'm' Treats the input string as multiple lines; use “^” to match the beginning of a line and “\$” to match the end of a line.

For example:

```
SELECT REGEXP_INSTR('Hello, World!', '([^\n]*!)', 1, 1) FROM DUAL
8
```

`REGEXP_REPLACE(string, pattern[, newstr[, start_at[, occurrence[, mparam]]]])`

Searches *string* from a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of the regular expression *pattern*, and returns the result of replacing all occurrences of *pattern* within *string* with another string, *newstr*. Both the *start_at* and *occurrence* parameters default to 1. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

- 'i' Matching is case-insensitive.
- 'c' Matching is case-sensitive.
- 'n' The “.” character matches newline characters.
- 'm' Treats the input string as multiple lines; use “^” to match the beginning of a line and “\$” to match the end of a line.

For example:

```
SELECT REGEXP_REPLACE('Hello, World!', '([^\n]*!)', 'Reader!') FROM DUAL
'Hello, Reader!'
```

`REGEXP_SUBSTR(string, pattern[, start_at[, occurrence[, mparam]]])`

Searches *string* from a starting position of *start_at* (an integer greater than 0) looking for the specified *occurrence* of the regular expression *pattern*, and returns the substring within *string* matching *pattern*. Both the *start_at* and *occurrence* parameters default to 1. The *mparam* argument can be used to modify the matching behavior of the function and can be set to one or more of the following characters:

- 'i' Matching is case insensitive.
- 'c' Matching is case sensitive.
- 'n' The “.” character matches newline characters.
- 'm' Treats the input string as multiple lines; use “^” to match the beginning of a line and “\$” to match the end of a line.

For example:

```
SELECT REGEXP_SUBSTR('Hello, World!', '([^\n]*!)') FROM DUAL
'World!'
```

REMAINDER(*m*, *n*)

Returns the remainder of *m* divided by *n*. This return value is equivalent to the expression:

```
m-n*ROUND(m/n)
```

The function *MOD* uses *FLOOR* instead of *ROUND*. For example:

```
SELECT REMAINDER(11, 4), MOD(11, 4) FROM DUAL
      -1          3
```

REPLACE(*string*, *search_string*[, *replacement_string*])

Returns *string* with every occurrence of *search_string* replaced with *replacement_string*. For example:

```
SELECT REPLACE('change', 'e', 'ing') FROM DUAL -> 'changing'
```

ROUND(*date*[, *format*])

Returns the *date* rounded to the unit specified by the *format* model. When *format* is omitted, the date is rounded to the nearest day. (For more on valid format specifiers, see the *TO_CHAR* function.) For example:

```
SELECT ROUND(TO_DATE('15-APR-1999'), 'MONTH') FROM DUAL
      01-APR-1999
```

ROUND(*number*[, *decimal*])

Returns *number* rounded to *decimal* places to the right of the decimal point. When *decimal* is omitted, *number* is rounded to an integer. Note that *decimal*, an integer, can be negative to round off digits to the left of the decimal point. For example:

```
SELECT ROUND(12345.6789, 2) FROM DUAL -> 12345.68
```

ROWIDTOCHAR(*rowid*), ROWIDTONCHAR(*rowid*)

ROWIDTOCHAR converts the *rowid* value to an 18-character-long *VARCHAR2* value; *ROWIDTONCHAR* converts *rowid* to an 18-character-long *NVARCHAR2* value. For example:

```
SELECT ROWIDTOCHAR(ROWID) FROM NUMS
ABAsxDAAKAAAAEqAAA
ABAsxDAAKAAAAEqAAB
ABAsxDAAKAAAAEqAAC
ABAsxDAAKAAAAEqAAD
```

RPAD(*string1*, *number*[, *string2*])

Returns *string1*, right-padded to the length *number* with the value of *string2*, repeated as needed. *string2* defaults to a single blank. For example:

```
SELECT RPAD('duck',8,'s') FROM DUAL -> 'duckssss'
```

RTRIM(*string*[, *set*])

Returns *string*, with all the rightmost characters that appear in *set* removed; *set* defaults to a single blank. For example:

```
SELECT RTRIM('  welcome  ', ' ') FROM DUAL -> '  welcome'
```

SCN_TO_TIMESTAMP(*scn*)

Returns the timestamp associated with the system change number (*scn*) argument. For example:

```
SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM NUMS WHERE c1 = 1
      15-APR-04 02.56.05.000000000 PM
```

SESSIONTIMEZONE

Returns the session's time zone offset. For example:

```
SELECT SESSIONTIMEZONE FROM DUAL -> -06:00
```

SET(*nested_table*)

Returns a nested table of distinct elements from the input *nested_table*. For more information, see the Oracle SQL Reference.

SIGN(*number*)

When *number* < 0, returns -1. When *number* = 0, returns 0. When *number* > 0, returns 1. For example:

```
SELECT SIGN(-3.1415926) FROM DUAL -> -1
```

SIN(*number*)

Returns the sine of *number*, where *number* is in radians. For example:

```
SELECT SIN( 0 ) -> 0.000000
```

SINH(*number*)

Returns the hyperbolic sine of *number*. For example:

```
SELECT SINH(180) FROM DUAL -> 7.4469E+77
```

SOUNDEX(*string*)

Returns a character string containing the phonetic representation of *string*. This function allows words that are spelled differently but sound alike in English to be compared for equality. For example:

```
SELECT SOUNDEX('thimble') FROM DUAL -> 'T514'
```

STATS_BINOMIAL_TEST, *STATS_CROSSTAB*, *STATS_F_TEST*, *STATS_KS_TEST*, *STATS_MODE*, *STATS_MW_TEST*, *STATS_ONE_WAY_ANOVA*, *STATS_T_TEST_INDEP*, *STATS_T_TEST_INDEPU*, *STATS_T_TEST_ONE*, *STATS_T_TEST_PAIRED*, *STATS_WSR_TEST*

Oracle provides many sophisticated statistical functions. For further information on the *STATS_** functions, see the Oracle SQL Reference.

STDDEV([DISTINCT | ALL] *expression*) [OVER (*window_clause*)]

Returns a sample standard deviation of a set of numbers shown as *expression*. See “ANSI SQL Window Functions,” earlier in this chapter, for details on the *window_clause*. For example:

```
SELECT STDDEV(col1) FROM NUMS -> 5.71547607
```

STDEV_POP(*expression*) [OVER (*window_clause*)]

Computes the population standard deviation and returns the square root of the population variance. See “ANSI SQL Window Functions,” earlier in this chapter, for details on the *window_clause*. For example:

```
SELECT STDDEV_POP(col1) FROM NUMS -> 4.94974747
```

STDDEV_SAMP(*expression*) [OVER (*window_clause*)]

Computes the cumulative sample standard deviation and returns the square root of the sample variance. See “ANSI SQL Window Functions” earlier in this chapter for details on the *window_clause*. For example:

```
SELECT STDDEV_SAMP(col1) FROM NUMS -> 5.71547607
```

SUBSTR(*string*, *start* [FROM *starting_position*] [FOR *length*])

Refer to the earlier section on *SUBSTRING* under “String Functions and Operators.” For example:

```
SELECT SUBSTR('Hello, World!',8,10) FROM DUAL -> 'World!'
```

SUBSTRB(*extraction_string*[, *length*])

Returns the portion of *string* beginning at the position *start* and continuing for *length* characters. If *length* is omitted, all characters from *start* onward are returned. If *start* is negative, it represents an offset from the right edge of the *string*. For example:

```
SELECT SUBSTR('Hello, World!', 8) FROM DUAL -> World!
```

Use *SUBSTRB* for bytes, *SUBSTRC* for UNICODE complete characters, *SUBSTR2* for UNICODE UCS2 code points, and *SUBSTR4* for UNICODE UCS4 code points.

SYS_CONNECT_BY_PATH(*column*, *char*)

For hierarchical queries, *SYS_CONNECT_BY_PATH* returns the path from the root to the node with the column name specified by the *column* parameter. The *char* parameter specifies the node separator for the return path. For more on Oracle hierarchical queries, refer to the Oracle SQL Reference.

SYS_CONTEXT(*namespace*, *attribute*[, *length*])

Returns the value of the *attribute* associated with the context *namespace*, usable in both SQL and PL/SQL statements. The *length* parameter optionally defines the size of the value returned by the function; it defaults to up to 256 bytes, but you may specify a value of between 1 and 4,000 bytes. For example:

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER') FROM DUAL  
'LOGIN'
```

SYS_DBURIGEN(*column*[, *rowid*][, ...][, 'text()'])

Returns a URL that can be used as a unique reference to the row specified by the *column* parameter. For columns that don't hold unique values, a *rowid* can be used directly after the column it identifies to guarantee that the URL only points to one row. Use the 'text()' option if you want the URL to point to the text within an XML document, instead of the document itself.

SYS_EXTRACT_UTC(*datetime*)

Returns the *datetime* argument converted to a UTC datetime value. For example:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2004-04-15 11:59:59.00 -08:00')  
FROM DUAL  
'15-APR-04 07.59.59.000000000 PM'
```

SYS_GUID()

Generates and returns a globally unique identifier (RAW value) made up of 16 bytes. For example:

```
SELECT SYS_GUID() FROM DUAL -> C0FD3FDC30148EAEE030440A49096A41
```

SYS_TYPEID(*object_value*)

Returns the type ID of the *object_value* parameter.

SYS_XMLAGG(*expr*[, *format*])

Returns a single XML document created by aggregating the XML documents or fragments in the *expr* parameter. The optional *format* parameter can be used to format the XML document. For more information, refer to the Oracle SQL Reference.

SYS_XMLGEN(*expression*[, *format*])

Returns a single XML document created from the *expression*. The optional *format* parameter can be used to format the XML document. For more information, refer to the Oracle SQL Reference.

SYSDATE

Returns the current date and time on the system in which the database is hosted. The value returned is of type *DATE*. For example:

```
SELECT SYSDATE FROM DUAL -> 26-JUN-2003
```

SYSTIMESTAMP

Returns the current date and time on the system in which the database is hosted. The value returned is of type *TIMESTAMP*. For example:

```
SELECT SYSTIMESTAMP FROM DUAL
```

```
26-JUN-2003 11.15.00.000000 PM -06:00
```

TAN(*number*)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) FROM DUAL -> -0.000093
```

TANH(*number*)

Returns the hyperbolic tangent of *number*. For example:

```
SELECT TANH(180) FROM DUAL -> 1
```

TIMESTAMP_TO_SCN(*timestamp_value*)

Returns the approximate system change number (SCN) associated with the *timestamp_value*. The return value is of type *NUMBER*.

TO_BINARY_DOUBLE(*expr*[, *format*[, *nls_parameter*]])

Converts *expr* to a *BINARY_DOUBLE*, optionally in the format specified by the *format* parameter. If *expr* is a character expression, the *format* and *nls_parameter* options have equivalent meanings, as they do in the *TO_CHAR* function. If *expr* is a numeric expression, *format* and *nls_parameter* must be omitted. For example:

```
SELECT c1, TO_BINARY_DOUBLE(c1) FROM NUMS
1           1.0E+000
2           2.0E+000
3           3.0E+000
```

TO_BINARY_FLOAT(*expr*[, *format*[, *nls_parameter*]])

Converts *expr* to a *BINARY_FLOAT*, optionally in the format specified by the *format* parameter. If *expr* is a character expression, the *format* and *nls_parameter* options have equivalent meanings, as they do in the *TO_CHAR* function. If *expr* is a numeric expression, *format* and *nls_parameter* must be omitted. For example:

```
SELECT c1, TO_BINARY_FLOAT(c1) FROM NUMS
1          1.0E+000
2          2.0E+000
3          3.0E+000
```

TO_CHAR(*character_expr*)

Converts *character_expr* to the database character set. For example:

```
SELECT TO_CHAR('Howdy') FROM DUAL
Howdy
```

TO_CHAR(*date* | *interval* [, *format* [, *nls_parameter*]])

Converts *date* or *interval* to a VARCHAR2 in the format specified by *format*.

When *format* is omitted, *date* is converted to the default date format. The *nls_parameter* option offers additional control over formatting. For example:

```
SELECT TO_CHAR(TO_DATE('15-APR-1999') , 'MON-DD-YYYY') FROM DUAL
APR-15-1999
```

Table 4-11 lists the available specifiers for *format* and their meanings.

Table 4-11. Oracle format specifiers

Format specifier	Meaning
AD or A.D.	AD indicator
AM or A.M.	Meridian indicator
BC or B.C.	BC indicator
D	Day of week (1–7)
DAY	Name of day
DD	Day of month (1–31)
DDD	Day of year (1–366)
DL	Long date format
DS	Short date format
DY	Abbreviated name of day
FF	Fractional seconds; to specify the precision, include a number (1–9) after the FF specifier
HH or HH12	Hour (1–12)
HH24	Hour (0–23)
J	Julian day; the number of days since January 1, 4713 BC
MI	Minute (0–59)
MM	Month (01–12)
MON	Abbreviated name of month
RM	Roman numeral month (I–XII)
SS	Second (0–59)
SSSS	Seconds past midnight (0–86,399)
YYYY	Four-digit year; BC dates are prefixed with a minus sign
TS	Short time format
TZD	Daylight savings information (example: PST versus PDT)
TZH	Time zone hour

Table 4-11. Oracle format specifiers (continued)

Format specifier	Meaning
TZM	Time zone minute
TZR	Time zone region
X	Local radix character
Y, YY, or YYY	One-, two-, or three-digit year
Y,YYY	Year with comma
YYYY	Four-digit year

TO_CHAR(*number*[, *format*[, *nls_parameter*]])

Converts *number* to a *VARCHAR2* in the *format* specified. When *format* is omitted, *number* is converted to a string long enough to hold the number. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_CHAR(123.45, '$999.99') FROM DUAL -> $123.45
```

TO_CLOB(*expr*)

Converts the character expression given by *expr* to the *CLOB* datatype. For example:

```
SELECT LENGTH(TO_CLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_DATE(*string*[, *format*[, *nls_parameter*]])

Converts *string* (a *CHAR* or *VARCHAR2*) to the *DATE* datatype. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_DATE('15/04/1999', 'DD/MM/YYYY') FROM DUAL
15-APR-1999
```

TO_DSINTERVAL(*string*[, *nls_parameter*])

Converts the character expression given by *string* to the *INTERVAL DAY TO SECOND* datatype. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT CURRENT_DATE, CURRENT_DATE-TO_DSINTERVAL('14 00:00:00')
FROM DUAL
15-APR-2003      01-APR-2003
```

TO_LOB(*long_column*)

Converts *LONG* or *LONG RAW* values in the column *long_column* to *LOB* values. Usable only by *LONG* or *LONG RAW* expressions and only in the *SELECT* list of a subquery in an *INSERT* statement.

TO_MULTI_BYTE(*string*)

Returns *string* with all of its single-byte characters converted to their corresponding multibyte characters.

TO_NCHAR(*expr*[, *format*[, *nls_parameter*]])

Synonymous with the *TO_CHAR* function, except the return datatype is *NCHAR*. For more on valid format specifiers, see the *TO_CHAR* function.

TO_NCLOB(*expr*)

Converts the character expression given by *expr* to the NCLOB datatype. For example:

```
SELECT LENGTH(TO_NCLOB('I am a SQL nut!')) FROM DUAL -> 15
```

TO_NUMBER(*string*[, *format*[, *nls_parameter*]])

Converts a numeric *string* (a CHAR or VARCHAR2) to the NUMBER datatype, optionally in the format specified by the *format* parameter. The *nls_parameter* option offers additional control over formatting options. For example:

```
SELECT TO_NUMBER('12345') FROM DUAL -> 12345
```

TO_SINGLE_BYTE(*string*)

Returns *string* with all of its multibyte characters converted to their corresponding single-byte characters.

TO_TIMESTAMP(*string*[, *format*[, *nls_parameter*]])

Converts the character expression provided by *string* to the TIMESTAMP datatype, optionally in the format specified by the *format* parameter. The *nls_parameter* option offers additional control over formatting options. (For more on valid format specifiers, see the TO_CHAR function.) For example:

```
SELECT TO_TIMESTAMP(CURRENT_DATE) FROM DUAL -> 04-MAY-04 12.00.00 AM
```

TO_TIMESTAMP_TZ(*string*[, *format*[, *nls_parameter*]])

Converts the character expression provided by *string* to the TIMESTAMP WITH TIME ZONE datatype, optionally in the format specified by the *format* parameter. The *nls_parameter* option offers additional control over formatting options. (For more on valid format specifiers, see the TO_CHAR function.) For example:

```
SELECT TO_TIMESTAMP_TZ('15-04-2006', 'DD-MM-YYYY') FROM DUAL  
15-APR-06 12.00.00.000000000 AM -07:00
```

TO_YMINTERVAL(*string*)

Converts the character expression provided by *string* to the INTERVAL YEAR TO MONTH datatype. For example:

```
SELECT TO_DATE('29-FEB-2000')+TO_YMINTERVAL('04-00') FROM DUAL  
29-FEB-04
```

TRANSLATE(*char_value*, *from_text*, *to_text*)

Returns *char_value* with each occurrence of a character in *from_text* replaced by its corresponding character in *to_text*. For example:

```
SELECT TRANSLATE('foobar', 'fa', 'bu') FROM DUAL -> 'boobur'
```

TRANSLATE(*text* USING [CHAR_CS | NCHAR_CS])

Converts *text* into the character set specified. Use CHAR_CS to convert *text* to the CHAR datatype or NCHAR_CS to convert *text* to the NCHAR datatype. For example:

```
SELECT TRANSLATE(N'foobar' USING CHAR_CS) FROM DUAL  
'foobar'
```

TREAT(*expr* AS [REF] [*schema.*]*type*)

Converts *expr* from its declared type to the type specified by the *type* parameter. For more information on the usage of this function, please look to the SQL Reference for the Oracle Database.

TRUNC(*base*[, *number*])

Returns *base* truncated to *number* decimal places. When *number* is omitted, *base* is truncated to an integer. *number* can be negative to truncate (make zero) digits to the left of the decimal point. For example:

```
SELECT TRUNC('123.456', 2) FROM DUAL -> 123.45
```

TRUNC(*date*[, *format*])

Returns *date* truncated to the unit specified by *format*. When *format* is omitted, *date* is truncated to the nearest whole day. (For more on valid format specifiers, see the TO_CHAR function.) For example:

```
SELECT TRUNC(TO_DATE('15/04/1999', 'MM/DD/YYYY'), 'YYYY') FROM DUAL
1999
```

TZ_OFFSET(*expr* | SESSIONTIMEZONE | DBTIMEZONE)

Returns the time zone offset corresponding to the argument. The character expression *expr* can either be the name of the time zone or a time zone offset. The SESSIONTIMEZONE and DBTIMEZONE arguments provide the time zone for the session or database, respectively. For example:

```
SELECT TZ_OFFSET('+08:00'), TZ_OFFSET(SESSIONTIMEZONE) FROM DUAL
+08:00
-07:00
```

UID

Returns an integer that uniquely identifies the currently logged-on session user. No parameters are needed. For example:

```
SELECT UID FROM DUAL -> 47
```

UNISTR(*string*)

Converts *string* to the NCHAR datatype, while converting any UNICODE-encoded values within *string*. For example:

```
SELECT UNISTR('El Ni\00F1o') FROM DUAL -> 'El Niño'
```

UPDATEXML(*instance*, *xpath*, *expr*[, *namespace*])

Updates the values held by nodes within *instance* to the new value in *expr*. Only those nodes returned by the XPath query contained in the *xpath* parameter are updated. The optional *namespace* parameter specifies the XML namespace in the query. For more information on XML queries, refer to the Oracle SQL Reference. For example:

```
SELECT UPDATEXML( XMLTYPE('<foo><bar>Hello, World!</bar></foo>'),
                   '/foo/bar', '<bar>Bye, World!</bar>' ) from DUAL
Bye, World!
```

USERENV(*option*)

Returns information about the current session in the VARCHAR2 datatype. This function has been deprecated and is only provided for backward compatibility. USERENV is a synonym for SYS_CONTEXT('USER_ENV', *option*).

Refer to the `SYS_CONTEXT` function with the `USERENV` namespace for current functionality. For example:

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL
  'AMERICAN_AMERICA.AL32UTF8'
```

`VALUE(table_alias)`

Takes a `table_alias` associated with a row in an object table and returns the object instance stored within the object table for that row.

`VAR_POP(expression) [OVER (window_clause)]`

Returns the population variance of the set of numbers represented by `expression` after discarding the NULLs in the set. See “ANSI SQL Window Functions,” earlier in this chapter, for details on the `window_clause`. For example:

```
SELECT VAR_POP(col1) FROM NUMS -> 24.5
```

`VAR_SAMP(expression) [OVER (window_clause)]`

Returns the sample variance of the set of numbers represented by `expression` after discarding the NULLs in the set. See “ANSI SQL Window Functions,” earlier in this chapter, for details on the `window_clause`. For example:

```
SELECT VAR_SAMP(col1) FROM NUMS -> 32.6666667
```

`VARIANCE([DISTINCT] expression) [OVER (window_clause)]`

Returns the variance of `expression`, calculated as follows: 0 if the number of rows in `expression` = 1, and `VAR_SAMP` if the number of rows in `expression` > 1. See “ANSI SQL Window Functions,” earlier in this chapter, for details on the `window_clause`. For example:

```
SELECT VARIANCE(col1) FROM NUMS -> 32.6666667
```

`VSIZE(expression)`

Returns the number of bytes in the internal representation of `expression`. When `expression` is `NULL`, returns `NULL`. For example:

```
SELECT vsize(1) FROM DUAL -> 2
```

`XMLAGG(instance[, order_by])`

Aggregate function that returns an XML document from a table of XML fragments contained in `instance`. The optional `order_by` clause allows the XML fragments in the result to be ordered. For more information on XML queries, refer to the Oracle SQL Reference.

`XMLCAST(value AS datatype)`

Casts the `value` to the type `datatype` and returns the result. For more information on XML queries, refer to the Oracle SQL Reference.

`XMLCDATA(value)`

Returns `value` as an XML CDATA section. For more information on XML queries, refer to the Oracle SQL Reference.

`XMLCOLATTVAL(expr [AS alias][, ...])`

Returns an XML fragment from the `expr` arguments. The optional `AS` clause can be used to change the value of the name attribute. For more information on XML queries, refer to the Oracle SQL Reference.

XMLCONCAT(*instance*[, ...])

Returns an XML instance that is the union of all XML *instance* parameters. For more information on XML queries, refer to the Oracle SQL Reference.

XMDIFF(*xml1*, *xml2*[, *hashLevel*, *flags*])

Returns an Xdiff schema that's the result of diffing *xml1* and *xml2*. For more information on XML queries, refer to the Oracle SQL Reference.

XMLELEMENT([*NAME*] *name*[, XMLATTRIBUTES(*expr* [AS *alias*][, ...])][, *value*[, ...]])

Returns an XMLELEMENT with the name specified in the *name* parameter and the attributes specified in the optional XMLATTRIBUTES clause. The *value* parameters provide the values of the XMLELEMENT result. For more information on XML queries, refer to the Oracle SQL Reference.

XMLEXISTS(*xquery*[, *passing_clause*])

Returns TRUE if the XQuery specified in *xquery* returns a nonempty XQuery result, and FALSE otherwise. For more information on XML queries, refer to the Oracle SQL Reference.

XMLFOREST(*value* [AS *alias*][, ...])

Returns an XML fragment constructed from the values provided in the *value* parameter list. The optional AS clause can be used to change the enclosing tag name. For more information on XML queries, refer to the Oracle SQL Reference.

XMLPARSE([DOCUMENT | CONTENT] *value* [WELLFORMED])

Returns an XML instance constructed as a result of parsing the XML document in *value*. For more information on XML queries, refer to the Oracle SQL Reference.

XMLPATCH(*xml*, *xdiff*)

Returns an XML instance as a result of patching the XML document in *xml* with the Xdiff found in *xdiff*. For more information on XML queries, refer to the Oracle SQL Reference.

XMLPI()

Generates an XML processing instruction, commonly used to provide application instructions associated with all or part of an XML document. For more information on XML queries, refer to the Oracle SQL Reference.

XMLQUERY(*query*)

Returns the results of executing the XMLQuery found in *query*. For more information on XML queries, refer to the Oracle SQL Reference.

XMLROOT(*value*, VERSION [*version* | NO VALUE][, STANDALONE {YES | NO | NO VALUE})]

Returns a new XML document using *value* as the body and the version information for the XML document prolog. For more information on XML queries, refer to the Oracle SQL Reference.

`XMLSEQUENCE(instance)`

Returns an array of XML fragments constructed from the top-level nodes in the XML instance provided by the *instance* parameter. For more information on XML queries, refer to the Oracle SQL Reference.

`XMLSERIALIZE(...)`

Returns an XML expression serialized to a string. For more information on XML queries, refer to the Oracle SQL Reference.

`XMLTABLE(...)`

Maps the result of an XQuery into a two-dimensional, relational view. For more information on XML queries, refer to the Oracle SQL Reference.

`XMLTRANSFORM(instance, stylesheet)`

Returns the result of applying the XSL stylesheet in the *stylesheet* parameter to the XML document contained in *instance*. For more information on XML queries, refer to the Oracle SQL Reference.

PostgreSQL-Supported Functions

This section lists the functions specific to PostgreSQL, with examples and corresponding results.

`ABSTIME(timestamp)`

Converts the *timestamp* value to the `ABSTIME` type. This function is provided for backward compatibility and may be removed in future versions. For example:

```
SELECT ABSTIME(CURRENT_TIMESTAMP) -> 2003-06-24 00:19:17-07
```

`ACOS(number)`

Returns the arccosine of *number*, ranging from -1 to 1 . The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

`AGE(timestamp)`

Has the same meaning as `AGE(CURRENT_DATE, timestamp)`.

`AGE(timestamp, timestamp)`

Returns the time between the two *timestamp* values. For example:

```
SELECT AGE( '2003-12-31', CURRENT_TIMESTAMP )
6 mons 7 days 00:34:41.658325
```

`AREA(object)`

Returns the area of an item. For example:

```
SELECT AREA( BOX '((0,0),(1,1))' ) -> 1
```

`ARRAY_APPEND(array, element)`

Returns the result of appending *element* to *array*. For example:

```
SELECT ARRAY_APPEND(ARRAY[1,2], 3) -> {1,2,3}
```

ARRAY_CAT(*array1, array2*)

Returns the result of appending *array2* to *array1*. For example:

```
SELECT ARRAY_CAT(ARRAY[1,2], ARRAY[3,4]) -> {1,2,3,4}
```

ARRAY_DIMS(*array*)

Returns the dimensions of *array*. For example:

```
SELECT ARRAY_DIMS(ARRAY[1,2]) -> '[1:2]'
```

ARRAY_LOWER(*array, i*)

Returns the lower bound of the dimension *i* of *array*. For example:

```
SELECT ARRAY_LOWER(ARRAY[1,2], 1) -> 1
```

ARRAY-prepend(*i, array*)

Returns the result of prepending *i* to *array*. For example:

```
SELECT ARRAY-prepend(0, ARRAY[1,2]) -> {0,1,2}
```

ARRAY_TO_STRING(*array, delimiter*)

Returns a string that is constructed by concatenating the elements of *array*, with *delimiter* used as a delimiter between each element. For example:

```
SELECT ARRAY_TO_STRING(ARRAY[1,2,3], ';') -> '1;2;3'
```

ARRAY_UPPER(*array, i*)

Returns the upper bound of the dimension *i* of *array*. For example:

```
SELECT ARRAY_UPPER(ARRAY[1,2], 1) -> 2
```

ASCII(*text*)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(*number*)

Returns the arcsine of *number*, ranging from -1 to 1 . The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(*number*)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATAN2(*float1, float2*)

Returns the arctangent of the two *float* values. *ATAN2(x, y)* is similar to *ATAN(y/x)*, with the exception that the signs of *x* and *y* are used to determine the quadrant of the result. For example:

```
SELECT ATAN2( 3.1415926, 0 ) -> 1.5707963267949
```

BIT_AND(*expression*)**BIT_OR(*expression*)**

Returns the bitwise AND or OR of every non-NULL value in *expression*. For example:

```
SELECT BIT_AND( column1 ) FROM table
```

0

`BOOL_AND(expression)`

`BOOL_OR(expression)`

Returns the logical AND or OR of every non-NULL value in *expression*. For example:

```
SELECT BOOL_OR( column1 ) FROM table
true
```

`BOX(box, box)`

Returns a *BOX* created by the intersection of the two boxes. If the two boxes do not intersect, the return value is NULL. For example:

```
SELECT BOX( BOX '((-1,-1),(1,1))', BOX '((0,0),(1,1))' )
(1,1),(0,0)
```

`BOX(circle)`

Returns a *BOX* with vertices that intersect *circle* so that the box is the maximum size that can be fully contained within *circle*. For example:

```
SELECT BOX(CIRCLE '((0,0),2.0)')
(1.41421356237309,1.41421356237309),
(-1.41421356237309,-1.41421356237309)
```

`BOX(point, point)`

Returns a *BOX* with the two *point* arguments as opposite corners. For example:

```
SELECT BOX(POINT(0,0), POINT(1,1)) -> (1,1),(0,0)
```

`BOX(polygon)`

Converts *polygon* to a *BOX*. For example:

```
SELECT BOX(POLYGON '((0,0),(1,1),(2,0))') -> (2,1),(0,0)
```

`BROADCAST/inet`

Constructs a broadcast address as text. For example:

```
SELECT BROADCAST('192.168.1.5/24') -> '192.168.1.255/24'
```

`BTRIM(s, c)`

Return *s* without the characters found in *c*. For example:

```
SELECT BTRIM('<<<trim_me>>>', '><') -> 'trim_me'
```

`CBRT(float8)`

Returns the cube root of *float8*. For example:

```
SELECT CBRT(8) -> 2
```

`CENTER(object)`

Returns a *POINT* object to the center of the argument. For example:

```
SELECT CENTER( CIRCLE '((0,0), 2.0)' ) -> (0,0)
```

`CHAR(text)`

Converts *text* to the *CHAR* type.

`CHAR_LENGTH(string) or CHARACTER_LENGTH(string)`

Returns the length of *string* in characters.

`CIRCLE(box)`

Returns a *CIRCLE* contained within *box*. For example:

```
SELECT CIRCLE(BOX '((0,0),(1,1))') -> <(0.5,0.5),0.707106781186548>
```

CIRCLE(*point*, *float8*)

Converts *point* to a *CIRCLE* with *float8* for the radius. For example:

```
SELECT CIRCLE(POINT '(0,0)', 2.0) -> <(0,0),2>
```

CLOCK_TIMESTAMP()

Returns the current date and time, which can change within a single SQL statement execution.

COALESCE(*list*)

Returns the first non-NULL value in *list*. For example:

```
SELECT COALESCE(NULL,1,2,3,NULL) -> 1
```

COS(*number*)

Returns the cosine of *number* as an angle expressed in radians. For example:

```
SELECT COS(0) -> 1.000000
```

COT(*number*)

Returns the cotangent of *number*. For example:

```
SELECT COT( 3.1415 ) -> -10792.88993953
```

CURRVAL(*s*)

Returns the current value of the sequence named *s*. For example:

```
SELECT CURRVAL( 'myseq' ) -> 99
```

DATE_PART(*text*, *value*)

Equivalent to *EXTRACT(*text*, *value*)*; for more details on the usage of *EXTRACT*, see the section on *EXTRACT* earlier in “ANSI SQL Scalar Functions.”

DATE_TRUNC(*precision*, *timestamp*)

Truncates *timestamp* to the specified *precision*. For example:

```
SELECT DATE_TRUNC('hour', TIMESTAMP '2003-04-15 23:58:30')
2003-04-15 23:00:00
```

DECODE(*s*, *type*)

Decodes an encoded string *s*. The *type* can be ‘base64’, ‘hex’, or ‘escape’. For example:

```
SELECT DECODE( ENCODE('darjeeling', 'base64'), 'base64' ) -> 'darjeeling'
```

DEGREES(*float8*)

Converts radians to degrees. For example:

```
SELECT DEGREES( 3.1415926 ) -> 179.999996929531
```

DIAMETER(*circle*)

Returns the diameter of *circle*. For example:

```
SELECT DIAMETER(CIRCLE(POINT '(0,0)', 2.0)) -> 4
```

ENCODE(*s*, *type*)

Encodes a string *s*. The *type* can be ‘base64’, ‘hex’, or ‘escape’. For example:

```
SELECT DECODE( ENCODE('darjeeling', 'base64'), 'base64' ) -> 'darjeeling'
```

EVERY(*expression*)

Synonym for *BOOL_AND(*expression*)*.

FLOAT(*int*)

Converts *int* to a floating point.

FLOAT4(*int*)

Converts *int* to a floating point.

HEIGHT(*box*)

Returns the vertical size of *box*. For example:

```
SELECT HEIGHT(BOX '((0,0),(1,1))') -> 1
```

HOST(*inet*)

Extracts the host address as text. For example:

```
SELECT HOST('192.168.1.5/24') -> '192.168.1.5'
```

INITCAP(*text*)

Converts the first letter of each word to uppercase. For example:

```
SELECT INITCAP( 'my name is inigo montoya.' )  
'My Name Is Inigo Montoya.'
```

INTEGER(*float*)

Converts a floating point to integer.

INTERVAL(*reltime*)

Converts *reltime* to an *INTERVAL*.

ISCLOSED(*path*)

Returns 't' if *path* is closed, and 'f' if it's open. For example:

```
SELECT ISCLOSED(PATH '((0,0),(1,1),(2,0))') -> 't'  
SELECT ISCLOSED(PATH '[(0,0),(1,1),(2,0)]') -> 'f'
```

ISFINITE(*interval*)

Returns 'f' if *interval* is open, and 't' otherwise. For example:

```
SELECT ISFINITE(INTERVAL '4 hours') -> 't'
```

ISFINITE(*timestamp*)

Returns 'f' if *timestamp* is either invalid or infinite, and 't' otherwise. For example:

```
SELECT ISFINITE(TIMESTAMP '2001-02-16 21:28:30') -> 't'
```

ISOPEN(*path*)

Returns an open *path*. For example:

```
SELECT ISOPEN(PATH '((0,0),(1,1),(2,0))') -> 'f'  
SELECT ISOPEN(PATH '[(0,0),(1,1),(2,0)]') -> 't'
```

JUSTIFY_DAYS(*interval*)

Returns the number of 30-day time periods within *interval*. For example:

```
SELECT JUSTIFY_DAYS(INTERVAL '90 days') -> 3
```

JUSTIFY_HOURS(*interval*)

Returns the number of 24-hour time periods within *interval*. For example:

```
SELECT JUSTIFY_HOURS(INTERVAL '48 hours') -> 2
```

JUSTIFY_INTERVAL(*interval*)

Returns the number of 30-day time periods and remaining 24-hour time periods within *interval*. For example:

```
SELECT JUSTIFY_HOURS(INTERVAL '2 mon - 12 hour') -> '59 days 12:00:00'
```

LENGTH(*object*)

Returns the length of *object*. For example:

```
SELECT LENGTH('Howdy!') -> 6
```

```
SELECT LENGTH(PATH '((-1,0),(1,0))') -> 4
```

LOG(*float8*[, *b*])

Returns a base-10 logarithm, unless *b* is specified, in which case it returns the logarithm of base *base*. For example:

```
SELECT LOG( 100 ) -> 2
```

LPAD(*exp1*, *int*, *exp2*)

Returns the string 'text', left-padded to the specified (*int*) length. For example:

```
SELECT LPAD('Duck', 10, 's') -> 'ssssssDuck'
```

LSEG(*box*)

Converts a *box* diagonal to a line segment. For example:

```
SELECT LSEG(BOX '((-1,0),(1,0))') -> [(1,0),(-1,0)]
```

LSEG(*point*, *point*)

Converts *points* to a line segment. For example:

```
SELECT LSEG(POINT '(-1,0)', POINT '(1,0)') -> [(-1,0),(1,0)]
```

LTRIM(*text*)

Returns *text* with all leading whitespace removed. For example:

```
SELECT LTRIM(' Howdy! ') -> 'Howdy! '
```

MASKLEN(*cidr*)

Returns the netmask length of *cidr*. For example:

```
SELECT MASKLEN('192.168.1.5/24') -> 24
```

MD5(*s*)

Returns the MD5 hash of *s*.

NETMASK(*inet*)

Returns the netmask for *inet*. For example:

```
SELECT NETMASK('192.168.1.5/24') -> '255.255.255.0'
```

NETWORK(*inet*)

Returns the network part of *inet*. For example:

```
SELECT NETWORK('192.168.1.5/24') -> '192.168.1.0/24'
```

NEXTVAL(*s*)

Returns the next number in the sequence named *s*. For example:

```
SELECT NEXTVAL('myseq') -> 100
```

NPOINTS(*object*)

Returns the number of points in *object*. For example:

```
SELECT NPOINTS(POLYGON '((1,1),(0,0))') -> 2
```

NULLIF(*input, value*)

Returns NULL if *input* = *value*; otherwise, returns *input*. For example:

```
SELECT NULLIF( 5, 6 ), NULLIF( 5, 5 )
      5      NULL
```

PATH(*polygon*)

Converts *polygon* to a path. For example:

```
SELECT PATH( '((0,0),(1,1),(2,0))' )
      ((0,0),(1,1),(2,0))
```

PCLOSE(*path*)

Converts *path* to a closed path. For example:

```
SELECT PCLOSE(PATH '[ (0,0),(1,1),(2,0)]')
      ((0,0),(1,1),(2,0))
```

PI()

Returns the constant π .

POLYGON(*path*)

Converts *path* to a POLYGON. For example:

```
SELECT POLYGON(PATH '((0,0),(1,1),(2,0))')
      ((0,0),(1,1),(2,0))
```

POINT(*circle*)

Returns the center of *circle*. For example:

```
SELECT POINT( CIRCLE '((0,0), 2.0)' ) -> (0,0)
```

POINT(*lseg1, lseg2*)

Returns the intersection of two line segments. For example:

```
SELECT POINT(LSEG '((-1,0),(1,0))', LSEG '((-2,-2),(2,2))')
      (0,0)
```

POINT(*polygon*)

Returns the center point of *polygon*. For example:

```
SELECT POINT(POLYGON '((0,0),(1,1),(2,0))')
      (1,0.333333333333333)
```

POLYGON(*box*)

Returns a four-point polygon. For example:

```
SELECT POLYGON(BOX '((0,0),(1,1))')
      ((0,0),(0,1),(1,1),(1,0))
```

POLYGON(*circle*)

Synonym for *POLYGON*(12, *circle*).

POLYGON(*npts, circle*)

Returns an approximation of *circle* as a polygon with *npts* vertices. For example:

```
SELECT POLYGON(6, CIRCLE '((0,0),2.0)')
      ((-2,0),
      (-0.99999999994107,1.73205080757228),
      (1.00000000001179,1.73205080756207),
```

```
(2,-2.04136478690279e-11),
(0.999999999976428,-1.73205080758249),
(-1.00000000002946,-1.73205080755187))
```

POOPEN(path)

Converts *path* to an open path. For example:

```
SELECT POOPEN(PATH '((0,0),(1,1),(2,0))')
[(0,0),(1,1),(2,0)]
```

POW(number, exponent)

Raises *number* to the specified *exponent*. For example:

```
SELECT POW( 2, 3 ) -> 8
```

QUOTE_IDENT(s)

Returns *s* properly escaped so that it can be used as an identifier in a SQL statement. For example:

```
SELECT QUOTE_IDENT( 'tea' ) -> '"tea"'
```

QUOTE_LITERAL(s)

Returns *s* properly escaped so that it can be used as a string literal in a SQL statement. For example:

```
SELECT QUOTE_LITERAL( 'you're here' ) -> 'you"re here'
```

RADIANS(float8)

Converts degrees to radians. For example:

```
SELECT RADIANS( 180 ) -> 3.14159265358979
```

RADIUS(circle)

Returns the radius of *circle*. For example:

```
SELECT RADIUS( CIRCLE '((0,0), 2.0)' ) -> (0,0)
```

RANDOM()

Returns a random value between 0.0 and 1.0. For example:

```
SELECT RANDOM() -> 0.785398163397448
```

REGEXP_MATCHES(s, pattern[, flags])

Returns substrings within *s* matching the regular expression *pattern*. The optional *flags* argument can select non-default processing by the regular expression matching engine. The most common options for flags are 'i' for case-insensitive matching, 'n' for newline-sensitive matching, and 'g' for global matching. For example:

```
SELECT REGEXP_MATCHES('catfish, cowfish', 'c..') -> 'cat'
```

REGEXP_REPLACE(s, pattern, replacement[, flags])

Returns *s* with all substrings matching the regular expression *pattern* replaced with the string found in *replacement*. The optional *flags* argument can select non-default processing by the regular expression matching engine. The most common options for flags are 'i' for case-insensitive matching, 'n' for newline-sensitive matching, and 'g' for global matching. For example:

```
SELECT REGEXP_REPLACE('abcabcabc', 'bc', 'nt') -> 'antabcabc'
```

```
SELECT REGEXP_REPLACE('abcabcabc', 'bc', 'nt', 'g') -> 'antantant'
```

`REGEXP_SPLIT_TO_ARRAY(s, pattern[, flags])`

Returns *s* split into an array using the regular expression *pattern* as the delimiter for array elements. The optional *flags* argument can select non-default processing by the regular expression matching engine. The most common options for flags are 'i' for case-insensitive matching, 'n' for newline-sensitive matching, and 'g' for global matching. For example:

```
SELECT REGEXP_SPLIT_TO_ARRAY('a b c', E'\\s+')
{a,b,c}
```

`REGEXP_SPLIT_TO_TABLE(s, pattern[, flags])`

Returns *s* split into a table using the regular expression *pattern* as the delimiter for rows. The optional *flags* argument can select non-default processing by the regular expression matching engine. The most common options for flags are 'i' for case-insensitive matching, 'n' for newline-sensitive matching, and 'g' for global matching. For example:

```
SELECT REGEXP_SPLIT_TO_TABLE('a b c', E'\\s+')
'a'
'b'
'c'
```

`RELTIME(interval)`

Converts *interval* to a *RELTIME*. Provided for backward compatibility and may be removed in a future release.

`ROUND(number[, p])`

Rounds *number* to *p* decimal places. The optional *p* argument defaults to 0 for classic rounding of integers. For example:

```
SELECT ROUND( 5.5 ) -> 6
SELECT ROUND( 5.5555, 2 ) -> 5.56
```

`RPAD(text, length, char)`

Pads *text* to the specified *length* using *char*. For example:

```
SELECT RPAD('Duck', 10, 's') -> 'Duckssssss'
```

`RTRIM(text)`

Returns *text* with all trailing whitespace removed. For example:

```
SELECT RTRIM(' St. Lucia ') -> ' St. Lucia'
```

`SET_MASKLEN(inet, size)`

Sets the netmask length for *inet* to *size*. For example:

```
SELECT SET_MASKLEN('192.168.1.5/24',16)
      '192.168.1.5/16'
```

`SETSEED(i)`

Seeds the random number generator with *i*.

`SETVAL(s, i)`

Sets the next number in the sequence named *s* to *i*. For example:

```
SETVAL('myseq', 0)
```

`SIGN(number)`

Returns the sign of *number*. For example:

```
SELECT SIGN( -69 ), SIGN(69)
      -1,1
```

SIN(*number*)

Returns the sine of *number*, where *number* is in radians. For example:

```
SELECT SIN( 0 ) -> 0.000000
```

STATEMENT_TIMESTAMP()

Returns the date and time at which the SQL statement began execution.

STRING_TO_ARRAY(*str1*, *delimiter*)

Returns an array that is constructed by extracting the elements of *str1* using *delimiter* as a delimiter between each element. For example:

```
SELECT STRING_TO_ARRAY('1;2;3', ',') -> {1,2,3}
```

SUBSTRING(*string* [FROM *start*] [FOR *bytes*]), SUBSTR(*string*, *start*[, *bytes*])

Extracts a substring of length *bytes* from *string* starting at the character position *start*. If *bytes* is omitted, the length returned is the remainder of the *string* from the *start* position. For example:

```
SELECT SUBSTRING( 'Inigo Montoya' FROM 7 FOR 4 ) -> 'Mont'
```

TAN(*number*)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TEXT(*char*)

Converts *char* to the *TEXT* type.

TIMEOFDAY()

Returns the current timestamp (using *CLOCK_TIMESTAMP*) as a string.

TIMESTAMP(*date*[, *time*])

Converts *date* to a timestamp.

TO_CHAR(*expression*, *text*)

Converts *expression* to a string. For example:

```
SELECT TO_CHAR(NUMERIC '-125.8', '999D99S') -> 125.80-
```

```
SELECT TO_CHAR (interval '15h 2m 12s','HH24:MI:SS') -> 15:02:12
```

TO_DATE(*string*, *format*)

Converts *string* to a date using the second argument for the input format.

For example:

```
SELECT TO_DATE('05 Dec 2000', 'DD Mon YYYY') -> 2000-12-05
```

Table 4-12 lists the available specifiers for *format* and their meanings.

Table 4-12. PostgreSQL format specifiers

Format specifier	Meaning
AD or A.D.	AD indicator
AM or A.M.	Meridian indicator
BC or B.C.	BC indicator
CC	Two-digit century
D	Day of week (1–7)
DAY	Full uppercase day name

Table 4-12. PostgreSQL format specifiers (continued)

Format specifier	Meaning
<i>Day</i>	Full camel case day name
<i>day</i>	Full lowercase day name
<i>DD</i>	Day of month (01–31)
<i>DDD</i>	Day of year (001–366)
<i>DY</i>	Abbreviated day name
<i>Dy</i>	Abbreviated camel case day name
<i>dy</i>	Abbreviated lowercase day name
<i>HH</i> or <i>HH12</i>	Hour (01–12)
<i>HH24</i>	Hour (00–23)
<i>I</i>	Last digits of ISO year
<i>IW</i>	ISO week number of year
<i>IY</i> , <i>IYY</i> , <i>IYYY</i>	Last 2 digits, 3 digits, or all digits of ISO year, respectively
<i>J</i>	Julian day number (days since January 1, 4713 BC)
<i>MI</i>	Minute (00–59)
<i>MM</i>	Month (01–12)
<i>MON</i>	Abbreviated month name
<i>Mon</i>	Abbreviated camel case month name
<i>mon</i>	Abbreviated lowercase month name
<i>MONTH</i>	Full uppercase month name
<i>Month</i>	Full camel case month name
<i>month</i>	Full lowercase month name
<i>MS</i>	Milliseconds (000–999)
<i>PM</i> or <i>P.M.</i>	Meridian indicator
<i>Q</i>	Quarter of the year
<i>RM</i>	Roman Numeral month (I–XII)
<i>rm</i>	Lowercase Roman Numeral month (i–xii)
<i>SS</i>	Second (00–59)
<i>SSSS</i>	Seconds past midnight (0–86,399)
<i>SYY</i>	Four-digit year; BC dates are prefixed with a minus sign
<i>TS</i>	Short time format
<i>TZ</i>	Uppercase time zone name
<i>tz</i>	Lowercase time zone name
<i>US</i>	Microseconds (000000–999999)
<i>W</i>	Week of month (1–5)
<i>WW</i>	Week of year (1–53)
<i>Y</i> , <i>YY</i> , <i>YYY</i> , <i>YYYY</i>	One-, two-, three-, or four-digit year, respectively
<i>Y,YYY</i>	Year with comma

TO_NUMBER(*string, format*)

Converts *string* to a numeric value using the second argument for the input format. For example:

```
SELECT TO_NUMBER('12,454.8-', '99G999D9S') -> -12454.8
```

TO_TIMESTAMP(*text, format*)

Converts *text* to a timestamp value using the second argument for the input format. (For more on valid format specifiers, see the *TO_DATE* function.) For example:

```
SELECT TO_TIMESTAMP('05 Dec 2000', 'DD Mon YYYY') ->
2000-12-05 00:00:00-08
```

TRANSACTION_TIMESTAMP()

Returns the date and time at which the current transaction was started.

TRANSLATE(*text, from, to*)

Converts the characters found in *text* that also exist in *from* to the corresponding characters in *to*. For example:

```
SELECT TRANSLATE('foo', 'fo', 'ab') -> 'abb'
```

TRUNC(*float8*)

Truncates (toward zero). For example:

```
SELECT TRUNC( PI() ) -> 3
```

VARCHAR(*string*)

Converts *string* to a VARCHAR.

WIDTH(*box*)

Returns the width of *box*. For example:

```
SELECT WIDTH( BOX '((0,0),(3,1))' ) -> 3
```

SQL Server-Supported Functions

This section provides an alphabetical listing of Microsoft SQL Server-supported functions, with examples and corresponding results.

ACOS(*number*)

Returns the arccosine of *number*, ranging from -1 to 1. The result ranges from 0 to π and is expressed in radians. For example:

```
SELECT ACOS( 0 ) -> 1.570796
```

APP_NAME()

Returns the application name for the current session, set by the application. For example:

```
SELECT APP_NAME() -> 'SQL Enterprise Manager'
```

ASCII(*text*)

Returns the ASCII code of the first character of *text*. For example:

```
SELECT ASCII('x') -> 120
```

ASIN(*number*)

Returns the arcsine of *number*, ranging from -1 to 1 . The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ASIN( 0 ) -> 0.000000
```

ATAN(*number*)

Returns the arctangent of any *number*. The resulting value ranges from $-\pi/2$ to $\pi/2$ and is expressed in radians. For example:

```
SELECT ATAN( 3.1415 ) -> 1.262619
```

ATN2(*float1*, *float2*)

Returns the angle (in radians) whose tangent is *float1*/*float2*. For example:

```
SELECT ATN2( 35.175643, 129.44 ) -> 0.265345
```

BINARY_CHECKSUM(* | *expression*[, ...*n*])

Returns the binary checksum for a list of expressions or for a row of a table. This example returns a list of user IDs where the stored password checksum doesn't match the current password's checksum:

```
SELECT userid AS 'Changed' FROM users
WHERE NOT password_chksum = BINARY_CHECKSUM( password )
```

CHAR(*integer_expression*)

Converts a numeric ASCII code to a character. For example:

```
SELECT CHAR( 78 ) -> 'N'
```

CHARINDEX(*substring*, *string*[, *start_location*])

Returns the position of the first occurrence of *substring* in *string*, optionally from the given *start_location*. For example:

```
SELECT CHARINDEX( 'he', 'Howdy, there!' ) -> 9
```

CHECKSUM(* | *expression*[, ...*n*])

Returns a checksum (computed over row values or expressions provided). The following example returns a list of user IDs for which the stored password checksum doesn't match the current password's checksum:

```
SELECT userid AS 'Changed' FROM users
WHERE NOT password_chksum = CHECKSUM( password )
```

CHECKSUM_AGG([*ALL* / *DISTINCT*] *expression*)

Returns the checksum of the values in a group. For example:

```
SELECT CHECKSUM_AGG( BINARY_CHECKSUM(*) ) FROM authors -> 67
```

COALESCE(*expression*[, ...*n*])

Returns the first non-NULL argument from a list of arguments. For example:

```
SELECT COALESCE( NULL, 1, 3, 5, 7 ) -> 1
```

COL_LENGTH(*table*, *column*)

Returns the length of *column* in bytes. For example:

```
SELECT COL_LENGTH('authors', 'au_fname') -> 50
```

COL_NAME(*table_id*, *column_id*)Returns the column name, given *table_id* and *column_id*. For example:**SELECT COL_NAME(OBJECT_ID('authors'), 1)****CONTAINS({*column* | *}, *contains_search_condition*)**Searches *column* for exact or “fuzzy” matches of the *contains_search_condition*. **CONTAINS** is an elaborate function used to perform full-text searches; refer to the vendor documentation for more information. This example returns all product IDs from the **products** table that contain the words “peanut” and “butter” in close proximity to each other.**SELECT productid FROM products
WHERE CONTAINS(productname, ' "peanut" NEAR "butter" ')****CONTAINSTABLE(*table*, *column*, *contains_search_condition*)**Returns a *table* with exact and “fuzzy” matches to *contains_search_condition*. **CONTAINSTABLE** is an elaborate function used to perform full-text searches. The following example returns all product ID’s from the **products** table that contain the words “peanut” and “butter” in close proximity to each other:**SELECT productid FROM products
WHERE CONTAINS(products, productname, ' "peanut" NEAR "butter" ')****CONVERT(*data_type*[*(length*)], *expression*[, *style*])**Converts data in *expression* from one datatype to another. For example:**SELECT CONVERT(VARCHAR(50), CURRENT_TIMESTAMP, 1) -> '06/29/03'****COS(*number*)**Returns the cosine of *number* as an angle expressed in radians. For example:**SELECT COS(0) -> 1.000000****COT(*number*)**Returns the cotangent of *number*. For example:**SELECT COT(3.1415) -> -10792.88993953****COUNT_BIG({ [ALL | DISTINCT] *expression* } | *)**Operates the same as COUNT, except the return type is a *BIGINT*. For example:**SELECT COUNT_BIG(names) FROM people -> 26743****DATABASEPROPERTYEX(*database*, *property*)**

Returns a database option or property. For example:

SELECT DATABASEPROPERTYEX('pubs', 'Version') -> 539**DATALENGTH(*expression*)**

Returns the number of bytes in a character or binary string. For example:

SELECT MAX(DATALENGTH(au_fname)) FROM authors -> 11**DATEADD(*datepart*, *number*, *date*)**Adds a number of *dateparts* (e.g., days) to a datetime value. For example:**SELECT DATEADD(Year, 10, CURRENT_TIMESTAMP) -> 2013-06-29 19:47:15.270**

DATEDIFF(*datepart*, *startdate*, *enddate*)

Calculates the difference between two datetime values expressed in the specified *datepart*. For example:

```
SELECT DATEDIFF( Day, CURRENT_TIMESTAMP,  
    DATEADD( Year, 1, CURRENT_TIMESTAMP ))
```

366

DATENAME(*datepart*, *date*)

Returns the name of a *datepart* (e.g., month) of a datetime argument. For example:

```
SELECT DATENAME(month, GETDATE()) -> 'June'
```

DATEPART(*datepart*, *date*)

Returns the value of a *datepart* (e.g., year) of a datetime argument. For example:

```
SELECT DATEPART(year, GETDATE()) -> 2003
```

DAY(*date*)

Returns an integer value representing the day of the *date* provided as a parameter. For example:

```
SELECT DAY('04/15/2004') -> 15
```

DB_ID([*database_name*])

Returns the database ID when provided with a *database_name*. For example:

```
SELECT DB_ID() -> 5
```

DB_NAME(*database_id*)

Returns the database name when provided with a *database_id*. For example:

```
SELECT DB_NAME( 5 ) -> 'pubs'
```

DEGREES(*numeric_expression*)

Converts radians to degrees. For example:

```
SELECT DEGREES( PI() ) -> 180
```

DIFFERENCE(*character_expression*, *character_expression*)

Compares how two arguments sound and returns a number from 0 to 4, with a higher result indicating a better phonetic match. For example:

```
SELECT DIFFERENCE( 'moe', 'low' ) -> 3
```

FILE_ID(*file_name*)

Returns the file ID for the logical *file_name*. For example:

```
SELECT FILE_ID( 'master' ) -> 1
```

FILE_NAME(*file_id*)

Returns the logical filename for the *file_id*. For example:

```
SELECT FILE_NAME( 1 ) -> 'master'
```

FILEGROUP_ID(*filegroup_name*)

Returns the filegroup ID for the logical *filegroup_name*. For example:

```
SELECT FILEGROUP_ID( 'PRIMARY' ) -> 1
```

FILEGROUP_NAME(*filegroup_id*)Returns the logical filegroup name for *filegroup_id*. For example:**SELECT FILEGROUP_NAME(1) -> 'PRIMARY'****FILEGROUPPROPERTY(*filegroup_name*, *property*)**Returns the filegroup property value for the specified *property*. For example:**SELECT FILEGROUPPROPERTY('PRIMARY', 'IsReadOnly') -> 0****FILEPROPERTY(*file*, *property*)**Returns the *file* property value for the specified *property*. For example:**SELECT FILEPROPERTY('pubs', 'SpaceUsed') -> 160****FORMATMESSAGE(*msg_number*, *param_value*[, ... *n*])**

Constructs a message from an existing message in the SYSMESSAGES table (similar to RAISEERROR). For example:

sp_addmessage 50001, 1, 'Table %s has %s rows.'**SELECT FORMATMESSAGE(50001, 'AUTHORS', (SELECT COUNT(*) FROM AUTHORS))**
'Table AUTHORS has 23.'**FREETEXT({*column* | *}, *freetext_string*)**Used for full-text searches. Returns rows with *column* values that match the meaning, but not exactly the value, of *freetext_string*.**FREETEXTTABLE(*table*, {*column* | *}, *freetext_string*[, *top_n_by_rank*])**Used for full-text searches. Returns rows from *table* with *column* values that match the meaning, but not exactly the value, of *freetext_string*. For example:**SELECT * from FREETEXTTABLE (authors, *, 'kev')****FULLTEXTCATALOGPROPERTY(*catalog_name*, *property*)**

Returns the full-text catalog properties. For example:

SELECT FULLTEXTCATALOGPROPERTY('Cat_Desc', 'LogSize')**FULLTEXTSERVICEPROPERTY(*property*)**

Returns the full-text service-level properties. For example:

SELECT FULLTEXTSERVICEPROPERTY('IsFulltextInstalled') -> 1**GETANSINULL({*database*})**

Returns the default nullability setting for new columns. For example:

SELECT GETANSINULL() -> 1**GETDATE()**

Returns the current date and time. For example:

SELECT GETDATE() -> 2003-06-27 19:26:59.893**GETUTCDATE()**

Returns the current date as a Coordinated Universal Time (UTC) date. For example:

SELECT GETUTCDATE() -> 2003-06-28 02:26:46.720

GROUPING(*column_name*)

Returns 1 when a row is added by *CUBE* or *ROLLUP*; otherwise, returns 0. For example:

```
SELECT royalty, SUM(advance) 'total advance', GROUPING(royalty) 'grp'
  FROM titles GROUP BY royalty WITH ROLLUP
    royalty      total advance      grp
    -----      -----      -----
    NULL          NULL          0
    10           57000.0000      0
    12           2275.0000      0
    14           4000.0000      0
    16           7000.0000      0
    24           25125.0000      0
    NULL          95400.0000      1
```

HOST_ID()

Returns the workstation ID. For example:

```
SELECT HOST_ID() -> 216
```

HOST_NAME()

Returns the process host name. For example:

```
SELECT HOST_NAME() -> 'PLATO'
```

IDENT_CURRENT(*table_name*)

Returns the last identity value generated for the specified table. For example:

```
SELECT IDENT_CURRENT('jobs') -> 876
```

IDENT_INCR(*table_or_view*)

Returns an identity column increment value. For example:

```
SELECT IDENT_INCR('jobs') -> 1
```

IDENT_SEED(*table_or_view*)

Returns an identity seed value. For example:

```
SELECT IDENT_SEED('jobs') -> 1
```

IDENTITY(*data_type[, seed, increment]*) AS *column_name*

Used in a *SELECT INTO* statement to insert an identity column into the destination table. For example:

```
SELECT IDENTITY(int, 1,1) AS IDINTO NewTableFROM OldTable
```

INDEX_COL(*table, index_id, key_id*)

Returns an index column name given a table name, an index ID, and the sequential number of the column in the index key. For example:

```
SELECT INDEX_COL(OBJECT_ID('authors'), 1, 1) -> NULL
```

INDEXPROPERTY(*table_id, index, property*)

Returns an index property (such as *FILLCFACTOR*). For example:

```
SELECT INDEXPROPERTY(OBJECT_ID('authors'), 'UPKCL_auidind', 'IsPadIndex')
      0
```

IS_MEMBER({group | role})

Returns true or false (1 or 0) depending on whether or not the user is a member of the specified Windows NT *group* or SQL Server *role*. For example:

```
SELECT IS_MEMBER( 'db_owner' ) -> 0
```

IS_SRVROLEMEMBER(role[, login])

Returns true or false (1 or 0) depending on whether or not the user is a member of the specified server *role*. For example:

```
SELECT IS_SRVROLEMEMBER( 'sysadmin' ) -> 0
```

ISDATE(expression)

Validates whether a character string can be converted to *DATETIME*. For example:

```
SELECT ISDATE(NULL), ISDATE(GETDATE())
0      1
```

ISNULL(check_expression, replacement_value)

Returns the first argument if it is not NULL; otherwise, returns the second argument. For example:

```
SELECT ISNULL( NULL, 'NULL' ) -> 'NULL'
```

ISNUMERIC(expression)

Validates whether a character string can be converted to *NUMERIC*. For example:

```
SELECT ISNUMERIC('3.1415'), ISNUMERIC('IRK')
1      0
```

LEFT(character_expression, integer_expression)

Returns the leftmost *integer_expression* characters of *character_expression*. For example:

```
SELECT LEFT( 'Wet Paint', 3 ) -> 'Wet'
```

LEN(string_expression)

Returns the number of characters in the expression. For example:

```
SELECT LEN( 'Wet Paint' ) -> 9
```

LOG(float_expression)

Returns the natural logarithm. For example:

```
SELECT LOG( PI() ) -> 1.1447298858494002
```

LOG10(float_expression)

Returns the base-10 logarithm. For example:

```
SELECT LOG10( PI() ) -> 0.49714987269413385
```

LTRIM(character_expression)

Trims leading space characters. For example:

```
SELECT LTRIM('    beaucoup    ') -> 'beaucoup    '
```

MONTH(date)

Returns the month part of the date provided. For example:

```
SELECT MONTH( GETDATE() ) -> 6
```

NCHAR(*integer_expression*)

Returns the UNICODE character with the given integer code. For example:

```
SELECT NCHAR(120) -> 'x'
```

NEWID()

Creates a new unique identifier of type *UNIQUEIDENTIFIER*. For example:

```
SELECT NEWID() -> '32B35185-F55E-4FE0-B2C8-B57B35815C12'
```

NULLIF(*expression*, *expression*)

Returns NULL if the two specified expressions are equivalent. For example:

```
SELECT NULLIF( 5, 5 ) -> NULL
```

OBJECT_ID(*object*)

Returns the object ID of *object*. For example:

```
SELECT OBJECT_NAME ( OBJECT_ID('authors') ) -> 8
```

OBJECT_NAME(*object_id*)

Returns the object name of the object with the given object ID. For example:

```
SELECT OBJECT_NAME ( OBJECT_ID('authors') ) -> 'authors'
```

OBJECTPROPERTY(*id*, *property*)

Returns the properties of objects in the current database. For example:

```
SELECT OBJECTPROPERTY ( object_id('authors'), 'ISTABLE' ) -> 1
```

OPEN {{[GLOBAL] *cursor_name*} | *cursor_variable_name*}

Opens a local or global cursor.

OPENDATASOURCE(*provider_name*, *init_string*)

Makes a connection to a data source without using a linked server name. For examples, refer to the “Loaders” section of the SQL Server User’s Guide.

OPENQUERY(*linked_server*, *query*)

Queries a remote data source previously configured as a linked server. For an example, refer to the “Loaders” section of the SQL Server User’s Guide.

OPENROWSET(*provider_name*, {*datasource*, *user_id*, *password* | *provider_string*}, {[*catalog*.][*schema*.]*object* | *query*})

Queries a remote data source without setting it up as a linked server.

PARSENAME(*object_name*, *object_piece*)

Returns the database name, owner name, server name, or object name for the object specified. *object_piece* is an integer between 1 and 4. For example:

```
SELECT PARSENAME('pubs..authors', 1) -> 'authors'
```

```
SELECT PARSENAME('pubs..authors', 2) -> NULL
```

```
SELECT PARSENAME('pubs..authors', 3) -> 'pubs'
```

```
SELECT PARSENAME('pubs..authors', 4) -> NULL
```

PATINDEX(''%*pattern*%', *expression*)

Returns the position of the first occurrence of a pattern in a string. For example:

```
SELECT PATINDEX('%Du%', 'Donald Duck') -> 8
```

PERMISSIONS([object_id[, column]])

Returns a numeric value representing a bitmap with the current user's permissions on the specified object or column. For example:

```
SELECT PERMISSIONS(OBJECT_ID('authors')) & 8 -> 8
```

PI()

Returns the constant *pi*. For example:

```
SELECT 2*PI() -> 6.2831853071795862
```

RADIANS(numeric_expression)

Converts degrees to radians. For example:

```
SELECT RADIANS( 90.0 ) -> 1.570796326794896600
```

RAND([seed])

Returns a pseudorandom *FLOAT* type value between *seed* and 1. For example:

```
SELECT RAND(PI()) -> 0.71362925915543995
```

REPLICATE(character_expression, integer_expression)

Repeats a string a number of times. For example:

```
SELECT REPLICATE( 'FOOBAR', 3 ) -> 'FOOBARFOOBARFOOBAR'
```

REPLACE(string_expression1, string_expression2, string_expression3)

Performs a search-and-replace on *string_expression1*, replacing each occurrence of *string_expression2* with *string_expression3*. For example:

```
SELECT REPLACE('Donald Duck', 'Duck', 'Trump') -> 'Donald Trump'
```

REVERSE(character_expression)

Reverses the characters of a string. For example:

```
SELECT REVERSE( 'Donald Duck' ) -> 'kcuD dlanoD'
```

RIGHT(character_expression, integer_expression)

Returns the rightmost *integer_expression* characters of *character_expression*. For example:

```
SELECT RIGHT( 'Donald Duck', 4 ) -> 'Duck'
```

ROUND(number, decimal[, function])

Returns *number* rounded to *decimal* places to the right of the decimal point. Note that *decimal*, an integer, can be negative to round off digits to the left of the decimal point. If a nonzero integer is provided for *function*, the return value will be truncated; otherwise, the value is rounded. For example:

```
SELECT ROUND(PI(), 2) -> 3.1400000000000001
```

ROWCOUNT_BIG()

Returns the number of rows affected by the most recent query. (Same as @@ROWCOUNT, but returns a *BIGINT* type.) For example:

```
SELECT ROWCOUNT_BIG() -> 1
```

RTRIM(character_expression)

Trims trailing space characters from the expression. For example:

```
SELECT RTRIM('    beaucoup    ') -> '    beaucoup'
```

SIGN(numeric_expression)

Returns -1 if the argument is negative, 0 if it is zero, and 1 if the argument is positive. For example:

```
SELECT SIGN(-PI()) -> -1.0
```

SIN(number)

Returns the sine of *number*, where *number* is in radians. For example:

```
SELECT SIN( 0 ) -> 0.000000
```

SOUNDEX(character_expression)

Returns a four-character code based on how the argument string sounds. For example:

```
SELECT SOUNDEX('char') -> 'C600'
```

SPACE(integer_expression)

Returns a string consisting of a given number of space characters. For example:

```
SELECT SPACE(5) -> '      '
```

STATS_DATE(table_id, index_id)

Returns the date and time that index statistics were last updated. For example:

```
SELECT i.name, STATS_DATE(i.id, i.indid)FROM sysobjects o, sysindexes i
WHERE o.name = 'authors' AND o.id = i.id
UPKCL_auidind 2000-08-06 01:34:00.153
aunmind        2000-08-06 01:34:00.170
```

STDEV(expression)

Returns the standard deviation of the values in *expression*. For example:

```
SELECT STDEV( qty ) FROM sales -> 16.409201831957116
```

STDEVP(expression)

Returns the standard deviation for the population of values in *expression*. For example:

```
SELECT STDEVP( qty ) FROM sales -> 16.013741264834152
```

STR(number[, length[, decimal]])

Converts *number* to a character string with *length* length and *decimal* decimal places.

STUFF(string1, start, length, string2)

Replaces the *length* characters within *string1*, starting with the character in the *start* position, with those in *string2*. For example:

```
SELECT STUFF( 'Donald Duck', 8, 4, 'Trump' ) -> 'Donald Trump'
```

SUBSTRING(string, start, length)

Extracts *length* characters from *string*, starting at the character in the *start* position. For example:

```
SELECT SUBSTRING( 'Donald Duck', 8, 4 ) -> 'Duck'
```

SUSER_ID([login])

Returns the system user ID of a given login name. This function will always return `NULL` with SQL Server 2000 or later, so you should avoid using it.

SUSER_SID([login])

Returns the security ID (SID) for the current user, or for the specified *login*. The SID is returned in binary format. For example:

```
SELECT SUSER_SID('montoyai')
0x68FC17A71010DE40B005BCF2E443B377
```

SUSER_SNAME([server_user_sid])

Returns the login name for the current user, or for the specified login SID. For example:

```
SELECT SUSER_SNAME() -> 'montoyai'
```

TAN(*number*)

Returns the tangent of *number*, where *number* is in radians. For example:

```
SELECT TAN( 3.1415 ) -> -0.000093
```

TEXTPTR(*column*)

Returns a pointer to a *TEXT*, *NTEXT*, or *IMAGE* column in *VARBINARY* format. For example:

```
SELECT TEXTPTR(pr_info)FROM pub_info WHERE pub_id = '0736'ORDER BY pub_id
0xFFE6F0000000005C0000001000100
```

TEXTVALID(*table.column*, *text_ptr*)

Returns true or false (1 or 0), depending on whether or not the provided pointer to a *TEXT*, *NTEXT*, or *IMAGE* column is valid. For example:

```
SELECT pub_id, 'Valid (if 1) Text data'
= TEXTVALID('pub_info.logo', TEXTPTR(logo)) FROM pub_info ORDER BY pub_id
0736      1
0877      1
1389      1
1622      1
1756      1
9901      1
9952      1
9999      1
```

TYPEPROPERTY(*datatype*, *property*)

Returns information about datatype properties. The *datatype* argument can contain the name of any datatype, and *property* can be a string containing one of the following:

Precision

The precision of the *datatype* is the number of digits or characters that it can store.

Scale

The scale is the number of decimal places for a numeric datatype. A NULL value will be returned if *datatype* is not a numeric datatype.

For example:

```
SELECT TYPEPROPERTY('decimal', 'PRECISION') -> 38
```

UNICODE(*ncharacter_expression*)

Returns the UNICODE code point for the first character of the input parameter. For example:

```
SELECT UNICODE('Hello!') -> 72
```

USER_ID([*user*])

Returns the user ID for *user* in the current database. If *user* is omitted, the current user's ID will be returned. For example:

```
SELECT USER_ID() -> 2
```

USER_NAME(*id*)

Returns the username of the user identified by *id* or, if no ID is provided, of the current user. For example:

```
SELECT USER_NAME() -> 'montoyai'
```

VAR(*expression*)

Returns the statistical variance for the values represented by *expression*. For example:

```
SELECT VAR(qty) FROM sales -> 269.26190476190476
```

VARP(*expression*)

Returns the statistical variance for the population represented by all values of *expression* in a group. VARP is an aggregate function. For example:

```
SELECT VARP(qty) FROM sales -> 256.43990929705217
```

YEAR(*date*)

Returns an integer that is the YEAR part of the specified date. For example:

```
SELECT YEAR( CURRENT_TIMESTAMP ) -> 2003
```



APPENDIX

Shared and Platform-Specific Keywords

The following tables display the keywords in the ANSI standard, and in the four platform-specific implementations of SQL that are discussed in this book, with the Sybase Adaptive Server. The keyword tables appear in the following order:

- Shared keywords
- SQL2003 keywords
- MySQL keywords
- Oracle keywords
- PostgreSQL keywords
- SQL Server keywords

Table A-1. Shared keywords

<i>ADD</i>	<i>ALL</i>	<i>ALTER</i>	<i>AND</i>
<i>AS</i>	<i>ASC</i>	<i>BY</i>	<i>CHECK</i>
<i>COLUMN</i>	<i>CREATE</i>	<i>DATE</i>	<i>DEFAULT</i>
<i>DELETE</i>	<i>DESC</i>	<i>DROP</i>	<i>FOR</i>
<i>FROM</i>	<i>IN</i>	<i>INTO</i>	<i>IS</i>
<i>LIKE</i>	<i>NOT</i>	<i>NULL</i>	<i>ON</i>
<i>OR</i>	<i>ORDER</i>	<i>REVOKE</i>	<i>SELECT</i>
<i>SET</i>	<i>TABLE</i>	<i>THEN</i>	<i>TO</i>
<i>UNIQUE</i>	<i>UPDATE</i>	<i>WITH</i>	

Table A-2. SQL2003 keywords

<i>ABSOLUTE</i>	<i>ACTION</i>	<i>ADD</i>	<i>ADMIN</i>
<i>AFTER</i>	<i>AGGREGATE</i>	<i>ALIAS</i>	<i>ALL</i>
<i>ALLOCATE</i>	<i>ALTER</i>	<i>AND</i>	<i>ANY</i>

Table A-2. SQL2003 keywords (continued)

ARE	ARRAY	AS	ASC
ASSERTION	AT	ATOMIC	AUTHORIZATION
BEFORE	BEGIN	BIGINT	BINARY
BIT	BLOB	BOOLEAN	BOTH
BREADTH	BY	CALL	CASCADE
CASCADED	CASE	CAST	CATALOG
CHAR	CHARACTER	CHECK	CLASS
CLOB	CLOSE	COLLATE	COLLATION
COLLECT	COLUMN	COMMIT	COMPLETION
CONDITION	CONNECT	CONNECTION	CONSTRAINT
CONSTRAINTS	CONSTRUCTOR	CONTAINS	CONTINUE
CORRESPONDING	CREATE	CROSS	CUBE
CURRENT	CURRENT_DATE	CURRENT_PATH	CURRENT_ROLE
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
CYCLE	DATA	DATALINK	DATE
DAY	DEALLOCATE	DEC	DECIMAL
DECLARE	DEFAULT	DEFERRABLE	DELETE
DEPTH	DREF	DESC	DESCRIPTOR
DESTRUCTOR	DIAGNOSTICS	DICTIONARY	DISCONNECT
DO	DOMAIN	DOUBLE	DROP
ELEMENT	END-EXEC	EQUALS	ESCAPE
EXCEPT	EXCEPTION	EXECUTE	EXIT
EXPAND	EXPANDING	FALSE	FIRST
FLOAT	FOR	FOREIGN	FREE
FROM	FUNCTION	FUSION	GENERAL
GET	GLOBAL	GOTO	GROUP
GROUPING	HANDLER	HASH	HOUR
IDENTITY	IF	IGNORE	IMMEDIATE
IN	INDICATOR	INITIALIZE	INITIALLY
INNER	INOUT	INPUT	INSERT
INT	INTEGER	INTERSECT	INTERSECTION
INTERVAL	INTO	IS	ISOLATION
ITERATE	JOIN	KEY	LANGUAGE
LARGE	LAST	LATERAL	LEADING
LEAVE	LEFT	LESS	LEVEL
LIKE	LIMIT	LOCAL	LOCALTIME
LOCALTIMESTAMP	LOCATOR	LOOP	MATCH
MEMBER	MEETS	MERGE	MINUTE
MODIFIES	MODIFY	MODULE	MONTH
MULTISET	NAMES	NATIONAL	NATURAL
NCHAR	NCLOB	NEW	NEXT
NO	NONE	NORMALIZE	NOT

Table A-2. SQL2003 keywords (continued)

NULL	NUMERIC	OBJECT	OF
OFF	OLD	ON	ONLY
OPEN	OPERATION	OPTION	OR
ORDER	ORDINALITY	OUT	OUTER
OUTPUT	PAD	PARAMETER	PARAMETERS
PARTIAL	PATH	PERIOD	POSTFIX
PRECEDES	PRECISION	PREFIX	PREORDER
PREPARE	PRESERVE	PRIMARY	PRIOR
PRIVILEGES	PROCEDURE	PUBLIC	READ
READS	REAL	RECURSIVE	REDO
REF	REFERENCES	REFERENCING	RELATIVE
REPEAT	RESIGNAL	RESTRICT	RESULT
RETURN	RETURNS	REVOKE	RIGHT
ROLE	ROLLBACK	ROLLUP	ROUTINE
ROW	ROWS	SAVEPOINT	SCHEMA
SCROLL	SEARCH	SECOND	SECTION
SELECT	SEQUENCE	SESSION	SESSION_USER
SET	SETS	SIGNAL	SIZE
SMALLINT	SPECIFIC	SPECIFICTYPE	SQL
SQLEXCEPTION	SQLSTATE	SQLWARNING	START
STATE	STATIC	STRUCTURE	SUBMULTISET
SUCCEEDS	SUM	SYSTEM_USER	TABLE
TABLESAMPLE	TEMPORARY	TERMINATE	THAN
THEN	TIME	TIMESTAMP	TIMEZONE_HOUR
TIMEZONE_MINUTE	TO	TRAILING	TRANSACTION
TRANSLATION	TREAT	TRIGGER	TRUE
UESCAPE	UNDER	UNDO	UNION
UNIQUE	UNKNOWN	UNTIL	UPDATE
USAGE	USER	USING	VALUE
VALUES	VARCHAR	VARIABLE	VARYING
VIEW	WHEN	WHENEVER	WHERE
WHILE	WITH	WRITE	YEAR
ZONE			

Keywords

Table A-3. MySQL keywords

ACCESSIBLE	ADD	ALL	ALTER
ANALYZE	AND	AS	ASC
ASENSITIVE	BEFORE	BETWEEN	BIGINT
BINARY	BLOB	BOTH	BY
CALL	CASCADE	CASE	CHANGE
CHAR	CHARACTER	CHECK	COLLATE

Table A-3. MySQL keywords (continued)

COLUMN	CONDITION	CONSTRAINT	CONTINUE
CONVERT	CREATE	CROSS	CURRENT_DATE
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DATABASE	DATABASES	DAY_HOUR	DAY_MICROSECOND
DAY_MINUTE	DAY_SECOND	DEC	DECIMAL
DECLARE	DEFAULT	DELAYED	DELETE
DESC	DESCRIBE	DETERMINISTIC	DISTINCT
DISTINCTROW	DIV	DOUBLE	DROP
DUAL	EACH	ELSE	ELSEIF
ENCLOSED	ESCAPED	EXISTS	EXIT
EXPLAIN	FALSE	FETCH	FLOAT
FLOAT4	FLOAT8	FOR	FORCE
FOREIGN	FROM	FULLTEXT	GRANT
GROUP	HAVING	HIGH_PRIORITY	HOUR_MICROSECOND
HOUR_MINUTE	HOUR_SECOND	IF	IGNORE
IN	INDEX	INFILE	INNER
INOUT	INSENSITIVE	INSERT	INT
INT1	INT2	INT3	INT4
INT8	INTEGER	INTERVAL	INTO
IS	ITERATE	JOIN	KEY
KEYS	KILL	LEADING	LEAVE
LEFT	LIKE	LIMIT	LINEAR
LINES	LOAD	LOCALTIME	LOCALTIMESTAMP
LOCK	LONG	LONGBLOB	LONGTEXT
LOOP	LOW_PRIORITY	MASTER_SSL_VERIFY_SERVER_CERT	MATCH
MEDIUMBLOB	MEDIUMINT	MEDIUMTEXT	MIDDLEINT
MINUTE_MICROSECOND	MINUTE_SECOND	MOD	MODIFIES
NATURAL	NO_WRITE_TO_BINLOG	NOT	NULL
NUMERIC	ON	OPTIMIZE	OPTION
OPTIONALLY	OR	ORDER	OUT
OUTER	OUTFILE	PRECISION	PRIMARY
PROCEDURE	PURGE	RANGE	READ
READ_ONLY	READ_WRITE	READS	REAL
REFERENCES	REGEXP	RELEASE	RENAME
REPEAT	REPLACE	REQUIRE	RESTRICT
RETURN	REVOKE	RIGHT	RLIKE
SCHEMA	SCHEMAS	SECOND_MICROSECOND	SELECT
SENSITIVE	SEPARATOR	SET	SHOW
SMALLINT	SPATIAL	SPECIFIC	SQL

Table A-3. MySQL keywords (continued)

<code>SQL_BIG_RESULT</code>	<code>SQL_CALC_FOUND_ROWS</code>	<code>SQL_SMALL_RESULT</code>	<code>SQLEXCEPTION</code>
<code>SQLSTATE</code>	<code>SQLWARNING</code>	<code>SSL</code>	<code>STARTING</code>
<code>STRAIGHT_JOIN</code>	<code>TABLE</code>	<code>TERMINATED</code>	<code>THEN</code>
<code>TINYBLOB</code>	<code>TINYINT</code>	<code>TINYTEXT</code>	<code>TO</code>
<code>TRAILING</code>	<code>TRIGGER</code>	<code>TRUE</code>	<code>UNDO</code>
<code>UNION</code>	<code>UNIQUE</code>	<code>UNLOCK</code>	<code>UNSIGNED</code>
<code>UPDATE</code>	<code>USAGE</code>	<code>USE</code>	<code>USING</code>
<code>UTC_DATE</code>	<code>UTC_TIME</code>	<code>UTC_TIMESTAMP</code>	<code>VALUES</code>
<code>VARBINARY</code>	<code>VARCHAR</code>	<code>VARCHARACTER</code>	<code>VARYING</code>
<code>WHEN</code>	<code>WHERE</code>	<code>WHILE</code>	<code>WITH</code>
<code>WRITE</code>	<code>XOR</code>	<code>YEAR_MONTH</code>	<code>ZEROFILL</code>

Table A-4. Oracle keywords

<code>ACCESS</code>	<code>ADD</code>	<code>ALL</code>	<code>ALTER</code>
<code>AND</code>	<code>ANY</code>	<code>AS</code>	<code>ASC</code>
<code>AUDIT</code>	<code>BETWEEN</code>	<code>BY</code>	<code>CHAR</code>
<code>CHECK</code>	<code>CLUSTER</code>	<code>COLUMN</code>	<code>COMMENT</code>
<code>COMPRESS</code>	<code>CONNECT</code>	<code>CREATE</code>	<code>CURRENT</code>
<code>DATE</code>	<code>DECIMAL</code>	<code>DEFAULT</code>	<code>DELETE</code>
<code>DESC</code>	<code>DISTINCT</code>	<code>DROP</code>	<code>ELSE</code>
<code>EXCLUSIVE</code>	<code>EXISTS</code>	<code>FILE</code>	<code>FLOAT</code>
<code>FOR</code>	<code>FROM</code>	<code>GRANT</code>	<code>GROUP</code>
<code>HAVING</code>	<code>IDENTIFIED</code>	<code>IMMEDIATE</code>	<code>IN</code>
<code>INCREMENT</code>	<code>INDEX</code>	<code>INITIAL</code>	<code>INSERT</code>
<code>INTEGER</code>	<code>INTERSECT</code>	<code>INTO</code>	<code>IS</code>
<code>LEVEL</code>	<code>LIKE</code>	<code>LOCK</code>	<code>LONG</code>
<code>MAXEXTENTS</code>	<code>MINUS</code>	<code>MLSLABEL</code>	<code>MODE</code>
<code>MODIFY</code>	<code>NOAUDIT</code>	<code>NOCOMPRESS</code>	<code>NOT</code>
<code>NOWAIT</code>	<code>NULL</code>	<code>NUMBER</code>	<code>OF</code>
<code>OFFLINE</code>	<code>ON</code>	<code>ONLINE</code>	<code>OPTION</code>
<code>OR</code>	<code>ORDER</code>	<code>PCTFREE</code>	<code>PRIOR</code>
<code>PRIVILEGES</code>	<code>PUBLIC</code>	<code>RAW</code>	<code>RENAME</code>
<code>RESOURCE</code>	<code>REVOKE</code>	<code>ROW</code>	<code>ROWID</code>
<code>ROWNUM</code>	<code>ROWS</code>	<code>SELECT</code>	<code>SESSION</code>
<code>SET</code>	<code>SHARE</code>	<code>SIZE</code>	<code>SMALLINT</code>
<code>START</code>	<code>SUCCESSFUL</code>	<code>SYNONYM</code>	<code>SYSDATE</code>
<code>TABLE</code>	<code>THEN</code>	<code>TO</code>	<code>TRIGGER</code>
<code>UID</code>	<code>UNION</code>	<code>UNIQUE</code>	<code>UPDATE</code>
<code>USER</code>	<code>VALIDATE</code>	<code>VALUES</code>	<code>VARCHAR</code>

Table A-5. PostgreSQL keywords

ABORT	ADD	ALL	ALLOCATE
ALTER	ANALYZE	AND	ANY
ARE	AS	ASC	ASSERTION
AT	AUTHORIZATION	AVG	BEGIN
BETWEEN	BINARY	BIT	BIT_LENGTH
BOTH	BY	CASCADE	CASCADED
CASE	CAST	CATALOG	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CLOSE	CLUSTER	COALESCE	COLLATE
COLLATION	COLUMN	COMMIT	CONNECT
CONNECTION	CONSTRAINT	CONTINUE	CONVERT
COPY	CORRESPONDING	COUNT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_SESSION
CURRENT_TIME	CURRENT_TIMESTAMP	CURRENT_USER	CURSOR
DATE	DEALLOCATE	DEC	DECIMAL
DECLARE	DEFAULT	DELETE	DESC
DESCRIBE	DESCRIPTOR	DIAGNOSTICS	DISCONNECT
DISTINCT	DO	DOMAIN	DROP
ELSE	END	ESCAPE	EXCEPT
EXCEPTION	EXEC	EXECUTE	EXISTS
EXPLAIN	EXTEND	EXTERNAL	EXTRACT
FALSE	FETCH	FIRST	FLOAT
FOR	FOREIGN	FOUND	FROM
FULL	GET	GLOBAL	GO
GOTO	GRANT	GROUP	HAVING
IDENTITY	IN	INDICATOR	INNER
INPUT	INSERT	INTERSECT	INTERVAL
INTO	IS	JOIN	LAST
LEADING	LEFT	LIKE	LISTEN
LOAD	LOCAL	LOCK	LOWER
MAX	MIN	MODULE	MOVE
NAMES	NATIONAL	NATURAL	NCHAR
NEW	NO	NONE	NOT
NOTIFY	NULL	NULLIF	NUMERIC
OCTET_LENGTH	OFFSET	ON	OPEN
OR	ORDER	OUTER	OUTPUT
OVERLAPS	PARTIAL	POSITION	PRECISION
PREPARE	PRESERVE	PRIMARY	PRIVILEGES
PROCEDURE	PUBLIC	REFERENCES	RESET
REVOKE	RIGHT	ROLLBACK	ROWS
SCHEMA	SECTION	SELECT	SESSION
SESSION_USER	SET	SETOF	SHOW

Table A-5. PostgreSQL keywords (continued)

SIZE	SOME	SQL	SQLCODE
SQLERROR	SQLSTATE	SUBSTRING	SUM
SYSTEM_USER	TABLE	TEMPORARY	THEN
TO	TRAILING	TRANSACTION	TRANSLATE
TRANSLATION	TRIM	TRUE	UNION
UNIQUE	UNKNOWN	UNLISTEN	UNTIL
UPDATE	UPPER	USAGE	USER
USING	VACUUM	VALUE	VALUES
VARCHAR	VARYING	VERBOSE	VIEW
WHEN	WHENEVER	WHERE	WITH
WORK	WRITE		

Table A-6. SQL Server keywords

ADD	ALL	ALTER	AND
ANY	AS	ASC	AUTHORIZATION
BACKUP	BEGIN	BETWEEN	BREAK
BROWSE	BULK	BY	CASCADE
CASE	CHECK	CHECKPOINT	CLOSE
CLUSTERED	COALESCE	COLLATE	COLUMN
COMMIT	COMPUTE	CONSTRAINT	CONTAINS
CONTAINSTABLE	CONTINUE	CONVERT	CREATE
CROSS	CURRENT	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURRENT_USER	CURSOR	DATABASE
DBCC	DEALLOCATE	DECLARE	DEFAULT
DELETE	DENY	DESC	DISK
DISTINCT	DISTRIBUTED	DOUBLE	DROP
DUMP	ELSE	END	ERRLVL
ESCAPE	EXCEPT	EXEC	EXECUTE
EXISTS	EXIT	EXTERNAL	FETCH
FILE	FILLFACTOR	FOR	FOREIGN
FREETEXT	FREETEXTTABLE	FROM	FULL
FUNCTION	GOTO	GRANT	GROUP
HAVING	HOLDLOCK	IDENTITY	IDENTITY_INSERT
IDENTITYCOL	IF	IN	INDEX
INNER	INSERT	INTERSECT	INTO
IS	JOIN	KEY	KILL
LEFT	LIKE	LINENO	LOAD
NATIONAL	NOCHECK	NONCLUSTERED	NOT
NULL	NULLIF	OF	OFF
OFFSETS	ON	OPEN	OPENDATASOURCE
OPENQUERY	OPENROWSET	OPENXML	OPTION

Table A-6. SQL Server keywords (continued)

OR	ORDER	OUTER	OVER
PERCENT	PIVOT	PLAN	PRECISION
PRIMARY	PRINT	PROC	PROCEDURE
PUBLIC	RAISERROR	READ	READTEXT
RECONFIGURE	REFERENCES	REPLICATION	RESTORE
RESTRICT	RETURN	REVERT	REVOKE
RIGHT	ROLLBACK	ROWCOUNT	ROWGUIDCOL
RULE	SAVE	SCHEMA	SECURITYAUDIT
SELECT	SESSION_USER	SET	SETUSER
SHUTDOWN	SOME	STATISTICS	SYSTEM_USER
TABLE	TABLESAMPLE	TEXTSIZE	THEN
TO	TOP	TRAN	TRANSACTION
TRIGGER	TRUNCATE	TSEQUAL	UNION
UNIQUE	UNPIVOT	UPDATE	UPDATETEXT
USE	USER	VALUES	VARYING
VIEW	WAITFOR	WHEN	WHERE
WHILE	WITH	WRITETEXT	

Index

Symbols

+ (addition arithmetic operator), 24, 27
= (assignment operator), 24
:= (assignment operator), Oracle, 24
@ (at sign), prefixing parameters, SQL Server, 114
& (bitwise AND operator), 24
^ (bitwise exclusive OR operator), 24
~ (bitwise NOT unary operator), 26
| (bitwise OR operator), 24
[...] (brackets), enclosing delimited identifiers, 20
-- (comments, single-line), 29
/*...*/ (comments, multiline), 29
+ (concatenation operator, SQL Server), 27
|| (concatenation operator), 477
/ (division arithmetic operator), 24, 27
..." (double quotes), enclosing delimited identifiers, 19, 20, 28
= (equal to comparison operator), 25, 27
> (greater than operator), 25, 28
>= (greater than or equal to operator), 25, 28
. (identifier qualifier separator), 28
< (less than operator), 25, 28
<= (less than or equal to operator), 25, 28
, (list item separator), 28

% (modula arithmetic operator), 24
* (multiplication arithmetic operator), 24, 27
- (negative unary operator), 26
<> (not equal to operator), 25, 27
!= (not equal to operator), 25, 27
!> (not greater than operator), 25
!< (not less than operator), 25
(...) (parentheses), affecting operator precedence, 26, 28
+ (positive unary operator), 26
(pound symbol)
prefixing temporary procedures, SQL Server, 114
prefixing temporary tables, SQL Server, 190
- (range indicator in CHECK constraint), 27
'...' (single quotes)
enclosing delimited identifiers, 20
enclosing string literals, 22, 28
-(subtraction arithmetic operator), 24, 27
_ (underscore), in identifiers, 18
_ (wildcard operator, matches any character), 317
% (wildcard operator, matches any string), 28, 317
[...] (wildcard operator, matches in set, SQL Server), 319
[^...] (wildcard operator, matches not in set, SQL Server), 319

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

A

- abbreviations in identifiers, 18
ABS function, 467
ABSTIME function, PostgreSQL, 524
access control, 5
ACOS function
 MySQL, 483
 Oracle, 500
 PostgreSQL, 524
 SQL Server, 535
ADDDATE function, MySQL, 483, 486
addition arithmetic operator (+), 24, 27
ADD_MONTHS function, Oracle, 500
AES_DECRYPT function, MySQL, 484
AES_ENCRYPT function, MySQL, 484
AGE function, PostgreSQL, 524
aggregate functions, 358, 438–454
 AVG, 440
 CORR, 441
 COUNT, 441
 COVAR_POP, 442
 COVAR_SAMP, 443
 CUME_DIST, 444, 459
 DENSE_RANK, 445, 460
 MAX, 446
 MIN, 446
 PERCENTILE_CONT, 448
 PERCENTILE_DISC, 449
 PERCENT_RANK, 447, 461
 RANK, 449, 461
 REGR_, 451
 STDDEV_POP, 452
 STDDEV_SAMP, 453
 SUM, 440
 VAR_POP, 453, 522
 VAR_SAMP, 454, 522
 as window functions, 458
 (see also GROUP BY clause;
 HAVING clause)
- aliases
 assigning, 24
 WHERE clause joins and, 354
ALL operator, 25, 60, 63–65, 358
ALTER DATABASE statement, 61, 75–101
 MySQL, 76
 Oracle, 76–91
 PostgreSQL, 91
 SQL Server, 93–101
ALTER DOMAIN statement, 60
ALTER FUNCTION statement, 61, 101–117
 MySQL, 106
 Oracle, 107–110
 PostgreSQL, 110
 SQL Server, 112–117
ALTER INDEX statement, 61, 117–133
 Oracle, 120–128
 PostgreSQL, 128–130
 SQL Server, 130–133
ALTER METHOD statement, 61, 133
ALTER PROCEDURE statement, 61, 101–117
 MySQL, 106
 Oracle, 107–110
 PostgreSQL, 110
 SQL Server, 112–117
ALTER ROLE statement, 61
 Oracle, 136
 PostgreSQL, 137
 SQL Server, 138
ALTER SCHEMA statement, 61, 141
ALTER SESSION statement, 403
ALTER TABLE statement, 61, 142–191
 MySQL, 148–156
 Oracle, 156–185
 PostgreSQL, 185–187
 SQL Server, 187–191
ALTER TRIGGER statement, 61, 191–204
 Oracle, 195–199
 PostgreSQL, 199
 SQL Server, 200–204
ALTER TYPE statement, 61, 204–216
 Oracle, 207–213
 PostgreSQL, 213–215
ALTER VIEW statement, 61, 216–225
 MySQL, 220
 Oracle, 220–223
 SQL Server, 223–225
ampersand (&), bitwise AND operator, 24
analytic functions (see window functions)
AND operator, 25, 357
angle brackets (see left angle bracket;
 right angle bracket)
ANSI standard for SQL (see SQL standard)
ANY operator, 25, 60, 63–65, 358
Any Type datatypes, Oracle, 39

APPENDCHILDXML function,
 Oracle, 500
APP_NAME function, SQL Server, 535
AREA function, PostgreSQL, 524
arithmetic operators, 24
ARRAY_APPEND function,
 PostgreSQL, 524
ARRAY_CAT function,
 PostgreSQL, 525
ARRAY_DIMS function,
 PostgreSQL, 525
ARRAY_LOWER function,
 PostgreSQL, 525
ARRAY_PREPEND function,
 PostgreSQL, 525
ARRAY_TO_STRING function,
 PostgreSQL, 525
ARRAY_UPPER function,
 PostgreSQL, 525
ASCII function
 MySQL, 484
 Oracle, 500
 PostgreSQL, 525
 SQL Server, 535
ASCIISTR function, Oracle, 500
ASIN function
 MySQL, 484
 Oracle, 500
 PostgreSQL, 525
 SQL Server, 536
assertions, 5
assignment operator (=), 24
assignment operator (:=), Oracle, 24
asterisk (*), multiplication arithmetic
 operator, 24, 27
at sign (@), prefixing parameters, SQL
 Server, 114
ATAN function
 MySQL, 484
 Oracle, 501
 PostgreSQL, 525
 SQL Server, 536
ATAN2 function
 MySQL, 484
 Oracle, 501
 PostgreSQL, 525
ATN2 function, SQL Server, 536
attributes (see columns)
AuthorizationID (user), 5
AVG DISTINCT function, 358
AVG function, 358, 440

B

BEGIN statement, PostgreSQL, 412
BEGIN TRANSACTION statement,
 SQL Server, 413
BENCHMARK function, MySQL, 484
BETWEEN operator, 25, 60, 65, 357
BFILE datatype, Oracle, 32, 39
BFILENAME function, Oracle, 501
BIGINT datatype, 32
 MySQL, 35
 PostgreSQL, 42
 SQL Server, 46
BIGSERIAL datatype, PostgreSQL, 33,
 42, 45
BIN function, MySQL, 484
BINARY datatype, 32
 MySQL, 35
 SQL Server, 46
binary datatypes, 30
BINARY function, MySQL, 484
BINARY_CHECKSUM function, SQL
 Server, 536
BINARY_DOUBLE datatype,
 Oracle, 32, 39
BINARY_FLOAT datatype, Oracle,
 32, 39
Bindings:1999 section, SQL99, 10
BIN_TO_NUM function, Oracle, 501
BIT datatype, 32
 deleted from SQL99, 10
 MySQL, 35
 PostgreSQL, 42
 SQL Server, 46
BIT VARYING datatype
 deleted from SQL99, 10
 PostgreSQL, 32, 42
BIT_AND function
 MySQL, 484
 PostgreSQL, 525
BITAND function, Oracle, 501
BIT_COUNT function, MySQL, 484
BIT_LENGTH function, 468
BIT_OR function
 MySQL, 485
 PostgreSQL, 525
bitwise AND operator (&), 24
bitwise exclusive OR operator (^), 24
bitwise NOT unary operator (~), 26
bitwise operators, 24
bitwise OR operator (|), 24
BIT_XOR function, MySQL, 485

BLOB datatype, 32
MySQL, 35
Oracle, 39
books and publications
“A Relational Model of Data for Large Shared Data Banks” (Codd), 2
(see also SQL standard)
BOOL datatype, 32
MySQL, 35
PostgreSQL, 42
BOOL_AND function, PostgreSQL, 526
BOOLEAN datatype, 30, 32, 42
Boolean literals, 22
BOOL_OR function, PostgreSQL, 526
BOX datatype, PostgreSQL, 32, 43
BOX function, PostgreSQL, 526
brackets ([...])
 enclosing delimited identifiers, 20
 wildcard operator, matches in set, SQL Server, 319
brackets, caret ([^...]), wildcard operator, matches not in set, SQL Server, 319
BROADCAST function, PostgreSQL, 526
BTRIM function, PostgreSQL, 526
built-in scalar functions, 463
BYTEA datatype, PostgreSQL, 32, 43

C

CALL statement, 61, 66–68
Call-Level Interfaace (CLI), package containing, 12
candidate key (see UNIQUE constraints)
Cantor, Georg (set theory developed by), 8
CARDINALITY function, Oracle, 501
caret (^), bitwise exclusive OR operator, 24
CASE function, 446–466
case of identifiers, 18
case sensitivity of identifiers, 21
CAST function, 466
catalogs, 4, 17
CBRT function, PostgreSQL, 526
CEIL function, 469
CENTER function, PostgreSQL, 526
CHAR datatype, 32
MySQL, 36
Oracle, 40
PostgreSQL, 43
SQL Server, 47
CHAR FOR BIT DATA datatype, 32
CHAR function
MySQL, 485
PostgreSQL, 526
SQL Server, 536
CHAR VARYING datatype, 34
Oracle, 41
SQL Server, 50
CHARACTER datatype, 32
Oracle, 40
PostgreSQL, 43
SQL Server, 47
character sets, 5
character string datatypes, 30
character string literals, 22
CHARACTER VARYING datatype, 34
Oracle, 41
PostgreSQL, 46
SQL Server, 50
CHARACTER_LENGTH function, PostgreSQL, 526
CHARINDEX function, SQL Server, 536
CHAR_LENGTH function, 468, 526
CHARSET function, MySQL, 485
CHARTOROWID function, Oracle, 501
CHECK constraints, 57
CHECKSUM function, SQL Server, 536
CHECKSUM_AGG function, SQL Server, 536
CHR function, Oracle, 501
CIDR datatype, PostgreSQL, 32, 43
CIRCLE datatype, PostgreSQL, 32, 43
CIRCLE function, PostgreSQL, 526
classes (see statement classes)
CLI (Call-Level Interface), package containing, 12
CLOB datatype, Oracle, 32, 40
CLOCK_TIMESTAMP function, PostgreSQL, 527
CLOSE CURSOR statement, 61, 68–70
CLUSTER_ID function, Oracle, 501
CLUSTER_PROBABILITY function, Oracle, 501
clusters, 4, 17
CLUSTER_SET function, Oracle, 501
COALESCE function
MySQL, 485
Oracle, 501

PostgreSQL, 527
SQL Server, 536
Codd, E. F.
 relational database rules by, 2–9
 relational database theory by, 2
 research by, leading to SEQUEL, 1
code examples, permission to use, x
COERCIBILITY function, MySQL, 485
COLLATION function, MySQL, 485
collations, 5
COLLECT function, Oracle, 501
collection datatypes, 30
COL_LENGTH function, SQL
 Server, 536
COL_NAME function, SQL Server, 537
colon, equal sign (:=), assignment
 operator, Oracle, 24
column-level constraints, 50
columns, 2, 4
 assertions for, 5
 constraints for, 5, 50–57, 144
 datatypes of, 5, 29
comma (,), list item separator, 28
comments, 29
COMMIT statement, 61, 70–73
comparison operators, 24, 357
COMPOSE function, Oracle, 501
composite (concatenated) indexes, 118, 119
COMPRESS function, MySQL, 485
compressed tables, Oracle, 170
COMPUTE clause, SQL Server, 388
CONCAT function, Oracle, 502
concatenated indexes, 118, 119
concatenation operator (||), 477
concatenation operator (+), SQL
 Server, 27
CONCAT_WS function, MySQL, 485
CONNECT BY clause, Oracle, 373
CONNECT statement, 61, 73–75
connection statements, 14
CONNECTION_ID function,
 MySQL, 485
constraints, 5, 50–57, 144
 CHECK, 57
 FOREIGN KEY, 52–55
 PRIMARY KEY, 51
 scope of, 50
 setting (see SET CONSTRAINT
 statement)
 syntax for, 50
 UNIQUE, 56
CONTAINS function, SQL Server, 537
CONTAINSTABLE function, SQL
 Server, 537
control statements, 14
CONV function, MySQL, 485
CONVERT function, 478
 Oracle, 502
 SQL Server, 537
CORR function, 441
correlated subqueries, 414, 415
CORR_K function, Oracle, 502
CORR_S function, Oracle, 502
COS function
 MySQL, 486
 Oracle, 502
 PostgreSQL, 527
 SQL Server, 537
COSH function, Oracle, 502
COT function
 MySQL, 486
 Oracle, 502
 PostgreSQL, 527
 SQL Server, 537
COUNT DISTINCT function, 359
COUNT function, 358, 441
COUNT_BIG function, SQL
 Server, 537
COVAR_POP function, 442
COVAR_SAMP function, 443
covering index, 118
covering query, 118
CRC32 function, MySQL, 486
CREATE DATABASE statement,
 61, 75–101
 MySQL, 76
 Oracle, 76–91
 PostgreSQL, 91
 SQL Server, 93–101
CREATE DOMAIN statement, 60
CREATE FUNCTION statement,
 61, 101–117
 MySQL, 106
 Oracle, 107–110
 PostgreSQL, 110
 SQL Server, 112–117
CREATE INDEX statement, 61, 117–133
 MySQL, 119
 Oracle, 120–128
 PostgreSQL, 128–130
 SQL Server, 130–133
CREATE METHOD statement,
 61, 105, 133

CREATE PROCEDURE statement, 61, 101–117
MySQL, 106
Oracle, 107–110
PostgreSQL, 110
SQL Server, 112–117
CREATE ROLE statement, 61, 135–139
CREATE SCHEMA statement, 61, 139–142
CREATE TABLE statement, 61, 142–191
MySQL, 148–156
Oracle, 156–185
PostgreSQL, 185–187
SQL Server, 187–191
CREATE TRIGGER statement, 61, 191–204
MySQL, 195
Oracle, 195–199
PostgreSQL, 199
SQL Server, 200–204
CREATE TYPE statement, 61, 204–216
Oracle, 207–213
PostgreSQL, 213–215
SQL Server, 215
CREATE VIEW statement, 61, 216–225
MySQL, 220
Oracle, 220–223
PostgreSQL, 223
SQL Server, 223–225
cross joins, 309, 311, 356
CUBE subclause, 360
CUBE_TABLE function, Oracle, 502
CUME_DIST function, 444, 459
CURDATE function, MySQL, 486
CURRENT_DATE function, 463
CURRENT_TIME function, 463
CURRENT_TIMESTAMP function, 463
CURRENT_USER function, 463
CURRVAL function, PostgreSQL, 527
CURSOR datatype, SQL Server, 32, 47
cursors, 226, 227–229, 261
closing (see CLOSE CURSOR statement)
creating (see DECLARE CURSOR statement)
opening (see OPEN statement)
retrieving records from (see FETCH statement)
CURTIME function, MySQL, 486
CV function, Oracle, 502

D

dashes (–), single-line comment, 29
Data Control Language (DCL), 13
Data Definition Language (DDL), 13
Data Manipulation Language (DML), 13
data statements, 14
data structures, 2, 3–5
database, 17
altering (see ALTER DATABASE statement)
connecting to (see CONNECT statement)
connections, switching between (see SET CONNECTION statement)
creating (see CREATE DATABASE statement)
design of, 142
disconnecting from (see DISCONNECT statement)
normalization of, 142
DATABASE function, MySQL, 486
database links, Oracle, 21
DATABASEPROPERTYEX function, SQL Server, 537
DATALENGTH function, SQL Server, 537
DATALINK datatype, 32
datalink datatype, 30
DATAOBJ_TO_PARTITION function, Oracle, 502
datatypes, 5, 29
MySQL, 35–38
Oracle, 39–42
platform comparisons of, 31–34
PostgreSQL, 42–46
SQL Server, 46–50
SQL2003, 30–31
date and time datatypes
arithmetic operations on, 24
list of, 31
DATE datatype, 32
MySQL, 36
Oracle, 40
PostgreSQL, 43
SQL Server, 47
date literals, 22
DATE_ADD function, MySQL, 486
DATEADD function, SQL Server, 537
DATEDIFF function, SQL Server, 538

DATE_FORMAT function,
 MySQL, 486

DATENAME function, SQL Server, 538

DATE_PART function,
 PostgreSQL, 527

DATEPART function, SQL Server, 538

DATE_SUB function, MySQL, 486

DATETIME datatype, 32
 MySQL, 36
 SQL Server, 47

DATETIME2 datatype, SQL Server, 47

DATETIMEOFFSET datatype, SQL
 Server, 32, 47

DATE_TRUNC function,
 PostgreSQL, 527

DAY function, SQL Server, 538

DAYNAME function, MySQL, 487

DAYOFMONTH function,
 MySQL, 487

DAYOFWEEK function, MySQL, 488

DAYOFYEAR function, MySQL, 488

DBCLOB datatype, 32

DB_ID function, SQL Server, 538

DB_NAME function, SQL Server, 538

DBTIMEZONE function, Oracle, 502

DCL (Data Control Language), 13

DDL (Data Definition Language), 13

DEC datatype, 32, 47

DECIMAL datatype, 32
 MySQL, 36
 Oracle, 40
 PostgreSQL, 43, 44
 SQL Server, 47

declarative processing, 7

DECLARE CURSOR statement, 61, 226–234
 MySQL, 229
 Oracle, 229
 PostgreSQL, 230, 326
 SQL Server, 231–234

DECODE function
 MySQL, 488
 Oracle, 503
 PostgreSQL, 527

DECOMPOSE function, Oracle, 503

DEFAULT function, MySQL, 488

DEGREES function
 MySQL, 488
 PostgreSQL, 527
 SQL Server, 538

DELETE privilege, 266

DELETE statement, 61, 234–242

DELETEXML function, Oracle, 503

delimited (quoted) identifiers, 19–21, 28

delimiters, system, 27

DENSE_RANK function, 445, 460

DEPTH function, Oracle, 503

DEREF function, Oracle, 503

DES_DECRYPT function, MySQL, 488

DES_ENCRYPT function, MySQL, 488

deterministic functions, 437

diagnostic statements, 14

dialects, SQL, 14

DIAMETER function, PostgreSQL, 527

DIFFERENCE function, SQL
 Server, 538

DISCONNECT statement, 61, 242

distributed partitioned views, SQL
 Server, 225

division arithmetic operator (/), 24, 27

DML (Data Manipulation
 Language), 13

DOUBLE datatype, 32, 36

DOUBLE PRECISION datatype, 32
 MySQL, 36
 Oracle, 40
 PostgreSQL, 43
 SQL Server, 47

double quotes ("..."), enclosing delimited
 identifiers, 19, 20, 28

DROP DATABASE statement, 61
 MySQL, 246
 Oracle, 247
 PostgreSQL, 250
 SQL Server, 251

DROP DOMAIN statement,
 PostgreSQL, 61, 250

DROP FUNCTION statement, 61
 MySQL, 246
 Oracle, 247
 PostgreSQL, 250
 SQL Server, 252

DROP INDEX statement, 62
 MySQL, 246
 Oracle, 248
 PostgreSQL, 250
 SQL Server, 252

DROP METHOD statement, 62

DROP PROCEDURE statement, 62
 MySQL, 246
 Oracle, 248
 SQL Server, 252

DROP ROLE statement, 62
 Oracle, 248
 PostgreSQL, 250
 SQL Server, 252
DROP SCHEMA statement, 62
 MySQL, 246
 PostgreSQL, 250
 SQL Server, 252
DROP statements, 244–253
DROP TABLE statement, 62
 MySQL, 247
 Oracle, 248
 PostgreSQL, 250
 SQL Server, 252
DROP TRIGGER statement, 62
 MySQL, 247
 Oracle, 249
 PostgreSQL, 251
 SQL Server, 253
DROP TYPE statement, 62
 Oracle, 249
 PostgreSQL, 251
 SQL Server, 253
DROP VIEW statement, 62
 MySQL, 247
 Oracle, 249
 PostgreSQL, 251
 SQL Server, 253
DUMP function, Oracle, 503

E

ELT function, MySQL, 488
EMPTY_BLOB function, Oracle, 503
EMPTY_CLOB function, Oracle, 503
ENCODE function
 MySQL, 488
 PostgreSQL, 527
ENCRYPT function, MySQL, 488
entities (see tables)
ENUM datatype, 32, 36
equal sign (=)
 assignment operator, 24
 comparison operator, 25, 27
EVERY function, PostgreSQL, 527
examples, permission to use, xii
EXCEPT set operator, 62, 254–257
exclamation point, equals sign (!=), not
 equal to operator, 25, 27
exclamation point, left angle bracket
 (!<), not less than operator, 25

exclamation point, right angle bracket
 (!>), not greater than
 operator, 25
EXECUTE privilege, 266
EXECUTE statement, 67
EXISTS operator, 25, 62, 64, 258, 357
EXISTSNODE function, Oracle, 503
EXP function, 469
explicit transactions, 70
EXPORT_SET function, MySQL, 488
eXtensible Markup Language (XML)
 in SQL2006, 10
 in SQL3, 12, 13
EXTRACT function, 470–472, 503
EXTRACTVALUE function
 MySQL, 489
 Oracle, 504

F

FEATURE_ID function, Oracle, 504
FEATURE_SET function, Oracle, 504
FEATURE_VALUE function,
 Oracle, 504
FETCH statement, 62, 259–265
FIELD function, MySQL, 489
FILEGROUP_ID function, SQL
 Server, 538
FILEGROUP_NAME function, SQL
 Server, 539
FILEGROUPPROPERTY function, SQL
 Server, 539
FILE_ID function, SQL Server, 538
FILE_NAME function, SQL Server, 538
FILEPROPERTY function, SQL
 Server, 539
files (devices), SQL Server, 97
filtering checks, 64
FIND_IN_SET function, MySQL, 489
FIRST function, Oracle, 504
FIRST_VALUE function, Oracle, 504
flashback queries, Oracle, 373, 376
FLOAT datatype, 32
 MySQL, 36
 Oracle, 40
 SQL Server, 47, 48
FLOAT function, PostgreSQL, 528
FLOAT4 datatype, PostgreSQL, 32, 43
FLOAT4 function, PostgreSQL, 528
FLOAT8 datatype, PostgreSQL, 32, 43
FLOOR function, 473
FOR XML clause, SQL Server, 390

FOREIGN KEY constraints, 52–55
FORMAT function, MySQL, 489
FORMATMESSAGE function, SQL Server, 539
Foundation:1999 section, SQL99, 10
FOUND_ROWS function, MySQL, 489
FREETEXT function, SQL Server, 539
FREETEXTTABLE function, SQL Server, 539
FROM_DAYS function, MySQL, 489
FROM_TZ function, Oracle, 504
FROM_UNIXTIME function, MySQL, 489
full joins, 309, 313, 356
FULLTEXTCATALOGPROPERTY function, SQL Server, 539
FULLTEXTSERVICEPROPERTY function, SQL Server, 539
functions, 437
aggregate, 358, 438–454
deterministic, 437
MySQL supported, 483–500
nondeterministic, 437
Oracle supported, 500–524
PostgreSQL supported, 524–535
scalar, 438, 463–483
SQL Server supported, 535–546
window, 438, 455–462
(see also specific functions)
functions, user-defined (see user-defined function (UDF))

G

GEOGRAPHY datatype, 32
GEOMETRY datatype, 32
GET DIAGNOSTIC statement, 67
GETANSINULL function, SQL Server, 539
GETDATE function, SQL Server, 539
GET_LOCK function, MySQL, 490
GETUTCDATE function, SQL Server, 539
GRANT statement, 62, 265–290
MySQL, 268–273
Oracle, 273–282
PostgreSQL, 282–284
SQL Server, 284–290
GRAPHIC datatype, 32
greater than operator ($>$), 25, 28
greater than or equal to operator (\geq), 25, 28

GREATEST function
MySQL, 490
Oracle, 505
GROUP BY clause, 358–364
GROUP_CONCAT function, MySQL, 490
GROUP_ID function, Oracle, 505
GROUPING function
Oracle, 505
SQL Server, 540
GROUPING SETS subclause, 360–364
GROUPING_ID function, Oracle, 505

H

HAVING clause, 364
heaps, Oracle, 181
HEIGHT function, PostgreSQL, 528
HEX function, MySQL, 490
HEXTORAW function, Oracle, 505
hierachic queries, Oracle, 373
HIERARCHYID datatype, SQL Server, 32, 47
HOST function, PostgreSQL, 528
HOST_ID function, SQL Server, 540
HOST_NAME function, SQL Server, 540
HOUR function, MySQL, 490

I

IDENT_CURRENT function, SQL Server, 540
identifier qualifier separator (.), 28
identifiers, 17, 17–22
case sensitivity of, 21
characters allowed in, 20
delimited (quoted), 19, 20, 21, 28
naming conventions for, 18
reserved, 20
rules for, 18, 19–22
size of, 19
uniqueness of, 21
IDENT_INCR function, SQL Server, 540
IDENTITY function, SQL Server, 540
IDENT_SEED function, SQL Server, 540
IF function, MySQL, 490
IFNULL function, MySQL, 490
IMAGE datatype, SQL Server, 32, 48
implicit transactions, 70
IN operator, 25, 62, 290–292, 358
INDEX_COL function, SQL Server, 540

indexes, 117, 118
altering (see *ALTER INDEX* statement)
concatenated, 119
creating (see *CREATE INDEX* statement)
index-organized table (IOT), Oracle, 127
INDEXPROPERTY function, SQL Server, 540
INET datatype, PostgreSQL, 32, 43
INET_ATON function, MySQL, 490
INET_NTOA function, MySQL, 490
INITCAP function
 Oracle, 505
 PostgreSQL, 528
INIT.ORA file, Oracle, 90
inner joins, 309, 311, 356
INSERT function, MySQL, 490
INSERT privilege, 266
INSERT statement, 62, 292–304
 MySQL, 295
 Oracle, 296–300
 PostgreSQL, 301
 SQL Server, 301–304
 (see also *MERGE* statement)
INSERTCHILDXML function, Oracle, 505
INSERTXMLBEFORE function, Oracle, 506
instances, 17
INSTR function
 MySQL, 491
 Oracle, 506
INT datatype, 32
 MySQL, 37
 PostgreSQL, 44
 SQL Server, 48
INT2 datatype, 32
INT4 datatype, PostgreSQL, 32, 44
INT8 datatype, PostgreSQL, 42
INTEGER datatype, 32
 MySQL, 37
 Oracle, 40
 PostgreSQL, 44
INTEGER function, PostgreSQL, 528
INTERSECT set operator, 62, 255, 304–307
interval datatype, 30
INTERVAL datatype, PostgreSQL, 32, 44
INTERVAL DAY TO SECOND
 datatype, 33, 40
INTERVAL function
 MySQL, 491
 PostgreSQL, 528
INTERVAL YEAR TO MONTH
 datatype, 33, 40
IOT (index-organized table), Oracle, 127
IS NULL or *IS NOT NULL*
 condition, 357
IS operator, 62, 307
ISCLOSED function, PostgreSQL, 528
ISDATE function, SQL Server, 541
ISFINITE function, PostgreSQL, 528
IS_FREE_LOCK function, MySQL, 491
IS_MEMBER function, SQL Server, 541
ISNULL function
 MySQL, 491
 SQL Server, 541
ISNUMERIC function, SQL Server, 541
isolation levels (see *SET TRANSACTION* statement; *START TRANSACTION* statement)
ISOPEN function, PostgreSQL, 528
IS_SRVROLEMEMBER function, SQL Server, 541
IS_USED_LOCK function, MySQL, 491
ITERATION_NUMBER function, Oracle, 506

J

Java Routines and Types (JRT), package containing, 13
JOIN subclause, 9, 62, 308–316, 355
 MySQL, 368
 Oracle, 374
 PostgreSQL, 383
joins, 8, 9, 355
 aliases and, 354
 cross joins, 309, 311, 356
 full joins, 309, 313, 356
 inner joins, 309, 311, 356
 left joins, 309, 312, 356
 natural joins, 313
 outer joins, 309, 312, 356
 partitioned outer joins, 375
 right joins, 309, 312, 356
 straight joins, 366
 theta joins, 9
 union joins, 309

JRT (Java Routines and Types), package containing, 13

JUSTIFY_DAYS function, PostgreSQL, 528

JUSTIFY_HOURS function, PostgreSQL, 528

JUSTIFY_INTERVAL function, PostgreSQL, 529

K

keywords, 17, 29, 547–554

- MySQL, 549
- Oracle, 551
- shared, 547
- SQL Server, 553
- SQL2003, 547

L

LAG function, Oracle, 506

LAST function, Oracle, 506

LAST_DAY function

- MySQL, 491
- Oracle, 507

LAST_INSERT_ID function, MySQL, 491

LAST_VALUE function, Oracle, 507

LCASE function, MySQL, 491

LEAD function, Oracle, 507

LEAST function

- MySQL, 491
- Oracle, 507

left angle bracket (<), less than operator, 25, 28

left angle bracket, equal sign (==), less than or equal to operator, 25, 28

left angle bracket, right angle bracket (<>), not equal to operator, 25, 27

LEFT function

- MySQL, 491
- SQL Server, 541

left joins, 309, 312, 356

LEN function, SQL Server, 541

LENGTH function

- MySQL, 491
- Oracle, 507
- PostgreSQL, 529

LENGTHB function, Oracle, 507

less than operator (<), 25, 28

less than or equal to operator (<=), 25, 28

levels of conformance, 11

LIKE operator, 25, 62, 316–320, 357

LINE datatype, PostgreSQL, 33, 44

list item separator (,), 28

literals, 17, 22

LN function, 473

LNNVL function, Oracle, 507

LOAD_FILE function, MySQL, 492

local partitioned views, SQL Server, 225

LOCALTIMESTAMP function, Oracle, 508

LOCATE function, MySQL, 492

LOG function

- MySQL, 492
- Oracle, 508
- PostgreSQL, 529
- SQL Server, 541

LOG10 function

- MySQL, 492
- SQL Server, 541

LOG2 function, MySQL, 492

logfiles

- Oracle, 90
- SQL Server, 97

logical operators, 25

LONG datatype, Oracle, 33, 40

LONG RAW datatype, Oracle, 33, 40

LONG VARCHAR datatype, 33

LONGVARGRAPHIC datatype, 33

LONGBLOB datatype, MySQL, 33, 37

LONGTEXT datatype, MySQL, 33, 37

LOWER function, 480

LPAD function

- MySQL, 492
- Oracle, 508
- PostgreSQL, 529

LSEG datatype, PostgreSQL, 33, 44

LSEG function, PostgreSQL, 529

LTRIM function

- MySQL, 492
- Oracle, 508
- PostgreSQL, 529
- SQL Server, 541

M

MACADDR datatype, PostgreSQL, 33, 44
MAKEDATE function, MySQL, 492
MAKE_REF function, Oracle, 508
MAKE_SET function, MySQL, 492
MAKETIME function, MySQL, 493
Management of External Data (MED),
 package containing, 12
MASKLEN function, PostgreSQL, 529
MATCH function, MySQL, 493
MAX function, 359, 446
MD5 function
 MySQL, 493
 PostgreSQL, 529
MED (Management of External Data),
 package containing, 12
MEDIAN function, Oracle, 508
MEDIUMBLOB datatype, MySQL, 33, 37
MEDIUMINT datatype, MySQL, 33, 37
MEDIUMTEXT datatype, MySQL, 33, 37
MERGE statement, 62, 320–324
metadata, 6
methods, 105, 134
 altering (see ALTER METHOD
 statement)
 creating (see CREATE METHOD
 statement)
MICROSECOND function,
 MySQL, 493
Microsoft SQL Server (see SQL Server)
MID function, MySQL, 493, 497
MIN function, 359, 446
MINUS set operator, Oracle, 256
minus sign (–)
 negative unary operator, 26
 range indicator in CHECK
 constraint, 27
 subtraction arithmetic operator,
 24, 27
MINUTE function, MySQL, 493
MOD function, 474
MODEL clause, Oracle, 377–381
modula arithmetic operator (%), 24
MONEY datatype, 33
 PostgreSQL, 44
 SQL Server, 48
MONTH function
 MySQL, 493
 SQL Server, 541
MONTHNAME function, MySQL, 493
MONTHS_BETWEEN function,
 Oracle, 508
Multimedia datatypes, Oracle, 39
multiplication arithmetic operator (*),
 24, 27
MySQL, 16
 ALL/ANY/SOME operators, 64
 ALTER DATABASE statement, 76
 ALTER FUNCTION statement, 106
 ALTER PROCEDURE
 statement, 106
 ALTER TABLE statement, 148–156
 ALTER VIEW statement, 220
 bitwise operators, 24
 COMMIT statement, 71
 CREATE DATABASE statement, 76
 CREATE FUNCTION
 statement, 106
 CREATE INDEX statement, 119
 CREATE PROCEDURE
 statement, 106
 CREATE SCHEMA statement, 140
 CREATE TABLE
 statement, 148–156
 CREATE TRIGGER statement, 195
 CREATE VIEW statement, 220
datatypes, 35–38
DECLARE CURSOR statement, 229
DELETE statement, 235
DROP statements, 246
FETCH statement, 262
functions, 483–500
GRANT statement, 268–273
identifier rules, 19
INSERT statement, 295
JOIN subclause, 314, 368
keywords, 549
ORDER BY clause, 329
RETURN statement, 332
REVOKE statement, 337–339
ROLLBACK statement, 346
SELECT statement, 365–369
SET statement, 392
SET TRANSACTION
 statement, 406
SQL statements, 60–63

START TRANSACTION
 statement, 411
straight joins, 366
TRUNCATE TABLE statement, 420
UPDATE statement, 427
website resources for, viii

N

naming conventions, 18
NANVL function, Oracle, 508
NATIONAL CHAR datatype, 33
 Oracle, 41
 SQL Server, 48
NATIONAL CHAR VARYING
 datatype, 33
 Oracle, 40
 SQL Server, 48
NATIONAL CHARACTER
 datatype, 33
 Oracle, 41
 SQL Server, 48
NATIONAL CHARACTER VARYING
 datatype, 33
 Oracle, 40
 SQL Server, 48
NATIONAL TEXT datatype, SQL
 Server, 33, 48
natural joins, 313
NCHAR datatype, 33
 MySQL, 37
 Oracle, 41
 SQL Server, 48
NCHAR function
 Oracle, 508
 SQL Server, 542
NCHAR VARYING datatype, 33, 40
NCLOB datatype, 33, 41
negative unary operator (\neg), 26
nested table subqueries, 414, 415
nested tables, Oracle, 170
NETMASK function, PostgreSQL, 529
NETWORK function, PostgreSQL, 529
NEWID function, SQL Server, 542
NEW_TIME function, Oracle, 508
NEXT_DAY function, Oracle, 509
NEXTVAL function, PostgreSQL, 529
NLS_CHARSET_DECL_LEN function,
 Oracle, 509
NLS_CHARSET_ID function,
 Oracle, 509

NLS_CHARSET_NAME function,
 Oracle, 509
NLS_INITCAP function, Oracle, 510
NLS_LOWER function, Oracle, 510
NLSSORT function, Oracle, 510
NLS_UPPER function, Oracle, 510
nondeterministic functions, 437
normalization, 2, 142
not equal to operator (\neq), 25
not equal to operator (\neq), 25, 27
not greater than operator (\neq), 25
not less than operator (\neq), 25
NOT operator, 25, 357
NOW function, 463, 464, 493
NPOINTS function, PostgreSQL, 529
NTEXT datatype, SQL Server, 33, 48
NTILE function, Oracle, 510
NULL values, 5
NULLIF function
 MySQL, 493
 Oracle, 510
 PostgreSQL, 530
 SQL Server, 542
NUMBER datatype, 33, 41
NUMERIC datatype, 33
 MySQL, 37
 PostgreSQL, 43, 44
 SQL Server, 47, 48
numeric datatypes, 31
numeric functions, 11, 467–477
numeric literals, 22
NUMTODSINTERVAL function,
 Oracle, 510
NUMTOYMINTERVAL function,
 Oracle, 510
NVARCHAR datatype, 33
 MySQL, 37
 SQL Server, 48
NVARCHAR2 datatype, Oracle, 33, 41
NVL function, Oracle, 511
NVL2 function, Oracle, 511

O

Object Language Binding (OBJ),
 package containing, 12
OBJECT_ID function, SQL Server, 542
OBJECT_NAME function, SQL
 Server, 542
OBJECTPROPERTY function, SQL
 Server, 542

objects, 4, 17
OCT function, MySQL, 493
OCTET_LENGTH function, 468
OID datatype, PostgreSQL, 33, 44
OLAP (Online Analytical Processing) functions, 11
OLD_PASSWORD function, MySQL, 494
Online Analytical Processing (OLAP) functions, 11
OPEN function, SQL Server, 542
open source movement, vii
OPEN statement, 62, 324–326
OPENDATASOURCE function, SQL Server, 542
OPENQUERY function, SQL Server, 542
OPENROWSET function, SQL Server, 542
operators, 17, 23, 27
 ALL/ANY/SOME operators, 63–65
 arithmetic operators, 24
 assignment operators, 24
 BETWEEN operator, 65
 bitwise operators, 24
 comparison operators, 24
 concatenation operator (||), 477
 logical operators, 25
 precedence for, 26
 unary operators, 26
 in WHERE clause, 357
 wildcard operator (_), matches any character, 317
 wildcard operator (%), matches any string, 28, 317
 wildcard operator ([...]), matches in set, 319
 wildcard operator ([^...]), matches not in set, 319
OR operator, 25, 357
Oracle, 16
 ALL/ANY/SOME operators, 64
 ALTER DATABASE statement, 76–91
 ALTER FUNCTION statement, 107–110
 ALTER INDEX statement, 120–128
 ALTER PROCEDURE statement, 107–110
ALTER ROLE statement, 136
ALTER SESSION statement, 403
ALTER TABLE statement, 156–185
ALTER TRIGGER statement, 195–199
ALTER TYPE statement, 207–213
ALTER VIEW statement, 220–223
assignment operator (:=), 24
basics, 90
CALL statement, 67
COMMIT statement, 72
compressed tables, 170
CONNECT BY clause, 373
CONNECT statement, 74
CREATE DATABASE statement, 76–91
CREATE FUNCTION statement, 107–110
CREATE INDEX statement, 120–128
CREATE PROCEDURE statement, 107–110
CREATE ROLE statement, 136
CREATE SCHEMA statement, 140
CREATE TABLE statement, 156–185
CREATE TRIGGER statement, 195–199
CREATE TYPE statement, 207–213
CREATE VIEW statement, 220–223
database links, 21
datatypes, 39–42
DECLARE CURSOR statement, 229
DELETE statement, 237–239
DISCONNECT statement, 243
DROP statements, 247–249
FETCH statement, 262
flashback queries, 373, 376
functions, 500–524
GRANT statement, 273–282
heaps, 181
hierachic queries, 373
identifier rules, 19
index-organized table (IOT), 127
INIT.ORA file, 90
INSERT statement, 296–300
INTERSECT operator, 306
JOIN subclause, 315, 374
keywords, 551

LIKE operator, 318
logfiles, 90
MERGE statement, 322
MINUS set operator, 256
MODEL clause, 377–381
nested tables, 170
OPEN statement, 326
ORDER BY clause, 329
organized tables, 181
PARTITION clause, 373
partitioned outer joins, 375
partitioned tables, 170–181
primary databases, 90
RETURN statement, 333
REVOKE statement, 339–341
ROLLBACK statement, 347
SELECT statement, 369–381
SET CONSTRAINT statement, 397
SET ROLE statement, 398
SET TIME ZONE statement, 403
SET TRANSACTION
 statement, 407
spatial data, 39
SQL statements, 60–63
standby databases, 90
subqueries, 418
system privileges, 277–282
TRUNCATE TABLE statement, 420
UNION set operator, 422
UPDATE statement, 428–430
website resources for, x
window function syntax, 456
XMLType tables, 182

ORA_HASH function, Oracle, 511
ORD function, MySQL, 494
ORDER BY clause, 62, 327–330, 364
organized tables, Oracle, 181
outer joins, 309, 312, 356
OVERLAY function, 480

P

parentheses (...)), affecting operator
 precedence, 26, 28
PARSENAME function, SQL
 Server, 542
PARTITION clause, Oracle, 373
partitioned outer joins, Oracle, 375
partitioned tables, Oracle, 170–181

partitioned views, SQL Server, 225
partitioning clause, 456
PASSWORD function, MySQL, 494
PATH datatype, PostgreSQL, 33, 45
PATH function
 Oracle, 511
 PostgreSQL, 530
PATINDEX function, SQL Server, 542
PCLOSE function, PostgreSQL, 530
percent sign (%)
 modula arithmetic operator, 24
 wildcard operator, matches any
 string, 28, 317
PERCENTILE_CONT function, 448
PERCENTILE_DISC function, 449
PERCENT_RANK function, 447, 461
period (.), identifier qualifier
 separator, 28
PERIOD_ADD function, MySQL, 494
PERIOD_DIFF function, MySQL, 494
permissions, 5
PERMISSIONS function, SQL
 Server, 543
Persistent Stored Modules (PSM),
 package containing, 12, 15
PG_CONNECT statement,
 PostgreSQL, 74
PI function
 MySQL, 494
 PostgreSQL, 530
 SQL Server, 543
platform comparisons
 datatypes, 31–34
 SQL statements, 60–63
 (see also specific platforms)
platforms included in this book, 16
PL/pgSQL, 15
PL/SQL, 15
plus sign (+)
 addition arithmetic operator, 24, 27
 concatenation operator, SQL
 Server, 27
 positive unary operator, 26
POINT datatype, PostgreSQL, 33, 45
POINT function, PostgreSQL, 530
POLYGON datatype, PostgreSQL, 33, 45
POLYGON function, PostgreSQL, 530
POOPEN function, PostgreSQL, 531
POSITION function, 474, 492

positional deletes, 234
positive unary operator (+), 26
PostgreSQL, 16
 ALTER DATABASE statement, 91
 ALTER FUNCTION statement, 110
 ALTER INDEX statement, 128–130
 ALTER PROCEDURE
 statement, 110
 ALTER ROLE statement, 137
 ALTER SCHEMA statement, 141
 ALTER TABLE statement, 185–187
 ALTER TRIGGER statement, 199
 ALTER TYPE statement, 213–215
 BEGIN statement, 412
 bitwise operators, 24
 CLOSE CURSOR statement, 69
 COMMIT statement, 72
 CREATE DATABASE statement, 91
 CREATE FUNCTION
 statement, 110
 CREATE INDEX
 statement, 128–130
 CREATE PROCEDURE
 statement, 110
 CREATE ROLE statement, 137
 CREATE SCHEMA statement, 141
 CREATE TABLE
 statement, 185–187
 CREATE TRIGGER statement, 199
 CREATE TYPE statement, 213–215
 CREATE VIEW statement, 223
 datatypes, 42–46
 DECLARE CURSOR statement, 230
 DELETE statement, 239
 DROP statements, 250
 EXCEPT set operator, 257
 FETCH statement, 263
 functions, 524–535
 GRANT statement, 282–284
 identifier rules, 19
 INSERT statement, 301
 INTERSECT operator, 306
 keywords, 551
 LIKE operator, 319
 ORDER BY clause, 329
 PG_CONNECT statement, 74
 RETURN statement, 333
 REVOKE statement, 341
 ROLLBACK statement, 347
 SELECT statement, 381–384
 SET CONSTRAINT statement, 397
 SET SESSION AUTHORIZATION
 statement, 401
 SET statement, 393
 SET TIME ZONE statement, 404
 SET TRANSACTION
 statement, 408
 SPI_CONNECT statement, 74
 SQL statements, 60–63
 subqueries, 418
 TRUNCATE TABLE statement, 420
 UNION set operator, 423
 UPDATE statement, 430
 website resources for, viii
pound symbol (#)
 prefixing temporary procedures, SQL
 Server, 114
 prefixing temporary tables, SQL
 Server, 190
POW function
 MySQL, 494
 PostgreSQL, 531
POWER function, 475, 494
POWERMULTISET function,
 Oracle, 511
POWERMULTISET_BY_
 CARDINALITY function,
 Oracle, 511
PREDICTION_functions, Oracle, 511
prefixes for identifiers, 18
PRESENTNNV function, Oracle, 512
PRESENTV function, Oracle, 512
PREVIOUS function, Oracle, 512
primary databases, Oracle, 90
PRIMARY KEY constraints, 51
privileges
 assigning (see GRANT statement)
 list of, 266
 revoking (see REVOKE statement)
 roles for (see roles)
procedural programming, 7
procedures, stored (see stored
 procedures)
projections, 8
 (see also SELECT clause)
PSM (Persistent Stored Modules),
 package containing, 12, 15

Q

QUARTER function, MySQL, 494
queries, multiple (see set operators)
(see also SELECT statement)
QUOTE function, MySQL, 494
quoted identifiers (see delimited
identifiers)
QUOTE_IDENT function,
PostgreSQL, 531
QUOTE_LITERAL function,
PostgreSQL, 531
quotes (see double quotes; single
quotes)

R

RADIANS function
MySQL, 494
PostgreSQL, 531
SQL Server, 543
RADIUS function, PostgreSQL, 531
RAND function
MySQL, 495
SQL Server, 543
RANDOM function, PostgreSQL, 531
range indicator in CHECK constraint
(-), 27
RANK function, 449, 461
RATIO_TO_REPORT function,
Oracle, 512
RAW datatype, Oracle, 33, 41
RAWTOHEX function, Oracle, 512
RAWTONHEX function, Oracle, 512
RDBMS (Relational Database
Management System), 2
REAL datatype, 33
MySQL, 37
Oracle, 41
PostgreSQL, 43
SQL Server, 48
records (rows), 2, 4
deleting (see DELETE statement;
TRUNCATE TABLE
statement)
inserting (see INSERT statement)
querying (see SELECT statement)
updating (see UPDATE statement)
REF function, Oracle, 512
REFERENCES privilege, 266

REFTOHEX function, Oracle, 512
REGEXP function, MySQL, 495
REGEXP_INSTR function, Oracle, 512
REGEXP_MATCHES function,
PostgreSQL, 531
REGEXP_REPLACE function
Oracle, 513
PostgreSQL, 531
REGEXP_SPLIT_TO_ARRAY function,
PostgreSQL, 532
REGEXP_SPLIT_TO_TABLE function,
PostgreSQL, 532
REGEXP_SUBSTR function,
Oracle, 513
REGR_functions, 451
Relational Database Management
System (RDBMS), 2
relational databases
history of, 1
rules of, 2–9
“A Relational Model of Data for Large
Shared Data Banks” (Codd), 2
RELEASE SAVEPOINT statement, 62,
330
RELEASE_LOCK function,
MySQL, 495
RELTIME function, PostgreSQL, 532
REMAINDER function, Oracle, 514
REPEAT function, MySQL, 495
REPLACE function
MySQL, 495
Oracle, 514
SQL Server, 543
REPLICATE function, SQL Server, 543
reserved identifiers, 20
reserved words, 17, 29
RETURN statement, 62, 332–334
REVERSE function
MySQL, 495
SQL Server, 543
REVOKE statement, 62, 334–345
right angle bracket (>), greater than
operator, 25, 28
right angle bracket, equal sign (>=),
greater than or equal to
operator, 25, 28
RIGHT function
MySQL, 495
SQL Server, 543

right joins, 309, 312, 356
RLIKE function, MySQL, 495
roles, 135
 changing (see ALTER ROLE statement)
 creating (see CREATE ROLE statement)
 deleting (see DROP ROLE statement)
ROLLBACK statement, 62, 345–349
ROLLUP subclause, 360
ROUND function
 MySQL, 495
 Oracle, 514
 PostgreSQL, 532
 SQL Server, 543
routines (see stored procedures; user-defined functions (UDFs))
row processing, 7
ROW_COUNT function, MySQL, 495
ROWCOUNT_BIG function, SQL Server, 543
ROWID datatype, Oracle, 33, 41
ROWIDTOCHAR function, Oracle, 514
ROWIDTONCHAR function, Oracle, 514
ROW_NUMBER function, 462
rows (see records)
ROWVERSION datatype, SQL Server, 33, 48
RPAD function
 MySQL, 495
 Oracle, 514
 PostgreSQL, 532
RTRIM function
 MySQL, 495
 Oracle, 514
 PostgreSQL, 532
 SQL Server, 543

S

sampling, 11
SAVE statement, SQL Server, 350
SAVEPOINT statement, 62, 349–351
savepoints
 releasing (see RELEASE SAVEPOINT statement)
 rolling back to (see ROLLBACK statement)
 setting (see SAVEPOINT statement)

scalar aggregates, 359
scalar functions, 438, 463–483
 ABS, 467
 BIT_LENGTH, 468
 built-in, 463
 CASE, 464–466
 CAST, 466
 CEIL, 469
 CHAR_LENGTH, 468
 CONVERT, 478
 CURRENT_DATE, 463
 CURRENT_TIME, 463
 CURRENT_TIMESTAMP, 463
 CURRENT_USER, 463
 EXP, 469
 EXTRACT, 470–472
 FLOOR, 473
 LN, 473
 LOWER, 480
 MOD, 474
 NOW, 463, 464
 numeric, 467–477
 OCTET_LENGTH, 468
 OVERLAY, 480
 POSITION, 474
 POWER, 475
 SESSION_USER, 463
 SQRT, 475
 string, 477–483
 SUBSTRING, 481
 SYSTEM_USER, 463
 TRANSLATE, 478
 TRIM, 482
 UPPER, 480
 USER, 463
 WIDTH_BUCKET, 476
scalar subqueries, 414, 415
SCHEMA function, MySQL, 496
schema statements, 14
schemas, 4, 17, 139, 140
 creating (see CREATE SCHEMA statement)
 setting for unqualified objects (see SET PATH statement)
 setting (see SET SCHEMA statement)
Schemata package, 10, 12
SCN_TO_TIMESTAMP function, Oracle, 514
search deletes, 234
SECOND function, MySQL, 496

SEC_TO_TIME function, MySQL, 496
SELECT clause, 9
SELECT privilege, 266
SELECT statement, 62, 351–391
 Codd’s rules in, examples of, 8
 GROUP BY clause, 358–364
 HAVING clause, 364
 JOIN subclause, 9, 62, 308–316, 355
 MySQL, 365–369
 Oracle, 369–381
 ORDER BY clause, 62, 327–330, 364
 PostgreSQL, 381–384
 SQL Server, 384–391
 WHERE clause (see WHERE clause)
selections, 8, 9
 (see also WHERE clause)
SEQUEL (Structured English Query Language), 1
SERIAL datatype, 33
 MySQL, 38
 PostgreSQL, 45
SERIAL4 datatype, 33, 45
SERIAL8 datatype, PostgreSQL, 33, 45
session statements, 14
sessions, 73
SESSIONTIMEZONE function,
 Oracle, 515
SESSION_USER function, 463, 496, 499
SET AUTHORIZATION statement, 62
SET CONNECTION statement, 62, 394
SET CONSTRAINT statement, 62, 396
SET datatype, MySQL, 33, 38
SET function, Oracle, 515
set operations, 7
set operators
 EXCEPT, 254–257
 INTERSECT, 255, 304–307
 MINUS, Oracle, 256
 UNION, 255, 421–424
SET PATH statement, 62, 397
SET ROLE statement, Oracle, 62, 398
SET SCHEMA statement, 62, 400
SET SESSION AUTHORIZATION
 statement, PostgreSQL, 401
SET statement, 62, 391–394
SET TIME ZONE statement,
 62, 402–404
SET TRANSACTION statement,
 63, 404–409
SET_MASKLEN function,
 PostgreSQL, 532
SETSEED function, PostgreSQL, 532
SETVAL function, PostgreSQL, 532
SHA function, MySQL, 496
SHA1 function, MySQL, 496
SIGN function
 MySQL, 496
 Oracle, 515
 PostgreSQL, 532
 SQL Server, 544
SIN function
 MySQL, 496
 Oracle, 515
 PostgreSQL, 533
 SQL Server, 544
single quotes (‘...’)
 enclosing delimited identifiers, 20
 enclosing string literals, 22, 28
SINH function, Oracle, 515
slash (/), division arithmetic
 operator, 24, 27
slash, asterisk /*...*/), enclosing
 multiline comments, 29
SLEEP function, MySQL, 496
SMALLDATETIME datatype, SQL
 Server, 33, 48
SMALLINT datatype, 34
 MySQL, 38
 Oracle, 41
 PostgreSQL, 45
 SQL Server, 49
SMALLMONEY datatype, SQL
 Server, 34, 49
SOME operator, 25, 60, 63–65, 358
sorting rules (see collations)
SOUNDEX function
 MySQL, 496
 Oracle, 515
 SQL Server, 544
SPACE function
 MySQL, 496
 SQL Server, 544
spatial data
 MySQL, 35
 Oracle, 39
SPL_CONNECT statement,
 PostgreSQL, 74
SQL (Structured Query Language)
 alternatives to, 7
 history of, 1

SQL Server, 17
 ALL/ANY/SOME operators, 65
 ALTER DATABASE
 statement, 93–101
 ALTER FUNCTION
 statement, 112–117
 ALTER INDEX statement, 130–133
 ALTER PROCEDURE
 statement, 112–117
 ALTER ROLE statement, 138
 ALTER TABLE statement, 187–191
 ALTER TRIGGER
 statement, 200–204
 ALTER VIEW statement, 223–225
 BEGIN TRANSACTION
 statement, 413
 bitwise operators, 24
 CLOSE CURSOR statement, 69
 COMMIT statement, 72
 COMPUTE clause, 388
 CONNECT statement, 74
 CREATE DATABASE
 statement, 93–101
 CREATE FUNCTION
 statement, 112–117
 CREATE INDEX
 statement, 130–133
 CREATE PROCEDURE
 statement, 112–117
 CREATE ROLE statement, 138
 CREATE SCHEMA statement, 141
 CREATE TABLE
 statement, 187–191
 CREATE TRIGGER
 statement, 200–204
 CREATE TYPE statement, 215
 CREATE VIEW statement, 223–225
 database roles, 289
 datatypes, 46–50
 DECLARE CURSOR
 statement, 231–234
 DELETE statement, 240–242
 DISCONNECT statement, 244
 DROP statements, 251–253
 EXCEPT set operator, 257
 FETCH statement, 264
 files (devices), 97
 FOR XML clause, 390
 functions, 535–546
 GRANT statement, 284–290
 identifier rules, 19
 INSERT statement, 301–304
 INTERSECT operator, 307
 JOIN subclause, 316
 keywords, 553
 LIKE operator, 319
 logfiles, 97
 MERGE statement, 323
 modula arithmetic operator (%), 24
 OPEN statement, 326
 ORDER BY clause, 330
 partitioned views, 225
 RETURN statement, 333
 REVOKE statement, 343
 ROLLBACK statement, 348
 SAVE statement, 350
 SELECT statement, 384–391
 SET CONNECTION statement, 395
 SET statement, 394
 SET TRANSACTION
 statement, 408
 SQL statements, 60–63
 subqueries, 418
 system privileges, 287–288
 system roles, 288
 temporary procedures, 114
 temporary tables, 190
 UNION set operator, 423
 UPDATE statement, 431–433
 website resources for, viii
 window function syntax, 456
SQL standard
 history of, 10
 levels of conformance to, 11
 sorting requirements of, 5
 (see also specific standards)
SQL statements, platform support
 for, 60–63
SQL2 (SQL92), 10, 11
SQL2006, 1, 10
SQL3 (SQL2003)
 compliance with, 13
 new features, 10
 statement classes, 13
 supplemental features packages, 11
SQL99, 10
SQL/CLI (Call-Level Interface)
 package, 12
SQL/Foundation package, 12
SQL/Framework package, 12

SQL/JRT (Java Routines and Types)
 package, 13

SQL/MED (Management of External Data) package, 12

SQL/OBJ (Object Language Binding) package, 12

SQL/PSM (Persistent Stored Modules) package, 12

SQL/PSM (Persistent Stored Module) package, 15

SQL/Schemata package, 12

SQL_VARIANT datatype, SQL Server, 34, 49

SQL/XML package, 13, 12

SQRT function, 475

square brackets (see brackets)

standby databases, Oracle, 90

START TRANSACTION statement, 63, 409–413

STATEMENT_TIMESTAMP function, PostgreSQL, 533

STATS_functions, Oracle, 515

STATS_DATE function, SQL Server, 544

STD function, MySQL, 496

STDDEV function
 MySQL, 496
 Oracle, 515

STDDEV_POP function, 452

STDDEV_SAMP function, 453
 Oracle, 515

STDEV function, SQL Server, 544

STDEVP function, SQL Server, 544

STDEV_POP function, Oracle, 515

stored procedures, 101, 104, 106
 altering (see ALTER PROCEDURE statement)
 calling (see CALL statement)
 creating (see CREATE PROCEDURE statement)
 return status of, 67
 returning from (see RETURN statement)

STR function, SQL Server, 544

straight joins, MySQL, 366

STRCMP function, MySQL, 497

string concatenation operator, 477

string functions, 477–483

string literals, 28

STRING_TO_ARRAY function, PostgreSQL, 533

STR_TO_DATE function, MySQL, 497

structured data (see data structures)

STUFF function, SQL Server, 544

SUBDATE function, MySQL, 486

subqueries, 63, 414–418

SUBSTR function, Oracle, 516

SUBSTRB function, Oracle, 516

SUBSTRING function, 481
 MySQL, 497
 PostgreSQL, 533
 SQL Server, 544

SUBSTRING_INDEX function, MySQL, 497

SUBTIME function, MySQL, 497

subtraction arithmetic operator (–), 24, 27

suffixes for identifiers, 18

SUM DISTINCT function, 359

SUM function, 359, 440

SUSER_ID function, SQL Server, 544

SUSER_SID function, SQL Server, 545

SUSER_SNAME function, SQL Server, 545

SYS_CONNECT_BY_PATH function, Oracle, 516

SYS_CONTEXT function, Oracle, 516

SYSDATE function
 MySQL, 493
 Oracle, 517

SYS_DBURIGEN function, Oracle, 516

SYS_EXTRACT_UTC function, Oracle, 516

SYS_GUID function, Oracle, 516

system delimiters, 27

SYSTEM_USER function, 463, 499

SYSTIMESTAMP function, Oracle, 517

SYS_TYPEID function, Oracle, 516

SYS_XMLAGG function, Oracle, 517

SYS_XMLGEN function, Oracle, 517

T

TABLE datatype, SQL Server, 34, 49

table-level constraints, 50

tables, 2, 3, 4, 142
 altering (see ALTER TABLE statement)
 creating (see CREATE TABLE statement)
 design of, 142

TABLESAMPLE clause, 11

TAN function
 MySQL, 497
 Oracle, 517
 PostgreSQL, 533
 SQL Server, 545

TANH function, Oracle, 517

temporal (date and time) datatypes
 arithmetic operations on, 24
 list of, 31

TEXT datatype, 34
 MySQL, 38
 PostgreSQL, 45
 SQL Server, 49

TEXT function, PostgreSQL, 533

TEXTPTR function, SQL Server, 545

TEXTVALID function, SQL Server, 545

theta joins, 9

tilde (~), bitwise NOT unary
 operator, 26

TIME datatype, 34
 MySQL, 38
 PostgreSQL, 45
 SQL Server, 49

time datatypes (see date and time
 datatypes)

TIME function, MySQL, 497

time zone, changing (see SET TIME
 ZONE statement)

TIMEDIFF function, MySQL, 498

TIME_FORMAT function,
 MySQL, 497

TIMEOFDAY function,
 PostgreSQL, 533

TIMESPAN datatype, 34

TIMESTAMP datatype, 34
 MySQL, 38
 Oracle, 41
 PostgreSQL, 46
 SQL Server, 49

TIMESTAMP function
 MySQL, 498
 PostgreSQL, 533

TIMESTAMP WITH TIME ZONE
 datatype, 34

TIMESTAMPADD function,
 MySQL, 498

TIMESTAMPDIFF function,
 MySQL, 498

TIMESTAMP_TO_SCN function,
 Oracle, 517

TIMESTAMPTZ datatype, 34

TIME_TO_SEC function, MySQL, 498

TIMETZ datatype, PostgreSQL, 34, 46

TINYBLOB datatype, 34, 38

TINYINT datatype, 34, 49

TINYTEXT datatype, MySQL, 34, 38

TO_BINARY_DOUBLE function,
 Oracle, 517

TO_BINARY_FLOAT function,
 Oracle, 517

TO_CHAR function
 Oracle, 518, 519
 PostgreSQL, 533

TO_CLOB function, Oracle, 519

TO_DATE function
 Oracle, 519
 PostgreSQL, 533

TO_DAYS function, MySQL, 498

TO_DSINTERVAL function,
 Oracle, 519

TO_LOB function, Oracle, 519

TO_MULTI_BYTE function,
 Oracle, 519

TO_NCHAR function, Oracle, 519

TO_NCLOB function, Oracle, 520

TO_NUMBER function
 Oracle, 520
 PostgreSQL, 535

TO_SINGLE_BYTE function,
 Oracle, 520

TO_TIMESTAMP function
 Oracle, 520
 PostgreSQL, 535

TO_TIMESTAMP_TZ function,
 Oracle, 520

TO_YMINTERVAL function,
 Oracle, 520

transaction statements, 14

transactions
 closing and making changes
 permanent (see COMMIT
 statement)
 implicit and explicit, 70
 savepoints in, releasing (see
 RELEASE SAVEPOINT
 statement)
 savepoints in, setting (see
 SAVEPOINT statement)
 settings for (see SET
 TRANSACTION statement)
 starting (see START
 TRANSACTION statement)
 undoing (see ROLLBACK statement)

TRANSACTION_TIMESTAMP
 function, PostgreSQL, 535
Transact-SQL, 15
TRANSLATE function, 478
 Oracle, 520
 PostgreSQL, 535
TREAT function, Oracle, 521
TRIGGER privilege, 266
triggers, 191, 193–195
 altering (see ALTER TRIGGER statement)
 creating (see CREATE TRIGGER statement)
TRIM function, 482
TRUNC function
 Oracle, 521
 PostgreSQL, 535
TRUNCATE function, MySQL, 498
TRUNCATE TABLE statement, 63, 235, 419–421
tuples (see records)
TYPEPROPERTY function, SQL Server, 545
TZ_OFFSET function, Oracle, 521

U

UCASE function, MySQL, 498
UDF (see user-defined function)
UDT (see user-defined type)
UID function, Oracle, 521
unary operators, 26
UNCOMPRESS function, MySQL, 498
UNCOMPRESS_LENGTH function, MySQL, 498
UNDER privilege, 266
underscore ()
 in identifiers, 18
 wildcard operator, 317
UNHEX function, MySQL, 498
UNICODE function, SQL Server, 546
UNION JOIN clause, deleted from SQL99, 10
union joins, 309
UNION set operator, 63, 255, 421–424
UNIQUE constraints, 56
UNIQUEIDENTIFIER datatype, SQL Server, 34, 49
UNISTR function, Oracle, 521
UNIX_TIMESTAMP function, MySQL, 498

UNSIGNED datatype attribute, MySQL, 35
UPDATE privilege, 266
UPDATE statement, 63, 424–433
 MySQL, 427
 Oracle, 428–430
 PostgreSQL, 430
 SQL Server, 431–433
 (see also MERGE statement)
UPDATE...SET ROW statement, deleted from SQL99, 10
UPDATEXML function
 MySQL, 499
 Oracle, 521
UPPER function, 480
UROWID datatype, Oracle, 34, 41
USAGE privilege, 266
user (AuthorizationID), 5
USER function, 463, 499
user-defined function (UDF), 101, 104, 106
 altering (see ALTER FUNCTION statement)
 creating (see CREATE FUNCTION statement)
 returning from (see RETURN statement)
user-defined type (UDT), 206
 altering (see ALTER TYPE statement)
 creating (see CREATE TYPE statement)
USERENV function, Oracle, 521
USER_ID function, SQL Server, 546
USER_NAME function, SQL Server, 546
UTC_DATE function, MySQL, 499
UTC_TIME function, MySQL, 499
UTC_TIMESTAMP function, MySQL, 499
UUID function, MySQL, 499

V

VALUE function, Oracle, 522
VAR function, SQL Server, 546
VARBINARY datatype, 34
 MySQL, 38
 SQL Server, 50
VARBIT datatype, PostgreSQL, 32, 42
VARCHAR datatype, 34
 Oracle, 41
 PostgreSQL, 46
 SQL Server, 50

VARCHAR FOR BIT DATA
datatype, 34
VARCHAR function, PostgreSQL, 535
VARCHAR2 datatype, Oracle, 34, 42
VARGRAPHIC datatype, 34
variables
declaring (see `DECLARE` statement)
setting (see `SET` statement)
VARIANCE function
MySQL, 499
Oracle, 522
VARP function, SQL Server, 546
VAR_POP function, 453, 522
VAR_SAMP function, 454, 522
vector aggregates, 359
VERSION function, MySQL, 499
vertical bar (|), bitwise OR operator, 24
views, 4, 7, 216, 218–220
virtual tables (see `views`)
VSIZE function, Oracle, 522

W

website resources
for this book, x
MySQL, viii
open source movement, v
Oracle, viii
PostgresSQL, viii
SQL Server, viii
WEEK function, MySQL, 499
WEEKDAY function, MySQL, 499
WEEKOFYEAR function, MySQL, 499
WHERE clause, 9, 63, 353, 356, 433–436
aliases and, 354
comparison operators in, 25
logical operators in, 25
theta joins in, 9
WIDTH function, PostgreSQL, 535
WIDTH_BUCKET function, 476
wildcard operator (_), matches any character, 317
wildcard operator (%), matches any string, 28, 317
wildcard operator ([...]), matches in set, SQL Server, 319
wildcard operator ([^...]), matches not in set, SQL Server, 319
window functions, 438, 455–462
aggregate functions as, 458
CUME_DIST, 444, 459

DENSE_RANK, 445, 460
framing clause, 458
Oracle syntax, 456
ordering clause, 457
partitioning clause, 456
PERCENT_RANK, 447, 461
RANK, 449, 461
ROW_NUMBER, 462
SQL Server syntax, 456

X

XML (eXtensible Markup Language)
in SQL2006, 10
in SQL3, 12, 13
XML datatype, 31, 34, 50
XMLAGG function, Oracle, 522
XMLCAST function, Oracle, 522
XMCDATA function, Oracle, 522
XMLCOLATTVAL function, Oracle, 522
XMLCONCAT function, Oracle, 523
XMLEDIFF function, Oracle, 523
XMLELEMENT function, Oracle, 523
XMLEXISTS function, Oracle, 523
XMLFOREST function, Oracle, 523
XMLPARSE function, Oracle, 523
XMLPATCH function, Oracle, 523
XMLPI function, Oracle, 523
XMLQUERY function, Oracle, 523
XMLROOT function, Oracle, 523
XMLSEQUENCE function, Oracle, 524
XMLSERIALIZE function, Oracle, 524
XMLTABLE function, Oracle, 524
XMLTRANSFORM function, Oracle, 524
XMLTYPE datatype, Oracle, 34, 42
XMLType tables, Oracle, 182
XOR function, MySQL, 500

Y

YEAR datatype, MySQL, 34, 38
YEAR function
MySQL, 500
SQL Server, 546
YEARWEEK function, MySQL, 500

Z

ZEROFILL datatype attribute, MySQL, 35

About the Authors

Kevin E. Kline is the director of SQL Server Solutions at Quest Software, a leading provider of award-winning tools for database management and application monitoring on the SQL Server platform. Kevin is also the president of the international Professional Association for SQL Server (PASS) and frequently contributes to database technology magazines, web sites, and discussion forums. He is the coauthor of *Transact-SQL Programming* (O'Reilly) and three other books on database technologies. Kevin is a top-rated speaker, appearing at international conferences such as Microsoft TechEd, DevTeach, PASS, Microsoft IT Forum, and SQL Connections.

Daniel Kline (Ph.D., Indiana University, 1997) specializes in Middle English literature and culture; Chaucer; literary and cultural theory; critical pedagogy; and digital medievalism. An associate professor of English at the University of Alaska, Anchorage, he is widely published in academic journals and is the author/webmaster of The Electronic Canterbury Tales (www.kankedort.net).

Brand Hunt is a director in architecture and engineering at Merrill Lynch. Beyond work, Brand enjoys board games, pinochle, and snowboarding with his family and friends.

Colophon

The animal on the cover of *SQL in a Nutshell*, Third Edition, is a chameleon. There are approximately 85 species of chameleons existing in the world today. They are mostly indigenous to Africa, although there are a few species found in Asia and in Europe. Most are tree dwellers. The chameleon is relatively small; the average adult size is between 6 and 12 inches. It lives mostly on insects and uses its long tongue to capture its prey. Indeed, the tongue is a critical tool. It can stretch up to 1.5 times the lizard's body length, and there is an adhesive pad on the end on which the insects are trapped. There are several other characteristics common to all species of chameleons. For example, its eyes are large and protruding, and the lizard can see 360 degrees without moving its head or body. Its toes are on either side of its feet, usually three on one side and two on the other. This is ideal for moving quickly and efficiently through tree branches.

Chameleons are best known for their ability to change their appearance to adapt to their physical environment. Actually, several types of reptiles can change their skin color, but the chameleon is far and away the most accomplished. This skill, which is moderated by the nervous system, obviously is invaluable for hunting prey and avoiding predators, and also helps to stabilize body temperature. The extent of this camouflage capability is related to the gender, age, and species of the lizard.

The cover image is from the Dover Pictorial Archive. The cover font is Adobe ITC Garamond. The text font is Linotype Birk; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans MonoCondensed.