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
- November 2021: Fourth Edition

Revision History for the Early Release

- 2021-07-22: First 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.

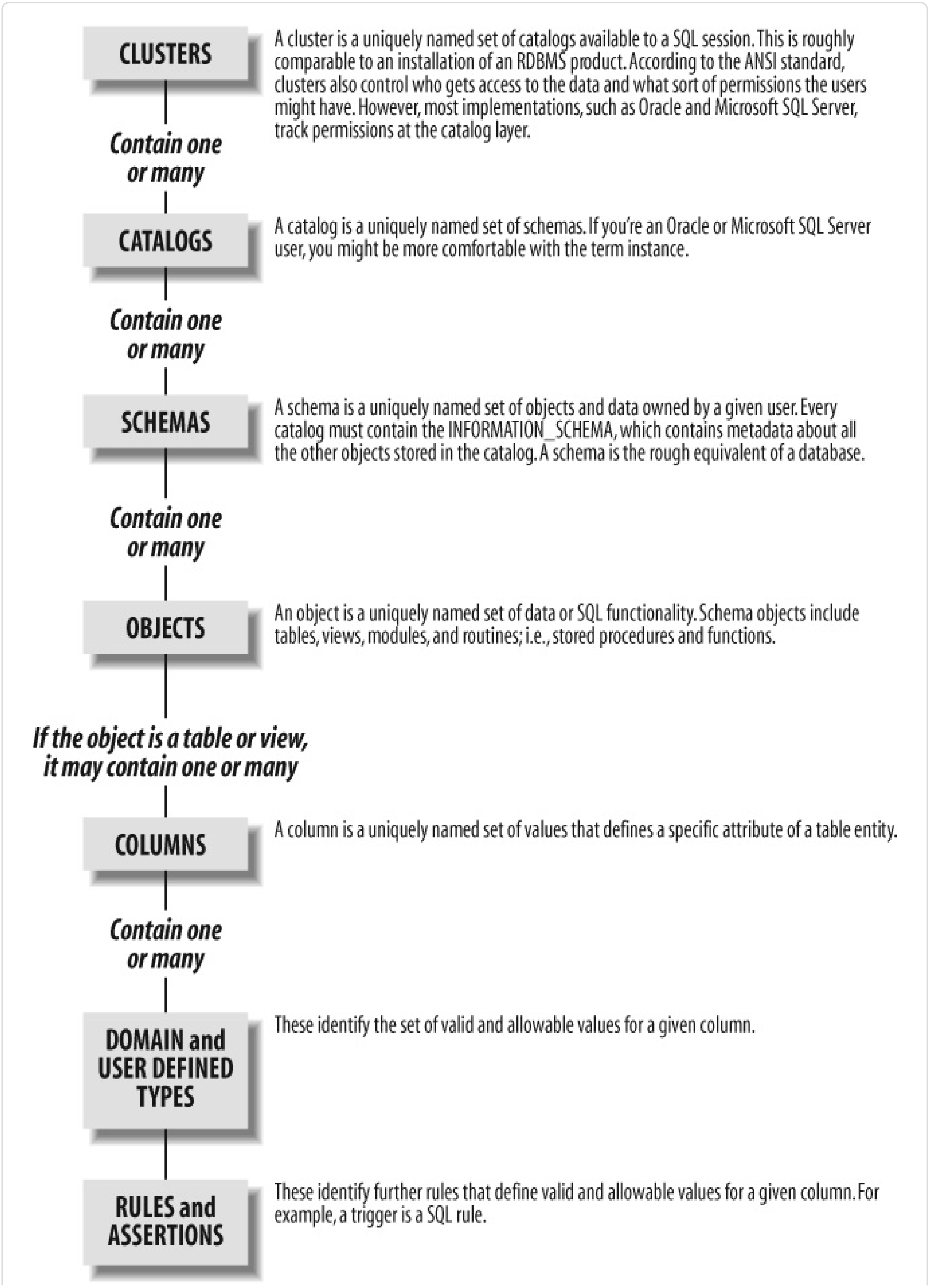
Data structures (rules 1, 2, and 8)

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*.



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.

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.

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.

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.

A column is a uniquely named set of values that defines a specific attribute of a table entity.

These identify the set of valid and allowable values for a given column.

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*
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.
Identifier may contain		
	SQL	Any number or character, and the underscore (_) symbol.
	MySQ	Any number, character, or symbol. Cannot be composed entirely of numbers.

	L	
	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 (_).
	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.
	PostgreSQL	Yes.
	SQL Server	Yes.
Quoted identifier symbol		
	SQL	Double quotes (“ “).
	MySQL	Single quotes (“) or double quotes (“ “) in ANSI compatibility mode.
	Oracle	Double-quotes (“ “).
	Post	Double-quotes (“ “).

	greSQL	
	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.
	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.
	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 Ser	No by default, but can be changed.

	ver	
Other rules		
	SQL	None.
	MySQL	May not contain numbers only.
	Oracle	Database links are limited to 128 bytes and may not be quoted identifiers.
	PostgreSQL	None.
	SQL Server	Microsoft commonly uses brackets rather than double quotes for quoted identifiers.

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

Remember, quoted identifiers (object names encapsulated within a special delimiter, usually double quotes 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
LIKE	*TRUE* if the operand matches a pattern
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
+	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	On all platforms:

>	nonstandard equivalent on several 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	
()	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.

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*.
	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.
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	
	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
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
	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
	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
	None	*MACADDR*			Y
	None	*MEDIUMBLOB*	Y		
	None	*MEDIUMINT*	Y		
	INT	*MEDIUMTEXT*	Y		
	None	*MONEY*			Y
Y	None		Y	Y	Y

		NATIONAL CHARACTER VARYING, NATIONAL CHAR VARYING, NCHAR VARYING, NVARCHAR			
Y	*NATIONAL CHARACTER VARYING*	*NCHAR, NATIONAL CHAR, NATIONAL CHARACTER*	Y	Y	Y
Y	*NATIONAL CHARACTER*	*NCLOB*		Y	
	NCLOB	*NTEXT, NATIONAL TEXT*			
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*			
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*			
		TIMESTAMP	Y	Y	Y

	INTERVAL				
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	
	CHARACTER	*VARCHAR FOR BIT DATA*			

	VARYING				
	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*.

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*
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*, *AURORA$JIS$UTILITY$*, *AURORA$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 Btree 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