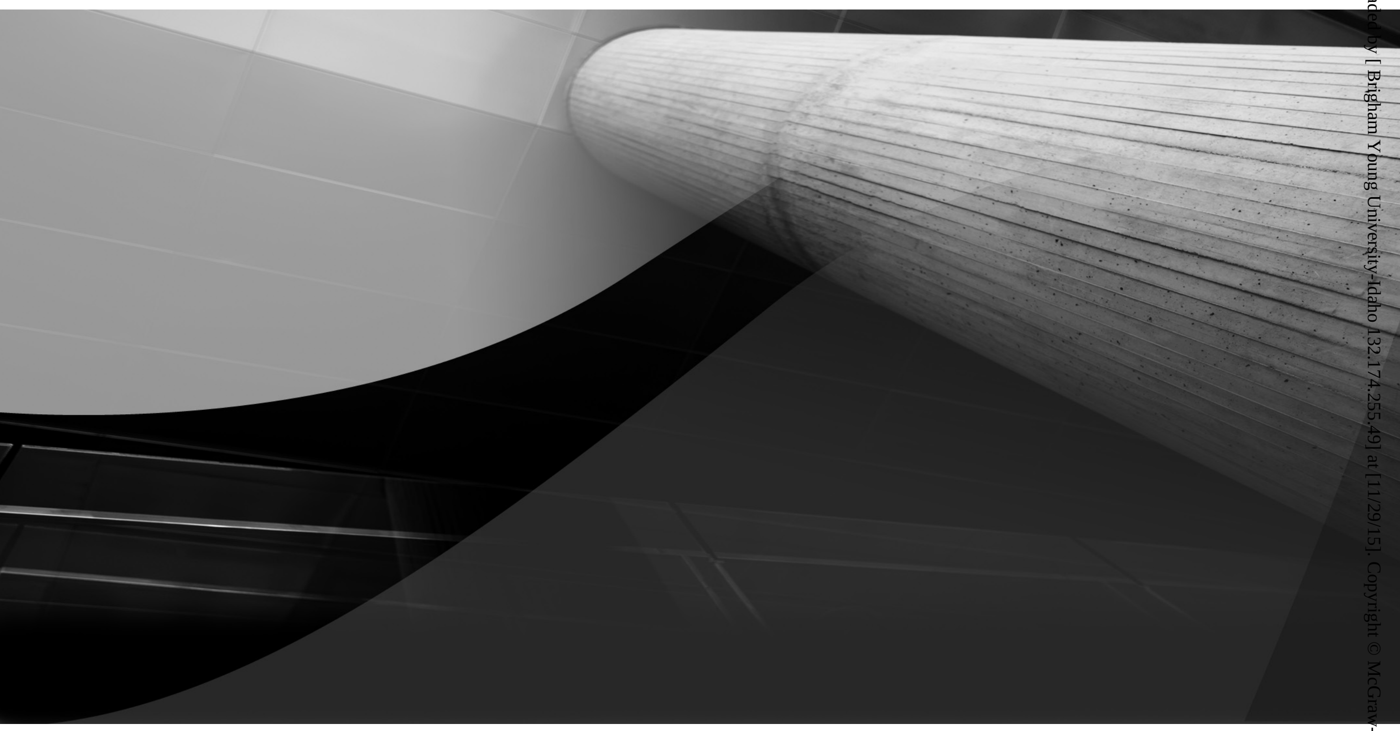


CHAPTER 6

Creating Users and Structures

In this chapter, you'll learn how to create users, databases, tables, indexes, constraints, sequences, and views in Oracle and MySQL databases. The chapter is organized by the following topics:

- Users
- Databases
- Tables
- Indexes

Users are synonymous with *schemas* in an Oracle database, but they are distinct from databases in MySQL databases. A *database* or *schema* is a private work area, but it is also a container of tables. *Tables* are two-dimensional containers of data, and *views* are logical filters to access subsets or supersets of tables. *Sequences* are structures that support automatic numbering of ID columns for tables. Sequences are described and explained in the "Tables" section in this chapter, because although they're independent data structures in an Oracle database, they're properties of tables in the MySQL database. *Constraints* are structures that restrict the type of data you can put in tables. Tables and constraints are also discussed in the "Tables" section.

Indexes are structures that hold search trees that help SQL statements find rows of interest faster. These search trees can be Balanced Trees (B-Trees), hash maps, and other mapping data structures.

Views are covered in Chapter 11, because that's where queries reign supreme; views are nothing more than named queries. Two types of views are used for reference: a *subset* view with a `SELECT` statement that filters data from a table, and a *superset* view with a `SELECT` statement that joins data from two or more tables.

If you wonder why stored programs aren't discussed in this chapter, it is because stored programs and triggers aren't data containers. Stored programs let you implement database-centric transaction management logic and APIs. As such, they are program logic containers. Stored programs are covered in Chapters 13–14 and database triggers are covered in Chapter 15.

The following sections introduce the generic behaviors of users, databases, tables, sequences, and views before qualifying the details of their implementation in Oracle and MySQL. The focus is on the things a developer needs to know.

Users

Users hold privileges to work in the database. Each database designates at least one default *super user*. The super user enjoys *all* privileges in the database. The following sections cover the behaviors of super users in Oracle and MySQL databases.

Oracle Users

The Oracle database defines two super users, `SYS` and `SYSTEM`, and follows the ANSI-SPARC architecture's three-tiered model. This architectural model divides the internal, conceptual, and external view of schemas or databases.

The *internal* view consists of the physical reality of how data is organized, which is specific to any DBMS. The internal view also contains the editable data catalog that maintains all the data about data, or *metadata*. This metadata contains all the definitions of users, databases, tables, indexes, constraints, sequences, data types, and views. Inside the internal view and with the proper credentials, a super user can alter the contents of the data catalog with Data Manipulation Language (DML) statements. That means an authorized user could use an `INSERT`, `UPDATE`, or `DELETE` statement to change critical metadata outside the administrative barrier of system privileges and Data Definition Language (DDL) statements.

NOTE
You should never use DML statements to change the data catalog values without the express instruction of Oracle Support.

The *conceptual* view consists of the community view of data. The *community view* is defined by the users with access privileges to the database, and it represents an administrator's view of data from the perspective of SQL. This view of data provides administrator friendly views of data stored in the data catalog.

It isn't possible to change the contents of the metadata in the community view, except through DDL statements such as `ALTER`, `CREATE`, and `DROP` statements. Developers can use only these DDL statements against objects, or in the case of the `ALTER` statement, against system and database environment settings. These types of environment settings enable such things as database traces measuring behavior and performance. More on these types of DDL statements can be found in the *Oracle Database 11*g *DBA Handbook*.

Oracle implements the concept of a community view as a collection of *striped views*. Striped views detect the user and allow them to see only things they have rights to access. These views typically start with `ALL_`, `DBA_`, and `USER_` prefixes, and you access them as you would any other table or view through queries with a `SELECT` statement. The `ALL_` and `DBA_` prefixed views are accessible only to the Oracle super users: `SYS`, `SYSTEM`, and user-defined accounts granted super privileges.

Every user has access to the community views prefixed with `USER_`. Those views provide access to structures only in the user's schema or personal work area. This approach is more secure than the read-only `information_schema` made available for the same purpose to all users in the MySQL Server.

The *external* view consists of access to the user's schema or database, which is a private work area. Users typically have complete control over the resources of their schema or database, but in some advanced architectures, users can have restricted rights. In those models, the user may be able to perform only the following tasks:

- Create tables and sequences
- Create or replace stored program units
- Provide or rescind grants and synonyms
- Limit access to memory, disk space, or network connections

Oracle's super users (`SYS` and `SYSTEM`) are synonymous with the two schemas for the internal (`SYS`) and conceptual (`SYSTEM`) views.

The differences between the definition of the internal view and the privileges conveyed when connecting as `SYS` aren't immediately visible. You cannot change things in the `SYS` schema when you connect as the `SYS` user, unless you connect with the `/ AS SYSOPER` (*system operator*) or `/ AS SYSDBA` (*system DBA*) privilege. You have full privileges as the system DBA but only a subset of privileges as the system operator. Typically, the only thing you perform with either of these responsibilities is routine maintenance or granting of specialized privileges. Routine maintenance would include starting and stopping the database. Specialized privileges include granting a user wider privileges or revoking privileges already granted, and defining the internal Java permissions though the `DBMS_JAVA` package. You can find more about using the `DBMS_JAVA` package in Chapter 14 of the *Oracle Database 11*g *PL/SQL Programming* book.

Although you can create new users and grant them privileges like the super user, you shouldn't alter the predefined roles of the super users. The next sections describe how you create users and grant privileges to or revoke privileges from a user.

Creating an Oracle User

Creating a user is synonymous with creating a schema in an Oracle database. Because of this, the topic is covered twice in this chapter: here and later in the "Database" section. Here we'll focus on the aspects of authentication, profile, and account status for an Oracle database user.

The SQL prototype to create a user allows you to identify the user with a password, an external SSL-authenticated certificate name, or a globally identified user name based on a Lightweight Directory Access Protocol (LDAP) entry. The certificate is a SSL (Secure Sockets Layer) file. It lets you encrypt your database credentials to support secure data communication.

The following syntax (similar to that in Chapter 2) lets you create a `student` user that is identified by a local database password:

```
CREATE USER student IDENTIFIED BY student;
```

One alternative to a local password is an SSL-authenticated certificate name, which would look like this:

```
CREATE USER student IDENTIFIED EXTERNALLY AS 'certificate_name';
```

The LDAP alternative would look the same but use a different source.

```
CREATE USER student IDENTIFIED EXTERNALLY AS 'CN=miles,O=apple,C=US';
```

Any of the three syntax methods can be used to create a private `student` work area, which is a schema. A number of other options are available for the default and temporary tablespaces of the work area, and quota syntax is available to limit the space authorized for a schema. These clauses are covered in the "Database" section later in the chapter.

Another clause allows you to assign a profile to users when you create them. That clause generally follows any tablespace assignments and quota limits. An example that assumes default assignment of tablespaces and quota limits would look like this with a local password:

```
CREATE USER student IDENTIFIED BY student
PROFILE profile_name;
```

Profiles allow you to restrict the number of concurrent user sessions, amount of CPU per call, and so forth. Profiles also let you impose restrictions or overriding password functions. The latter allows you to enhance the base security provided by the Oracle database, like surrounding the castle gate with a moat.

You can also set a password as expired. With this setting, when the user signs on with a provided password, he or she will be prompted to change it immediately. This is the best practice for issuing user accounts. Accounts are unlocked by default, but sometimes an account should be locked. For example, you might need to create the schema to reference it in another schema before planned use of the schema. These clauses generally follow all of those previously discussed. A sample `CREATE` statement with these clauses would look like this:

```
CREATE USER student IDENTIFIED BY student
PROFILE profile_name
PASSWORD EXPIRE
ACCOUNT LOCK;
```

You can use an `ALTER` statement to unlock the user account when the time comes to activate it. Chapter 7 shows the `ALTER` statement syntax to unlock an account.

Restricting access through the Oracle Transparent Network Substrate (TNS) is accomplished by configuring the Oracle networking stack. This is different from the authentication model in the MySQL database, where the user's point of access is part of his or her unique identification. For example, you can configure the `sqlnet.ora` file to restrict connections within a domain.

The following shows how to enable or exclude client machine access. The parameter lines go into the `sqlnet.ora` file on the server.

```
tcp.validnode_checking = yes
tcp.invited_nodes = (192.168.0.91)
tcp.excluded_nodes = (192.168.0.129)
```

The first parameter allows you to check whether the IP address is authorized or not. The second line shows you how to authorize a client, and the third line shows you how to prohibit a client from connecting to the Oracle database server.

After the user connects to the database, you can provide fine-grain access control through SQL configuration. For example, you can restrict a user's access down to the column level, as with a MySQL database, but you must put the logic into a stored program. Then, you grant execute privileges on the program only to users who should have limited access privileges. Unlike the MySQL Server, Oracle Database 11*g* doesn't support grants of privileges at the column level.

You can find full documentation on Oracle networking in the *Oracle Database Net Services Reference 11*g.

This concludes the basics of setting up a new user account. You can explore more on the topic in the *Oracle Database 11*g *SQL Reference* manual online.

Granting Oracle Privileges

Creating an account in an Oracle database doesn't automatically enable it for use. First you must grant basic permissions to use the account. The `SYSTEM` user or an administrator account created with the `CREATE ANY USER` privilege should run these commands.

As presented in Chapter 2, these are the basic privileges that you would want to extend to a default user. There you saw the syntax to limit access to physical space. Chapter 3 covers system

Oracle Network Tracing

Sometimes you need to trace what's happening in the Oracle portion of the network communication stack. You do that by configuring the `sqlnet.ora` file. It is possible to set four levels of tracing: Oracle Worldwide Support (16), Administration (10), User (4), and Tracing Off (0).

When you add the following parameters to the `sqlnet.ora` file, you generate a server-side network trace file:

```
trace_level_server = 10
trace_file_server = server.trc
trace_directory_server = <path_to_trace_dir>
```

The `trace_level_server` value designates the desired level of tracing. The setting shown here provides values at the local administrator level.

An alternative to server-side tracing is client-side tracing, which can be accomplished by adding a parameter to the `sqlnet.ora` file on the client, like this:

```
trace_level_client = 10
trace_unique_client = on
trace_file_client = sqlnet.trc
trace_directory_client = <path_to_trace_dir>
```

Network tracing is a valuable tool when you're debugging your application stack. You might likewise need to debug the processing instruction sets, and that is included in Chapter 13, where you'll learn how to write stored programs.

and object privileges. However, privileges don't work when you create a user without a default and temporary tablespace clause, unless you also grant `UNLIMITED TABLESPACE` as shown here:

```
GRANT create cluster, create indextype, create operator
,     create procedure, create sequence, create session
,     create table, create trigger, create type
,     create view, unlimited tablespace TO sample;
```

This type of `GRANT` statement lets you create a user in a small, developer-only environment, but you shouldn't do this in a production database. It works because you avoid assigning default and temporary tablespaces by granting unlimited space rights. This is never a good thing to do, except on your laptop! Some might say that you shouldn't do it on your laptop either, but this is something for you to decide.

The `ALTER` statement also lets you assign a default and temporary tablespace after a user is created. Both the `CREATE` and `ALTER` statements let you assign quotas to the default tablespace, but you can no longer assign a quota to the temporary tablespace. Any attempt to do so raises an error. This change became effective with Oracle Database 10*g*. You can find the syntax for the `ALTER` statement in Chapter 7.

Most commercial databases define user profiles and assign them when creating new users. You can find out more about that in the *Oracle Database 11*g *DBA Handbook*.

NOTE
Beginning with Oracle Database 10g Release 2, you can no longer assign a temporary tablespace quota.

A sample grant of select privileges, typically made by a user for his or her own schema objects, would look like this:

```
GRANT SELECT ON some_tablename TO some_user;
```

Sometimes a user wants to grant privileges to another user with the privilege to extend that privilege to a third party. This is the infrequent pattern of grants reserved for setting up administrative users. You append a `WITH GRANT OPTION` clause to give another user the right to provide others with the privileges you've conveyed to them:

```
GRANT SELECT ON some_tablename TO some_user WITH GRANT OPTION;
```

Oracle also supports the concept of a *synonym*, which simplifies how another user can access your object. Without a synonym, the other user would need to put your user name and a dot (`.`) in front of the object before accessing it. The dot is called a *component selector*. A synonym creates an alias that maps the user name, component selector, and object name to a synonym (alias) name in the user's work area or schema.

You don't need to use a component selector on objects that you create in your schema. They're natively available to you. The `SYS` super user has access to every object in the Oracle Database 11*g* Server by simply addressing objects by their fully qualified location—schema name, component selector, and object name. This makes perfect sense when you recall that the user and schema names are synonymous.

You create a synonym like this:

```
CREATE SYNONYM some_tablename FOR some_user.some_tablename;
```

Typically, the local table name is the same as the table name in the other schema, but not always. You can also grant privileges on a table to a `PUBLIC` account, which gives all other users access to the table. Public synonyms also exist to simplify how those users access the table.

You would grant the `SELECT` privilege to the `PUBLIC` account with this syntax:

```
GRANT SELECT ON some_tablename TO PUBLIC;
```

After granting the privilege, you create a public synonym with this syntax:

```
CREATE PUBLIC SYNONYM some_tablename FOR some_user.some_tablename;
```

As a rule of thumb, use the `PUBLIC` account only when you're granting privileges to invoker rights stored programs. Chapter 3 discusses the default definer and invoker rights models. Chapter 13 shows you how to define stored programs that run under definer or invoker rights models.

Revoking Privileges

You can revoke any privilege from a user provided you or a peer super user made the grant. Let's say you just finished reading Chapter 7 on the `ALTER` statement and realized that you should remove the `UNLIMITED TABLESPACE` privilege from the `student` user. That command would look like this:

```
REVOKE unlimited tablespace FROM student;
```

The funny thing about this revocation is that it doesn't immediately disable a user from writing to the tablespace generally. That's because revocation only disallows the allocation of another extent to any table previously created by the user. An extent is a contiguous block of space inside a tablespace. Extents are added when an `INSERT` or `UPDATE` statement can't add anything more in the allocated space. The number of extents allocated to a table is a measure of the fragmentation of the table on disk. Chapter 7 examines how you can defragment storage.

You can revoke privileges from the `PUBLIC` account with the same type of syntax:

```
REVOKE SELECT ON some_tablename FROM PUBLIC;
```

When you revoke privileges that included a `WITH GRANT OPTION` clause, make sure you also revoke the granting option. There should be a routine process in place for validating the grants and privileges to ensure that they comply with your company's governance policy and appropriate laws, such as Sarbane-Oxley in the United States. This is referred to as hardening. You can find more about hardening in an application context in the book *Oracle E-Business Suite Security*.

MySQL Users

The MySQL database defines one `root` super user. MySQL, like Oracle, supports the ANSI-SPARC architecture's three-tiered model. As mentioned, that model divides responsibilities into the internal, conceptual, and external view of schemas or databases.

MySQL provides access to the internal view to the `root` super user. The internal view or data catalog is stored in the `mysql` database. It is possible to edit the catalog as the `root` super user with DML statements, such as `INSERT`, `UPDATE`, and `DELETE`. Although it's not a best practice, it is done by DBAs from time to time.

The conceptual view was added in MySQL 5 as the `information_schema` database. It is a read-only copy of the `mysql` database. At the time of writing, all users have read-only access to this database. It actually provides more access to the database catalog than minimally authorized users should have, because they can explore definitions of other databases. This access to the `information_schema` compromises the principle of information hiding.

Like an Oracle database user schema, MySQL databases are the external views. Users that hold permissions on user-defined databases can run DDL statements against the database's tables, constraints, indexes, and views; and they can run DML statements against tables and views.

Creating a MySQL User

MySQL users are defined by three key attributes: the user name, password, and access point of origin. The user name is stored as a plain text string. The password is stored as an encrypted string. The access point origin, known as `host`, designates the permitted origin of communication with the MySQL database. The options for hostname are `localhost`, `hostname`, IP address, domain or subdomain address, or a wildcard, which is the percent (`%`) character.

The most generic way to create a user in MySQL excludes reference to the access point of origin. You would use the following syntax to implicitly create a user that can connect from anywhere:

```
CREATE USER 'some_username' IDENTIFIED BY 'some_password';
```

Alternatively, you could create the same user by qualifying anywhere, like this:

```
CREATE USER 'some_username'@'%' IDENTIFIED BY 'some_password';
```

When you want to link an access point through a DNS server lookup, you would qualify the user with the following syntax. This is the easiest configuration when you know the hostname, and IP addresses are provided through DHCP licenses that can change over time.

```
CREATE USER 'some_username'@'hostname.company.com'
IDENTIFIED BY 'some_password';
```

You can exclude the hostname and substitute an IP address to accomplish the same task. That's possible when user machines are assigned static IP addresses.

```
CREATE USER 'some_username'@'192.168.1.124'
IDENTIFIED BY 'some_password';
```

Limiting connections to machines within your company's domain is a common configuration for limiting developer connections. You would substitute a `%` as the wildcard for hostname before the domain name as the access point of origin to accomplish this, like so:

```
CREATE USER 'some_username'@'%.company.com'
IDENTIFIED BY 'some_password';
```

Another option lets you limit access points of origin to a subdomain within a company. Two options are available for that. The first uses the `%` wildcard operation and the other the IP number and netmask. Here's the wildcard subnet syntax:

```
CREATE USER 'some_username'@'192.168.%'
IDENTIFIED BY 'some_password';
```

Here's the more complex IP number and netmask:

```
CREATE USER 'some_username'@'10.0.0.0/255.255.255.0'
IDENTIFIED BY 'some_password';
```

This matches the first 24 bits of its IP number, which maps to 10.0.0. It lets a specified user connect from any host in the subnet 10.0.0!

The most restrictive connection is `localhost`, which means the user can connect only from the server where the MySQL database is installed. A user with only `localhost` as an access point of origin can't connect through a web server. Here's the syntax for a server only connection:

```
CREATE USER 'some_username'@'localhost'
IDENTIFIED BY 'some_password';
```

It's important to know that the `IDENTIFIED BY` clause may be *excluded when you create a user*. That means the user can initially connect to the database without providing a password.

Creating users without a password is not a good practice, but you can do it, so you should know how to add a password.

The `root` super user can add a password once the user is connected with the following syntax:

```
SET PASSWORD FOR 'some_username'@'%' = PASSWORD('some_password');
```

It is also possible for a connected user to set his or her own password. The syntax is virtually the same. Only the `FOR 'some_username'@'%'` clause is excluded, like this:

```
SET PASSWORD = PASSWORD('some_password');
```

Other, more advanced ways of creating users can be used as well. Specifically, it's possible in MySQL to create a user with the same name but different hosts and passwords. When you choose to do this, it means all subsequent grants of privileges must include the user, password (through the `IDENTIFIED BY` clause), and host.

At this point, a user is capable of connecting to the MySQL database but he or she can't do anything meaningful. A MySQL user enjoys more access than an Oracle user connecting with the /NOLOG option as demonstrated in Chapter 2. This appears as a security weakness that Oracle will fix eventually, but at the time of writing, a user without any permission can see the `information_schema` and connect to it. This means that once a nonprivileged user is connected, he can explore the information about databases that he can't see with the `SHOW` command.

Two-Face User

Sometimes you want to create a user who has different levels of access when they're working through different types of connections to the database. You can do this in the MySQL database, because users are uniquely resolved by a combination of their user name, password, and host connection designator.

Let's say you want a developer to have full control when she is working on the database server and only limited control when she is connecting through the network from a client. You define two instances of the user account. The first lets the `two_face` user connect from the server that's hosting the MySQL database, like this:

```
CREATE USER 'two_face'@'localhost'
IDENTIFIED BY 'some_password';
```

The second `two_face` user connects from somewhere in the company domain, like this:

```
CREATE USER 'two_face'@'hostname.company.com'
IDENTIFIED BY 'some_password';
```

Now you can grant different privileges to the `two_face` user based on where she is when she connects to the database server. The only catch is that all `GRANT` statements must include the user's unencrypted password. Naturally, this means you'll need to create a trivial password while managing the grants. You can reset the password later with the `SET PASSWORD` command, and the grants migrate to the correct user.

Granting MySQL Privileges

Granting privileges in MySQL differs slightly from how it's done in Oracle. Specifically, users are created separately from databases. This means users don't have private work areas when you create them. You have to create a user and a database, and then grant privileges on the database to the user. The "Databases" section of this chapter shows you how to create databases, but for this discussion you don't need to know that syntax.

Chapter 3 lists the privileges that you can grant. You have the option of granting all privileges or individual privileges. If you want to provide a user the equivalent privileges that an Oracle user enjoys, you grant all privileges on a database.

The syntax to grant all privileges on all objects uses the name of a database, a period, and a * (wildcard operator that represents any table or subroutine). Here's an example of the syntax:

```
GRANT ALL ON some_database.* TO some_username;
```

You can also grant limited privileges to a table, column of a table, or stored programs (also called a routine). The following syntax is used to grant privileges to query, update, and delete anything from a table:

```
GRANT ALL ON some_database.some_table TO some_username;
```

The alternative of granting only `SELECT` privileges would result in a `GRANT` statement like this:

```
GRANT SELECT ON some_database.some_table TO some_username;
```

Grants to a user who has two or more access points of origin requires that you append the `IDENTIFIED BY` clause with a plain text password. Here's the sample syntax:

```
GRANT SELECT ON some_database.some_table
TO some_username IDENTIFIED BY some_password;
```

Sometimes you want to grant restricted privileges on some columns to protect data in other columns of a table. MySQL supports this capability by letting you grant privileges at the column level. You do this by providing a comma-delimited list of authorized columns in parentheses. The following syntax prevents an update to a surrogate key column (typically an automatically numbered ID column), while providing rights to update another column, and the right to select (query), insert, or delete rows from the same table:

```
GRANT SELECT
,     INSERT
,     UPDATE (allowed_column_list)
,     DELETE ON some_database.some_table TO some_username;
```

This grants `SELECT`, `INSERT`, and `DELETE` privileges on the entire table, which is more often the case. The `UPDATE` privileges are restricted to a designated column.

You don't have the ability to perform this type of restricted grant in the Oracle Database 11*g*. The only way to deploy equivalent functionality would be in a row-level database trigger. Chapter 15 covers database triggers.

The next examples show you how to grant privileges on stored functions and procedures. Stored functions and procedures are the two types of stored programs supported by the MySQL database. This syntax uses the `EXECUTE` option to grant privileges to run a stored function, which is what you should always do. Some examples on the Internet use the ALL option, but that grants a user the ability to run and perform DDL commands on any schema. You don't want to extend the ability to drop a function or procedure beyond the user who is responsible for the code module. More or less, the rule of thumb is never to grant anything that's not absolutely required.

```
GRANT EXECUTE ON FUNCTION some_database.some_function TO some_username;
```

Alternatively, you simply replace the function name with a procedure name to give permissions to run a procedure:

```
GRANT EXECUTE ON PROCEDURE some_database.some_procedure TO some_username;
```

You can also grant privileges with the right to grant them to others. That's done by appending a WITH GRANT OPTION clause. The next example gives privileges to select data from a table and the ability to let that user grant the same privilege to select data to others:

```
GRANT SELECT ON some_database.some_table TO some_username
WITH GRANT OPTION;
```

As you can see, there's quite a combination of alternatives when you grant privileges in the MySQL database. The rule with these, as qualified in Chapter 3, is simple: grant privileges only as situations and plans require.

Revoking MySQL Privileges

Revoking privileges works the same way in MySQL as it does in Oracle. You can revoke a selection privilege from a user with the following syntax:

```
REVOKE SELECT ON some_database.some_table FROM some_username;
```

Conveniently, the MySQL database lets you revoke all privileges and the grant option in a single command. The neat thing about the command is you don't have to validate the privileges held by the user first, because it removes all privileges. Here's the syntax for the command:

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM some_username;
```

This works in all cases where a user has only one access point of origin. You need to qualify the access point of origin when a user exists in the database with two or more different hostname values. This would be the syntax for a company domain hostname:

```
REVOKE ALL PRIVILEGES, GRANT OPTION
FROM 'some_username'@'%.company.com';
```

The user name and hostname must be enclosed in single or double quotes and joined by the @ symbol. The database uses the two arguments to resolve the ambiguity of the user existing two or more times in the data catalog.

Databases

The implementation of Oracle and MySQL databases differ in some respects, but they serve the same purpose. Databases are synonymous with schemas (or scheme in the Latin plural) and they act as work areas. Most Oracle documentation refers to them as schemas, while most MySQL documentation refers to them as databases. MySQL added *schema* as an alias for the *database* keyword in MySQL 5.0.2.

The significant difference between an Oracle schema and MySQL database is in who owns them. A schema is synonymous with a user in an Oracle database, because it's a private work area for the individual user. This means an Oracle schema is owned by its user. Dependent on the permissions of that Oracle user, he or she can grant privileges to others to use objects in their schema. A database in MySQL isn't owned by anyone in particular but is a private work area. Permissions to work in a MySQL database are granted by the prerogative of the `root` super user. The `root` user may grant limited or unlimited access on a database to one or more users. Grants of permissions can be as narrow as rights to query a single table or column in a table.

The next two segments show you how to define and use databases in Oracle and MySQL. These sections demonstrate the syntax that supports how you grant object privileges. They extend concepts discussed in Chapter 3 on database security.

Oracle Schemas

A brief sidebar in Chapter 2, "Create a Default Oracle User," discussed how you can create a basic database schema. This segment expands on that small example, which allowed you to create a `student` Oracle user/schema with a minimum set of permissions. A user in an Oracle database typically holds permissions to work in a single database schema only. Broader permissions can be granted with syntax covered in Chapter 3, but such grants are discouraged, as mentioned there.

The following syntax (from Chapter 2) lets you create a `student` account:

```
CREATE USER student IDENTIFIED BY student
DEFAULT TABLESPACE users QUOTA 50M ON users
TEMPORARY TABLESPACE temp;
```

It creates a limited account, because a quota limits physical size of the database container. You can add up to 50 megabytes of material in the `student` database, which is physically stored in the `users` tablespace. The `users` tablespace is available in all sample Oracle databases. Note that a *tablespace* is a logical unit that can contain one or more users. One or more physical data files are assigned to a tablespace to make it useful. You would define a `my_tablespace` tablespace of your own like this:

```
CREATE TABLESPACE my_tablespace
DATAFILE 'C:\Oracle\Data\my_tablespace01.dbf' SIZE 5242880
AUTOEXTEND ON NEXT 1310720 MAXSIZE 32767M
LOGGING ONLINE PERMANENT BLOCKSIZE 8192
EXTENT MANAGEMENT LOCAL AUTOALLOCATE DEFAULT NOCOMPRESS
SEGMENT SPACE MANAGEMENT AUTO;
```

This is the syntax Oracle uses to create the sample database's `users` tablespace. This designates the physical data file, assigns it an initial size of 5 MB, the amount to increase file size (an extent), and a maximum size. Then, it enables logging changes, places the tablespace online with a designated blocksize and delegated extent management of an uncompressed file. More on this can be found in the Oracle DBA Handbook (McGraw-Hill). After creating a tablespace, you can add more space to the tablespace by adding physical files. Chapter 7 contains the syntax for adding files to a tablespace.

You can discover the syntax for existing tablespaces by using the following query as the SYSTEM user:

```
SET LONG 300000
SELECT dbms_metadata.get_ddl('TABLESPACE','USERS') from dual;
```

The SET LONG command expands the display size of output from the built-in function. The first call parameter designates the object type you want to view, and the second call parameter provides the object name. The function returns the DDL command that created the target object.

MySQL Databases

Another sidebar in Chapter 2 discussed how to set up a MySQL database. Like Oracle, the super user holds the privilege for creating databases. This is the root super user in MySQL, and this user can grant the system privilege to create databases to other administrative users. The super user can also grant object privileges to users to work in databases.

MySQL users can be granted permissions directly from a super user to work concurrently in multiple databases. MySQL databases are like schemas in an Oracle database, but they are unassigned private work areas or containers. They hold tables, views, and stored programs.

A super user creates a database with either of the following syntaxes:

```
CREATE DATABASE studentdb;
```

or

```
CREATE SCHEMA studentdb;
```

By default, MySQL 5.5 and later versions use the InnoDB engine and use the InnoDB tablespaces defined in the my.ini (Windows) or my.cnf (Linux) file. A default community edition configuration of those files would look like this:

```
# The home directory for InnoDB
innodb_data_home_dir="C:/MySQL55 InnoDB Datafiles/"
# Memory pool that is used by InnoDB to store metadata information.
innodb_additional_mem_pool_size=3M
# If set to 1, InnoDB will flush (fsync) the transaction logs to the
# disk at each commit, which offers full ACID behavior.
innodb_flush_log_at_trx_commit=1
# The size of the buffer InnoDB uses for buffering log data.
innodb_log_buffer_size=2M
# InnoDB, unlike MyISAM, uses a buffer pool to cache indexes and row data.
innodb_buffer_pool_size=107M
# Size of each log file in a log group.
innodb_log_file_size=54M
# Number of threads allowed inside the InnoDB kernel.
innodb_thread_concurrency=8
```

After creating the database, the super user would create a student user and grant the user access to a studentdb database with syntax that does the following: specifies from where they can connect to the database, such as `localhost`, an IP address, or domain, or provides connection from any location. The syntax possibilities are shown in the following examples.

- Connecting only from the `localhost` (or same machine as the database server):

  ```
  CREATE USER 'student'@'localhost' IDENTIFIED BY 'student';
  GRANT ALL ON studentdb.* TO 'student'@'localhost';
  ```

- Connecting only from a designated IP address:

  ```
  CREATE USER 'student'@'172.16.123.129' IDENTIFIED BY 'student';
  GRANT ALL ON studentdb.* TO 'student'@'172.16.123.129';
  ```

- Connecting only from a domain:

  ```
  CREATE USER 'student'@'*.mydomain.com' IDENTIFIED BY 'student';
  GRANT ALL ON studentdb.* TO 'student'@'*.mydomain.com';
  ```

- Connecting from anywhere:

  ```
  CREATE USER 'student'@'%' IDENTIFIED BY 'student';
  GRANT ALL ON studentdb.* TO 'student'@'%';
  ```

Note that the user must be defined for the access route designated in the grant of permissions. If the access route and permissions don't agree, the grant won't work. If you're not sure which databases exist, you can run the following show command:

```
show databases;
```

This lists the databases that an individual may access. The `root` user sees all available database in the MySQL server.

Tables

Database tables are two-dimensional record structures that hold data. Grants of permissions to read and write data are most often made to tables. Sometimes grants restrict access to columns in tables.

Although databases contain tables, tables contain data organized by data types. A data type is the smallest container in this model. It defines what type of values can go into its container. Data values such as numbers, strings, or dates belong respectively in columns defined as numeric, variable length string, and date data types. Data types that hold a single value are scalar data types (or, to borrow some lingo from Java, primitive data types). Tables are seldom defined by a single column. They are typically defined by a set of columns. The set of columns that defines a table is a type of data structure. It is more complex than a single data type because it contains a set of ordered data types. The position of the elements and their data types define the structure of a table. The definition of this type of structure is formally a *record structure*, and the elements are fields of the data structure.

This record structure description can be considered the first dimension of a two-dimensional table. The rows in the table are the second dimension. Rows are organized as an unordered list, because relational operations should perform against all rows regardless of their positional order.

Tables are defined by the DDL `CREATE TABLE` command. The command provides names for columns, data types, default values, and constraints. The column, data type, and default values must always be defined on the same line but constraints can be defined two places. Defining a constraint on the same line as a table column is defining an in-line constraint. This is the typical pattern for *column constraints*, such as a `NOT NULL` or a single column `PRIMARY KEY`. You can opt to define column constraints after all columns are defined. When you do so, the constraints are out-of-line constraints. Sometimes constraints involve more than one column. Constraints that apply to two or more columns are *table constraints*.

A `NOT NULL` constraint is always a column constraint. The ANSI SQL standard requires that all columns in tables be unconstrained by default. An unconstrained or *nullable column* is an optional column when you insert or update a row. A *not null column* is a mandatory column when you insert or update a row. Both the Oracle and MySQL databases adhere to this ANSI standard use. Note that Microsoft SQL Server doesn't adhere to the standard, because it makes all columns mandatory by default.

RDBMS implementations comprise five basic groups of data types: numbers, characters, date-time intervals, large objects, and Boolean data types. The Boolean data type was added in ANSI SQL:1999, and it includes three-valued logic: true, not true, and null. It adopts three-valued logic because the ANSI SQL-89 and later standards accept that any column can be a null allowed—or, simply put, a column can contain no value or be empty.

Although RDBMSs determine which data types they'll support, they also determine how they'll implement them. Some data types are scalar or primitive data types, and others are built on those primitive data types. Only Oracle (and PostgreSQL) supports building composite data types that can be implemented as nested tables.

The Oracle Database 11*g* does not support a Boolean data type. The MySQL database supports an alias Boolean, which maps to a `TINYINT` data type. The `TINYINT` data type holds a numeric value of either `0` or `1`.

The lack of a Boolean data type does sometimes cause problems, because standard comparison operators don't work with null values. Null values require the `IS` or `IS NOT` comparison operator, which is a reference operator rather than a value comparison operator. Table design and management should take into consideration the processing requirements to handle three-valued logic of pseudo-Boolean data types effectively.

Oracle and MySQL also support cloning tables. Rather than repeat an example in both the Oracle and MySQL sections of this chapter, the syntax is better placed here:

```
CREATE TABLE target_table_name AS SELECT * FROM source_table_name;
```

This syntax replicates the structure of an existing table in a new table. It also clones, or copies, all the data from the source table to the target table. When you incorporate a storage

Three-Valued Logic

Three-valued logic means basically that if you find something is true when you look for truth, it is true. By the same token, when you check whether something is false and it is, then it is false. The opposite case isn't proved. That means when something isn't true, you can't assume it is false, and vice versa.

The third case is that if something isn't true, it can be false or null. Likewise, if something isn't false, it can be true or null. Something is null when a Boolean variable is defined but not declared, or when an expression compares something against another variable that is null.

clause in an Oracle database, this process allows you to disable constraints, move the table contents, drop the table, and re-create it with contiguous space. Naturally, you should drop the extra copy after re-creating the table.

The following sections address how you create tables in Oracle and MySQL databases. They individually highlight and compare similarities, differences, and portability.

Oracle Tables

Oracle supports scalar data types, object types, and collections of scalar data types and object types. The collections are also known as object tables. Oracle databases support two uses of object tables. One lets you return a result set through a stored function, which is extremely useful and demonstrated in Chapter 13. The other lets you define a table with columns that use an object table as their data type—basically a table within a table. A table within a table is called a nested table.

This section covers the generalized syntax and how to define the following:

- Sequences
- Scalar data type columns
- Nested table data types
- Column and table constraints
- Externally organized tables
- Partitioned tables

You'll create small, single-column tables in the "Scalar Data Type Columns" sections and explore some database constraints. The nested table data types section shows you how to define a SQL record structure and then deploy it in a table. The column and table constraints section reviews in detail the available approaches to database constraints, some of which are covered in earlier data type sections. The partitioned tables section demonstrates approaches to partitioning the data storage.

Oracle Database 11*g* provides two options when creating tables: You can create ordinary tables that persist from session to session or create temporary tables that exist only during the duration of the session. As a rule, temporary tables are not liked by DBAs because they inherently fragment disks. You should make sure that you work with your DBA when you opt for temporary tables, because the DBA might have created a special tablespace for temporary tables to minimize impacts on other tables.

CAUTION
It's a bad idea to create temporary tables without consulting the DBA about them. This will ensure that you don't inadvertently fragment the production database.

You would create a table like this, where the ellipses represents columns and constraints:

```
CREATE TABLE table_name (...);
```

The general and basic prototype for a relational table with the `CREATE TABLE` statement without storage clause options is:

```
CREATE [GLOBAL] [TEMPORARY] TABLE [schema_name.] table_name
( column_name data_type [{DEFAULT expression | AS (virtual_expression)}]
 [[CONSTRAINT] constraint_name constraint_type]
,[column_name data_type [{DEFAULT expression | AS (virtual_expression)}]
 [[CONSTRAINT] constraint_name constraint_type]
,[...]
,[CONSTRAINT constraint_name constraint_type(column_list)
  [REFERENCES table_name(column_list)]]
,[...]);
```

You create a temporary table by inserting two keywords and one of two clauses. The clauses can be `ON COMMIT PRESERVE ROWS` or `ON COMMIT DELETE ROWS`. The former lets the rows remain during the session, whereas the latter makes the table transactional. You would create a session-based temporary table, like so:

```
CREATE GLOBAL TEMPORARY TABLE table_name
ON COMMIT PRESERVE ROWS
AS
SELECT i.item_title FROM item WHERE i.item_title LIKE 'Star%';
```

Figure 6-1 shows you how the `CREATE TABLE` statement defines a permanent table with different types of column and table constraints. The figure is annotated to help you see available possibilities when you create tables.

In-line constraints are always single column constraints, and they apply to the column defined on the same line. Out-of-line constraints are defined after the last column in a table. When an out-of-line constraint applies to a single column, it is a column constraint. A table constraint is an out-of-line constraint that applies to two or more columns defined in the table. Oracle's `CHECK` constraint and generic `PRIMARY KEY`, `UNIQUE`, and `FOREIGN KEY` constraints can apply to more than a single column, and that makes them possible table constraints.

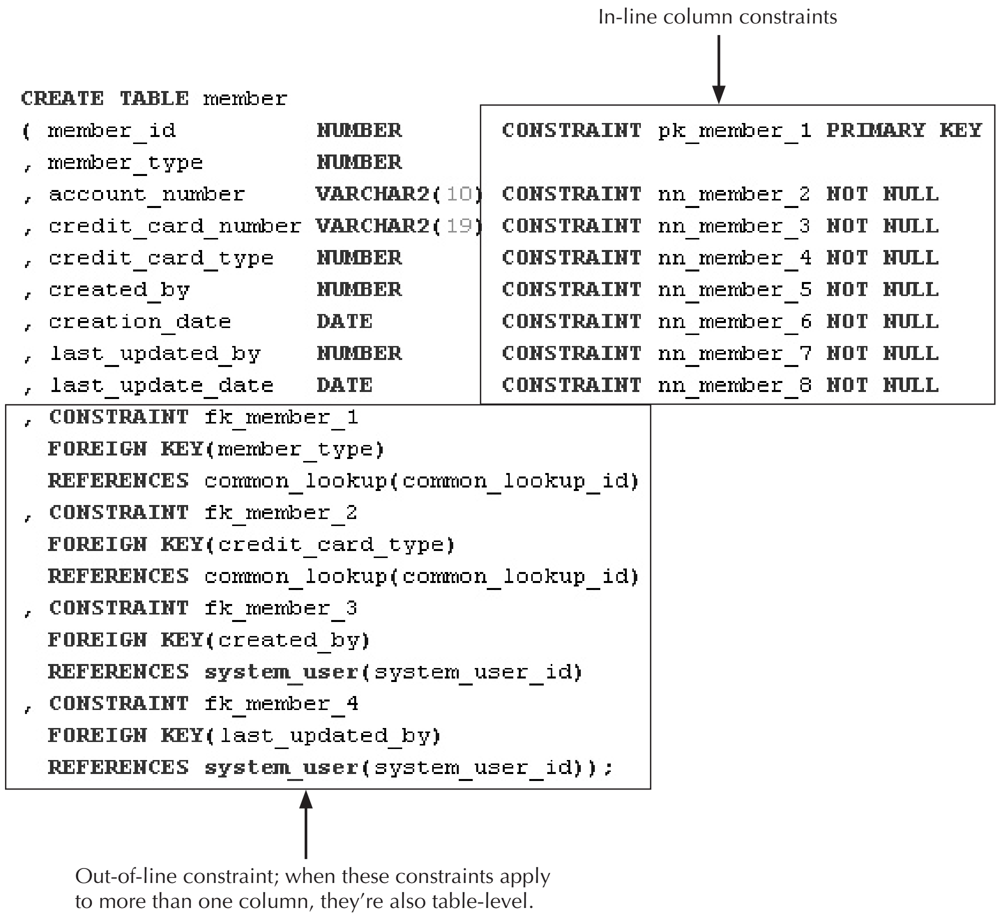
Sequences

Oracle Database 11*g* doesn't support automatic numbering in tables. It provides a separate `SEQUENCE` data structure for use in surrogate keys. Surrogate keys are artificial numbering sequence values that uniquely define rows. They're typically used in joins, because subsequent redefinition of a natural key doesn't invalidate their ability to support joins across tables. The "Indexes" section later in this chapter qualifies how to use surrogate key columns with the natural key to define row uniqueness and optimize joins.

A typical sequence holds a starting number, an incrementing unit, and a buffer cycling value. Each time you call the sequence with a `sequence_name.NEXTVAL` statement, the value of the sequence increases by one (or whatever else was chosen as the `INCREMENT BY` value). This occurs until the system consumes the last sequence value in the buffer cycle. When the last value has been read from the buffer cache, a new cycle of values is provided to the instance. The default for the cycle or sequence buffer is a set of 20 number values.

You create a `SEQUENCE` structure with the default values, like this:

```
CREATE SEQUENCE sequence_name;
```


In-line column constraints
Out-of-line constraint; when these constraints apply to more than one column, they're also table-level.

```
CREATE TABLE member
( member_id          NUMBER         CONSTRAINT pk_member_1 PRIMARY KEY
, member_type        NUMBER
, account_number     VARCHAR2(10)   CONSTRAINT nn_member_2 NOT NULL
, credit_card_number VARCHAR2(19)   CONSTRAINT nn_member_3 NOT NULL
, credit_card_type   NUMBER         CONSTRAINT nn_member_4 NOT NULL
, created_by         NUMBER         CONSTRAINT nn_member_5 NOT NULL
, creation_date      DATE           CONSTRAINT nn_member_6 NOT NULL
, last_updated_by    NUMBER         CONSTRAINT nn_member_7 NOT NULL
, last_update_date   DATE           CONSTRAINT nn_member_8 NOT NULL
, CONSTRAINT fk_member_1
  FOREIGN KEY(member_type)
  REFERENCES common_lookup(common_lookup_id)
, CONSTRAINT fk_member_2
  FOREIGN KEY(credit_card_type)
  REFERENCES common_lookup(common_lookup_id)
, CONSTRAINT fk_member_3
  FOREIGN KEY(created_by)
  REFERENCES system_user(system_user_id)
, CONSTRAINT fk_member_4
  FOREIGN KEY(last_updated_by)
  REFERENCES system_user(system_user_id));
```

FIGURE 6-1. *Oracle `CREATE TABLE` statement*

Sometimes application development requires preseeding (inserting before releasing an application to your customer base) rows in tables. Such inserts are done manually without the sequence value or with a sequence starting at the default `START WITH` value of `1`. After preseeding the data, you drop the sequence to modify the `START WITH` value because Oracle doesn't provide an alternative to modifying it.

Preseeding generally inserts 10 to 100 rows of data, but after preseeding data, the `START WITH` value is often set at 1001. This leaves developers an additional 900 rows for additional post-implementation seeding of data. You create a sequence starting at that value like this:

```
CREATE SEQUENCE sequence_name START WITH 1001;
```

Oracle requires that you couple sequences with database triggers to mimic automatic numbering. Chapter 8 shows you how to call the sequence to insert values, and Chapter 15 shows you how to write the necessary trigger that supports automatic numbering.

Scalar Data Type Columns

Oracle supports only four of the five groups in SQL: numbers, characters, date-time-intervals, and large objects data types. While a Boolean is available in Oracle's Procedural Language extension, PL/SQL, it isn't provided for as a data type in SQL. Your only alternative to a Boolean data type would be to implement a number data type that mimics a Boolean, as you'll see in the "Boolean" section a bit later.

You also have support for virtual columns, which are created by concatenating or calculating values from other column values in the same row. This became available in Oracle Database11*g*. Virtual columns are typically scalar values, and they're discussed in the last subsection of this section.

Figure 6-2 shows you the data types for SQL. Each standalone box contains a group, and within each group are other boxes that contain subgroups.

NOTE
Remember that a Boolean doesn't exist as a default type.

Oracle also supports ANSI-compliant data types that automatically map to native Oracle data types. Writing scripts in the ANSI standard data types makes your scripts more portable to MySQL databases. Table 6-1 shows you the data type mapping when you use the ANSI standard aliases. Some types don't exist in the Oracle ANSI set, such as `TEXT` for a character large object. Oracle uses `CLOB` for that data type.

Number Data Type Numbers have four subgroups: three use proprietary Oracle math libraries—binary integers, `PLS_INTEGER`, and `NUMBER` data types. The new IEEE 754 variable data types use the operating system math libraries and are recommended when you want to do more than financial mathematics. For example, a cube root of 27 has mixed results in PL/SQL with the `**` (double asterisks) exponential operator and a `NUMBER` data type, but works perfectly with a `BINARY_DOUBLE` data type.

You can put whole numbers or decimal numbers in any of the numeric data types except the integer types—they only take integers. Number data types allow you to qualify precision and scale. *Precision* is the number of allowed digits that fall before and after a decimal point. *Scale* is the number of allowed digits that follow the decimal point. This same concept applies to the `DEC`, `DECIMAL`, `NUMERIC`, `BINARY_DOUBLE`, and `DOUBLE_PRECISION` data types. For example, the following sets 12 as the maximum number of digits with 2 digits on the right of the decimal point. You define the precision and scale for `DECIMAL` numbers inside parentheses and separated by a comma.

```
SQL> CREATE TABLE sample_number
  2  (column_name  NUMBER(12,2));
```

You can inspect the table by describing it in a SQL*Plus session, or displaying it in SQL*Developer. Here's the SQL*Plus command:

```
DESCRIBE sample_number
```

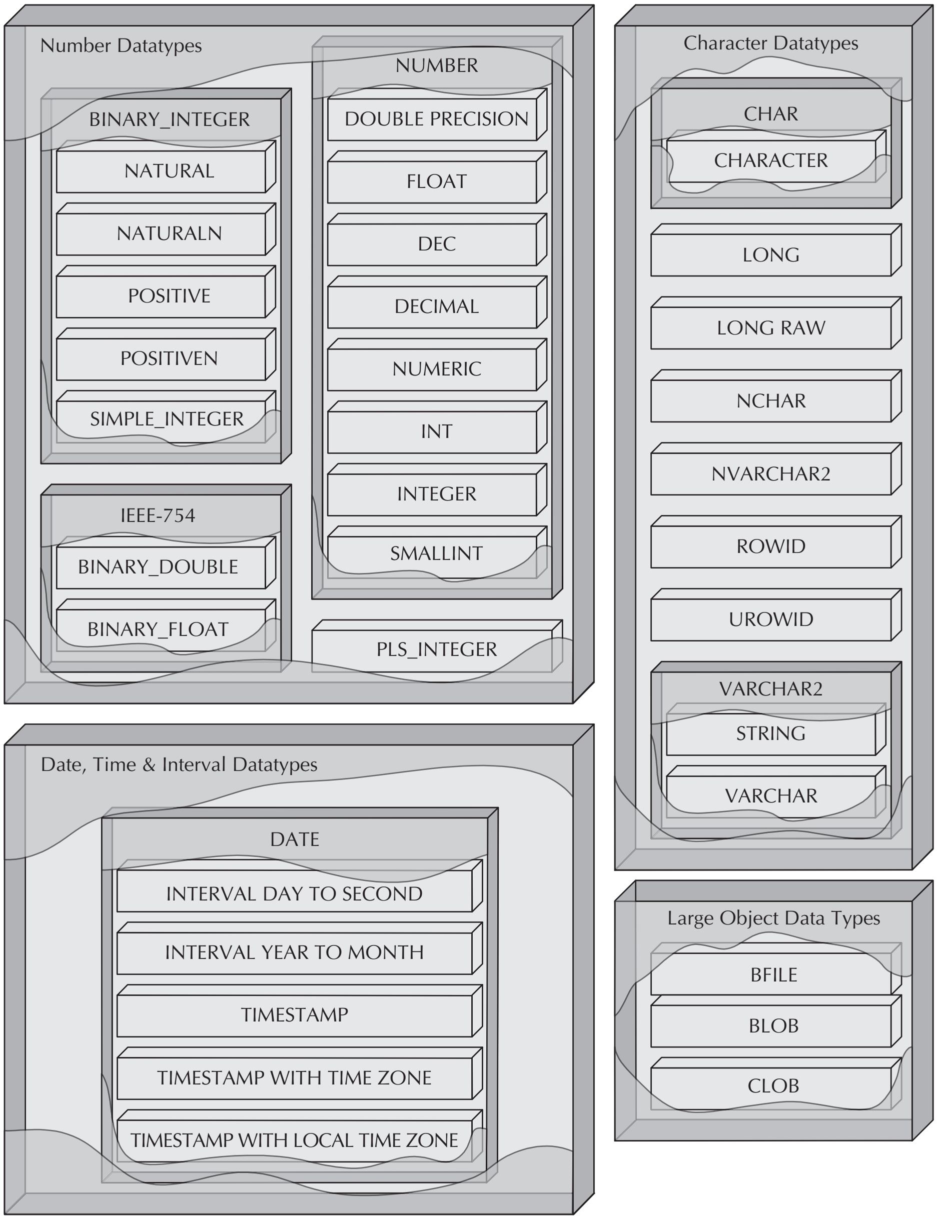


FIGURE 6-2. *SQL scalar types*

ANSI Data Type	Native Data Type	Physical Size
`BLOB`	`BLOB`	8 to 32 terabytes
`CHAR(n)`	`CHAR(n)`	2000 bytes
`DATE`	`DATE`	Date and time to hundredths of a second
`DECIMAL(p,s)`	`NUMBER(p,s)`	1×10^{38}
`DOUBLE PRECISION`	`FLOAT(126)`	1×10^{26}
`FLOAT`	`FLOAT(126)`	1×10^{26}
`INT`	`NUMBER(38)`	1×10^{38}
`INTEGER`	`NUMBER(38)`	1×10^{38}
`NUMERIC(p,s)`	`NUMBER(p,s)`	1×10^{38}
`REAL`	`FLOAT(63)`	1×10^{63}
`SMALLINT`	`NUMBER(38)`	1×10^{38}
`TIMESTAMP`	`TIMESTAMP(6)`	Date and time to hundredths of a second
`VARCHAR(n)`	`VARCHAR2(n)`	4000 bytes

See text for explanation of parenthetical numbers/characters

TABLE 6-1. *Oracle ANSI SQL Data Type Map*

It displays the following:

```
Name                           Null?    Type
------------------------------ -------- --------------
COLUMN_NAME                             NUMBER(12,2)
```

The preceding syntax creates a one column table with a single numeric column. The column value is optional because it doesn't have a column `NOT NULL` constraint, which would appear under the `Null?` title header. That means you could insert a row in the table that consists of only null values.

You can create a table with a mandatory column by adding a `NOT NULL` constraint on the column. The constraint can be added as an in-line or out-of-line constraint. Here's the in-line constraint syntax for a mandatory column with a system generated constraint name:

```
SQL> CREATE TABLE sample_number
  2  (column_name NUMBER(12,2) NOT NULL);
```

The table with a `NOT NULL` constrained column looks like this:

```
Name Null? Type
------------------------------ -------- --------------
COLULMN_NAME NOT NULL NUMBER(12,2)
```

As a matter of best practice, it is always better to name constraints. You would use a different syntax to create a table with a named NOT NULL constraint. Named constraints are much easier to find when you explore the Oracle 11*g* catalog. An example of the type of error raised without a constraint name appears later in this section for a CHECK constraint. The syntax for a named constraint is as follows:

```
SQL> CREATE TABLE sample_number
  2  (column_name NUMBER(12,2)  CONSTRAINT nn_sample1  NOT NULL);
```

The current table requires a value, which means you can't insert a null value. A 12-digit number with two placeholders to the right of the decimal point follows this pattern:

```
SQL> INSERT INTO sample_number
  2  VALUES ( 1234567890.99 );
```

A different rule applies to the BINARY_FLOAT and FLOAT data types. They have only a precision value and no scale. You can assign scales dynamically, or the values to the right of the decimal point can vary. That's because the nature of a floating decimal point allows for dynamic values to the right of the decimal point.

You would define a floating data type like this:

```
SQL> CREATE TABLE sample_float
  2  (column_name FLOAT(12));
```

Inserting values follows this pattern:

```
SQL> INSERT INTO sample_float
  2  VALUES (12345678.0099);
```

Oracle doesn't natively support an unsigned integer (positive integers), which MySQL supports. By design, Oracle supports both positive and negative numbers in all numeric data types. You can create a numeric data type and then use a CHECK constraint to implement the equivalent of an unsigned integer. The following table design shows that technique with an in-line constraint:

```
SQL> CREATE TABLE unsigned_int
  2  ( column_name NUMBER(38,0) CHECK (column_name >= 0));
```

This column definition allows entry of only a zero or positive integer. The check constraint is entered as an in-line constraint because it affects only a single column. CHECK constraints must be entered as out-of-line constraints when they work with multiple columns. Multiple column constraints are table constraints. An exception is raised if you attempt to insert a negative integer, like this:

```
INSERT INTO unsigned_int VALUES (-1)
*
ERROR at line 1:
ORA-02290: check constraint (SCHEMA_NAME.SYS_C0020070) violated
```

This type of error message isn't as helpful as a named constraint, but you should know how to read it. The SCHEMA_NAME (synonymous with the user name) is the first element of the error, and the system-generated constraint name is the second element.

You can name CHECK constraints in Oracle like this:

```
SQL> CREATE TABLE unsigned_int
  2  ( column_name NUMBER(38,0)
  3    CONSTRAINT ck_unsigned_int_01 CHECK (column_name >= 0));
```

Although the constraint drops down to line 3, there is no comma separating the definition of the column from the constraint. This means the constraint is an in-line constraint. An out-of-line constraint in this example would differ only by having a comma in line 3, like this:

```
SQL> CREATE TABLE unsigned_int
  2  ( column_name NUMBER(38,0)
  3  , CONSTRAINT ck_unsigned_int_01 CHECK (column_name >= 0));
```

Then the same error is raised with this message:

```
INSERT INTO unsigned_int VALUES (-1)
*
ERROR at line 1:
ORA-02290: check constraint (STUDENT.CK_UNSIGNED_INT_01) violated
```

By incorporating the name of the table in the constraint, you can immediately identify the violation without having to read the data catalog to associate it with a table and business rule.

Date Data Type Dates and timestamps are date types. Their implementation is through complex or real numbers. The integer value represents the date and the decimal value implements time. The range of dates or timestamps is an *epoch*. An epoch is a set of possible dates and date-times that are valid in the database server.

The DATE data type in an Oracle database is a date-time value. As such, you can assign a date-time that is accurate to hundredths of a second. The default date format mask in an Oracle database is dd-mon-rr or dd-mon-yyyy. The rr stands for relative date, and the database server chooses whether the date belongs in the current or last century. The yyyy format mask requires that you explicitly assign the four-digit year to dates.

Here's the syntax to create a DATE column:

```
SQL> CREATE TABLE sample_date
  2  ( column_name DATE DEFUALT SYSDATE);
```

A DATE data type can be assigned a date-time that is equal to midnight by enclosing the date column in a TRUNC function. The TRUNC function shaves off any decimal value from the date-time value. The SYSDATE is a current date-time function available inside an Oracle database.

A more accurate timestamp and timestamps with local or general time zone are also available in Oracle 11*g*. They're more accurate because they measure time beyond hundredths of a second. You would define a TIMESTAMP like this:

```
SQL> CREATE TABLE sample_timestamp
  2  ( column_name TIMESTAMP DEFUALT SYSTIMESTAMP);
```

You also have INTERVAL DAY TO SECOND and INTERVAL DAY TO MONTH data types. They measure intervening time (like the number of minutes or seconds between two timestamps), which is similar to the TIME data type in MySQL.

Character Data Type Character data types have several subgroups in an Oracle database. They can be summarized as fixed-length, long, Unicode, row identifiers, and variable-length strings. In all cases, character data types work very much alike. You specify how many characters you plan to store as the maximum number.

This is the syntax for a fixed-length string:

```
SQL> CREATE TABLE variable_string
  2  ( column_name  CHAR(20) CONSTRAINT nn_varstr_01 NOT NULL);
```

You can insert or update a fixed-length string with any number of characters up to the maximum number but the database inserts whitespace after the last character up to the maximum number of characters. That's why you have the RTRIM function to strip trailing characters when they're returned in a query result.

This variable length string is the equivalent:

```
SQL> CREATE TABLE variable_string
  2  ( column_name  VARCHAR2(20) CONSTRAINT nn_varstr_01 NOT NULL);
```

This definition allocates 20 bytes of space. An alternative syntax lets you define space by the number of characters, which supports Unicode strings. That syntax requires including a CHAR flag inside the parentheses. Here's the syntax for a fixed-length string:

```
SQL> CREATE TABLE variable_string
  2  ( column_name  CHAR(20 CHAR) CONSTRAINT nn_varstr_01 NOT NULL);
```

The variable length equivalent is shown here:

```
SQL> CREATE TABLE variable_string
  2  ( column_name  VARCHAR2(20 CHAR) CONSTRAINT nn_varstr_01 NOT NULL);
```

The Oracle database also includes national language character types. They are designed to store Unicode and different character sets in the same database. The syntax for NCHAR or NVARCHAR2 uses the same definition pattern. You specify the maximum size of the string in bytes or characters inside parentheses. CHAR should be used when you define Unicode columns.

The maximum size of a fixed-length CHAR variable is 2000 bytes, and the maximum length of a VARCHAR or VARCHAR2 is 4000 bytes. Beyond that, you can implement a LONG data type, which is 32,767 (the positive portion of 2^{16}), but it's soon to be deprecated. The last choice for a very long string is a Character Large Object (CLOB), which has a maximum size of 8 terabytes when the block size is 8 kilobytes and 32 terabytes when the block size is 32 kilobytes. Oracle makes these large object types available through a built-in API, which you must use to work with these data types.

Large Strings You can define a table with a CLOB or NCLOB column similar to the following example, but it lacks a physical maximum size and name for the storage clause. The physical size for a CLOB is set by the configuration of the database. As mentioned, the maximum size is set by the block size and is typically 8 terabytes. The Oracle database assigns a system generated name, which can make calculating and maintaining its storage difficult for DBAs. This is true any time you leave something to an implicit behavior, such as naming a constraint or internal storage

```
SQL> CREATE TABLE clob_table
  2  ( column_id   NUMBER
  3  , column_name CLOB DEFAULT '');
```

The DEFAULT keyword assigns an empty string to the column_name column, which is equivalent to an INSERT or UPDATE statement putting a call to EMPTY_CLOB() in the column as its value. The DEFAULT value places the default value in a column only when the INSERT statement uses an overriding signature that excludes the column_name column, like so:

```
SQL> INSERT INTO clob_table (column_id) VALUES (1);
```

You can discover that storage clause by using the DBMS_METADATA package and the GET_DDL function. This does the same thing as a SHOW CREATE TABLE statement in a MySQL database. It reads the data catalog and provides the complete syntax for creating the table. The easiest way to get this information is to run the command from the SQL*Plus prompt. You'll need to expand the default 80 characters of space allotted for displaying a LONG data type before you run the query. Also, you'll need to remember that Oracle stores all metadata strings in uppercase, which means the actual parameters (or arguments) to the GET_DDL function must be in uppercase for it to work.

Here are the SQL*Plus and SQL commands:

```
-- Reset the display value for a large string.
SET LONG 300000
-- Query the data catalog for the full create statement.
SELECT dbms_metadata.get_ddl('TABLE', 'CLOB_TABLE') FROM dual;
```

Some liberty has been taken with reformatting the output to increase readability of it for the book, but this is what would be returned from the query:

```
CREATE TABLE "STUDENT"."CLOB_TABLE"
  ("COLUMN_NAME" CLOB DEFAULT '')
  PCTFREE 10 PCTUSED 40 INITRANS 1 MAXTRANS 255
  NOCOMPRESS NOLOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
          PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "USERS"
  LOB ("COLUMN_NAME") STORE AS BASICFILE
  (TABLESPACE "USERS"
   ENABLE STORAGE IN ROW CHUNK 8192 RETENTION NOCACHE LOGGING
   STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
           PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT))
```

The better way to define a CLOB or BLOB column includes some explicit syntax and a name for the BASICFILE. A BASICFILE means that the CLOB isn't encrypted. The other option in this syntax creates an encrypted file with the SECUREFILE keyword. You should note that a SECUREFILE clause requires that you store the CLOB or BLOB column in a tablespace with automatic segment-space management.

The right way to use a `CREATE TABLE` statement with a LOB is to name the `CLOB` or `BLOB` file. This extra step makes your life easier later, as described in the "`XMLTYPE` Data Type" section a bit later.

Here's the syntax that assigns a user-defined `SEGMENT_NAME` at table creation:

```
SQL> CREATE TABLE sample_table
  2  ( column_name CLOB DEFAULT '')
  3  LOB (column_name) STORE AS sample_table_clob
  4    (TABLESPACE users ENABLE STORAGE IN ROW CHUNK 32768
  5     PCTVERSION 10 NOCACHE NOLOGGING
  6     STORAGE (INITIAL 1048576
  7              NEXT 1048576
  8              MINEXTENTS 1
  9              MAXEXTENTS 2147483645));
```

The additional four parameters in the storage clause will be appended by using the default values as noted in the output from the preceding call to the `GET_DDL` function. The company's DBA should provide guidelines on the `STORAGE` clause settings.

TIP
The default creates `BLOB` and `CLOB` storage with logging enabled. As a rule, these types should have logging turned off.

Boolean It is possible to mimic a Boolean data type in an Oracle database. To accomplish this, you define a number column and assign a table-level constraint on that column. The constraint would allow only a `0` (for false) or `1` (for true). The syntax to implement a column that performs like a Boolean data type is shown next:

```
SQL> CREATE TABLE sample_boolean
  2  ( column_name NUMBER
  3  , CONSTRAINT boolean_values
  4    CHECK (column_name = 0 OR column_name = 1));
```

This type of column would allow only a null, 0, or 1 to be inserted into the table. Anything else would trigger a constraint violation error, like this:

```
INSERT INTO sample_boolean VALUES (2)
*
ERROR at line 1:
ORA-02290: check constraint (STUDENT.BOOLEAN_VALUES) violated
```

After dropping the original `sample_boolean` table, you can re-create it with the addition of a `NOT NULL` constraint. Now it implements two-valued logic, because it disallows the insertion of a null value.

```
SQL> CREATE TABLE sample_boolean
  2  ( column_name NUMBER CONSTRAINT no_null NOT NULL
  3  , CONSTRAINT boolean_values
  4    CHECK (column_name = 0 or column_name = 1));
```

You can also use other comparison operators such as greater than, less than, greater than or equal to, and so forth. It is also possible to use SQL lookup operators such as `IN`, `=ANY`, `=SOME`, or `=ALL`.

BLOB Data Type The `BLOB` data type is very much like the `CLOB` data type. You can store a binary signature for an image or document inside a `BLOB` data type. The maximum size is 8 to 32 terabytes and the behaviors and syntax mirror those for the `CLOB` data type. I won't repeat the syntax here since it is the same `BLOB` data type already discussed.

XMLTYPE Data Type The `XMLTYPE` is a specialized form of a `CLOB` data type. You use it as the column data type, but then you provide a specialized storage clause that identifies its storage as a `CLOB` data type. This process follows the pattern that Oracle databases use consistently between subtypes and types.

Here's the syntax to create an `XMLTYPE` column:

```
SQL> CREATE TABLE item
  2  ( item_id          NUMBER CONSTRAINT pk_item PRIMARY KEY
  3  , item_title       VARCHAR2(30) CONSTRAINT nn_item_01 NOT NULL
  4  , item_description XMLTYPE)
  5  XMLTYPE item_description STORE AS CLOB item_desc_clob
  6  ( TABLESPACE some_tablespace_name
  7    STORAGE (INITIAL 819200 NEXT 819200 )
  8    CHUNK 8192 NOCACHE LOGGING);
```

Notice that line 5 has the storage clause highlighted. That's because a lot of the Oracle documentation simply instructs you to use `STORE AS CLOB` clause. Unfortunately, that fails to provide a meaningful storage name for matching segments to LOBs in the data catalog. You join the `ALL_`, `DBA_`, and `USER_LOBS` view to the equivalent `USER_SEGMENTS` view on the `SEGMENT_NAME` column value. The name of the `CLOB` in the storage clause facilitates that join. System-generated names can be matched, but that match requires a complex regular expression in the SQL syntax. It's better to provide a name and simplify the DBA's life upfront.

The use of XML inside the Oracle database continues to grow release by release. The XML Developer's Kit (XDK) for Oracle is complex and an awesome resource to delve into when you're going to use XML inside the Oracle database. I'd recommend you start by reading the *Oracle Database Developer's Kit Programmer's Guide*.

Virtual Columns A virtual column lets you create what are sometimes known as *derived columns*. You need to decide whether virtual or derived best describes them for you, but Oracle has opted for *virtual*, and that's what might crop up on a certification exam. A virtual column lets you store a formula that joins strings from other columns in the same row together or calculate values—the functions are stored in the table, not the values.

The following demonstrates the syntax to create a virtual column that concatenates strings:

```
SQL> CREATE TABLE employee
  2  ( employee_id NUMBER
  3  , first_name  VARCHAR2(20)
  4  , last_name   VARCHAR2(20)
  5  , full_name   VARCHAR2(40) AS (first_name || ' ' || last_name));
```

Line 5 is the virtual column. Instead of a column data type, two columns are joined together with white space in between.

The next example demonstrates a virtual column that uses math operations against values in other columns:

```
SQL> CREATE TABLE salary
  2  ( salary_id    NUMBER       CONSTRAINT pk_salary PRIMARY KEY
  3  , salary       NUMBER(15,2) CONSTRAINT nn_salary_01 NOT NULL
  4  , bonus        NUMBER(15,2)
  5  , compensation NUMBER(15,2) AS (salary + bonus));
```

Line 5 in this example shows you how to use a math operation in a virtual column. Virtual columns are marked in the database catalog. You can display the `VIRTUAL_COLUMN` in the `ALL_`, `DBA_`, or `USER_TAB_COLS` view to see if a column is virtual. Any column that has a `'YES'` in that column is a virtual column. The formula for the virtual column is in the `DATA_DEFAULT` column of the same view.

Nested Table Data Types This section shows you how to create single-level and multiple-level nested tables. Oracle Database 11*g* supports nested table data types, but MySQL database doesn't. Oracle documentation calls the nesting of tables within nested tables a *multi-level collection*.

Nested tables are tables within a column inside a row of another table. These are object types defined in SQL before they can become data types for columns in a table. The definition can be a one- or two-step process. A one-step process occurs when you create an object type that is a collection of a scalar data type. A two-step process occurs when you create an object type that contains a set of variables. The set of variables in this case becomes a *record structure*. Elements of record structures are *fields*. You create an object type (or record structure) as a schema-level object type, and then you create a collection of the object type.

As an object relational database management system (ORDBMS), Oracle databases let you define object types. Object types have two roles in Oracle databases: One role is as a SQL data type, which you see in this chapter as a nested table. The other acts like a traditional object type inside an object-oriented programming language (OOPL) such as Java, C#, or C++. Instantiable object types are advanced PL/SQL concepts. You can find a full description of how you can define and work with them in Chapter 14 of *Oracle Database 11*g *PL/SQL Programming*.

Array or List?

An array or list is a collection of the same type of data, which can be a scalar data type or record structure. An array is a structure that has a fixed maximum number of elements that is uniquely indexed by a sequential set of numbers. A sequential set of numbers is also known as a *densely populated index*. Programmers iterate (move across arrays one-by-one) using sequential index values.

A list is like an array but different. A list has no maximum number of elements and can be indexed by a sequential or nonsequential index. A nonsequential set of numbers or strings acts like a sparsely populated index. Programmers must iterate through elements of the list by using an iterator to traverse the links between each element from the first until the last. This behavior is similar to a singly linked list in the C/C++ programming languages.

The elements in arrays or lists can be ordered or unordered. More often than not, arrays are ordered sets. Lists are more frequently unordered sets. The closest corollary to a database table is an unordered list.

Two possible syntaxes can be used for creating a collection of a scalar data type. One creates an array, which has a fixed size; the other creates a list, which has no fixed size. The array is created with this syntax:

```
SQL> CREATE TYPE street_array IS VARRAY(3) OF VARCHAR2(30);
  2  /
```

This creates an array of no more than *three* 30-character variable length strings. The forward slash is required to execute the statement terminated by a semicolon. This is a case where the semicolon acts like a statement terminator, not an execution command.

Alternatively, you can create a list of 30 character variable length strings with this:

```
SQL> CREATE TYPE street_list IS TABLE OF VARCHAR2(30);
  2  /
```

You can then define a table that uses the user-defined type `STREET_LIST` as a column data type. The following defines an `ADDRESS` table by using it to capture one to however many street addresses might be required by an address:

```
SQL> CREATE TABLE address
  2  ( address_id     NUMBER
  3  , street_address STREET_LIST
  4  , city           VARCHAR2(30)
  5  , state          VARCHAR2(2)
  6  , postal_code    VARCHAR2(10))
  7  NESTED TABLE street_address STORE AS street_table;
```

The `STREET_LIST` collection is a column data type on line 3 and requires that you add a `NESTED TABLE` clause to the `CREATE TABLE` statement. Line 7 defines the storage name of `STREET_TABLE`. Chapter 8 shows examples of how you insert into nested tables.

A question some ask is, "How do you create a nested table within a nested table?" Oracle's documentation labels nesting a table within a table as a *multi-level collection*. The most difficult part of nesting a table within a nested table is learning how to write the storage clause. Then, the storage clause syntax is straightforward.

The following extends the original design by changing the `ADDRESS` table into an object type and then nesting it in an `EMPLOYEE` table. The first step in this example creates an `ADDRESS_TYPE`, like this:

```
SQL> CREATE OR REPLACE TYPE address_type AS OBJECT
  2  ( address_id     NUMBER
  3  , street_address STREET_LIST
  4  , city           VARCHAR2(30)
  5  , state          VARCHAR2(2)
  6  , postal_code    VARCHAR2(10));
  7  /
```

You create a list collection of the `ADDRESS_TYPE` with the following syntax:

```
SQL> CREATE OR REPLACE TYPE address_list AS TABLE OF address_type;
  2  /
```

The EMPLOYEE table holds a nested table of the ADDRESS_TYPE, which in turn holds a nested table of STREET_ADDRESS, as shown next:

```
SQL> CREATE TABLE employee
  2  ( employee_id    NUMBER
  3  , first_name     VARCHAR2(20)
  4  , middle_name    VARCHAR2(20)
  5  , last_name      VARCHAR2(20)
  6  , home_address   ADDRESS_LIST)
  7  NESTED TABLE home_address STORE AS address_table
  8  (NESTED TABLE street_address STORE AS street_table);
```

Line 7 defines the nested table storage for the ADDRESS_TYPE associated with the HOME_ADDRESS column. The parentheses on line 8 indicate that the nested table for STREET_ADDRESS is part of the previous nested table. There's an internally managed link that connects the EMPLOYEE table with the nested HOME_ADDRESS table, and another link that connects the STREET_ADDRESS table to the HOME_ADDRESS column (or nested table).

Although you've seen how to implement nested tables, they're complex and not as flexible to changing business requirements. Use them only when they meet a specific need that can't be met by normal primary-to-foreign key relationships. For more information on nested tables, check the *Oracle Database 11*g *SQL Language Reference*.

Column and Table Constraints

There are two key differences between column and table constraints. One difference is that a column constraint effects only a single column in the table. The other difference is that a column constraint can be done as an in-line constraint, whereas a table constraint must be done as an out-of-line constraint. That's a natural thing if you pause to think about why that's true. A table constraint applies to multiple columns and in-line constraints only apply to the single column that shares the same line.

The Nested Table Design Pattern

Nested tables are an advanced implementation made possible by the Oracle Object Relational Model (ORM). They provide an internal connection between tables that acts like an inner class in OOPLs.

Like inner classes in OOPL, there's no way to go directly to the inner class. You must first go through the outer (or container) class. This type of relationship between tables in database design is known as an *ID-dependent relationship*. The only way to discover the ID is through the table holding the key to the nested tables.

If subsequent discovery of the business model identifies another use for nested table data, the design must be changed to open up access to the nested data. That means removing the nested table and making it an ordinary table connected by a primary-to-foreign key relationship. That type of change is typically expensive. Ordinary table primary-to-foreign key relationships are more flexible (and for the curious, their official label is *non–ID-dependent relationships*).

As mentioned, the only constraint that is always a column constraint is the `NOT NULL` constraint. The remaining Oracle constraints, `CHECK`, `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints can be column or table constraints. You can find coverage of the `NOT NULL` constraint in the "Number Data Type" section earlier in this chapter. The biggest difference between `NOT NULL` constraints in an Oracle database versus a MySQL database is that you can name them in Oracle but can't in MySQL. The reason isn't immediately clear, but MySQL treats whether a column is nullable or not null as a property of the column. That's why they're not in the `TABLE_CONSTRAINTS` view of the `INFORMATION_SCHEMA`.

Column Constraints In-line column constraints are most frequently a `PRIMARY KEY` or `NOT NULL` constraint. The `PRIMARY KEY` works in that context when a surrogate key or a single column natural key fills the role of a primary key. Surrogate keys typically follow a naming pattern that takes the name of the table and adds an `_ID` as a suffix. A `PRIMARY KEY` constraint on a single column makes the column both not null and unique within the table. `NOT NULL` constraints apply to single columns only, and they make entry of a column value mandatory when you insert a row or update a row. The only other column constraint is a `CHECK` constraint in Oracle and the `ENUM` and `SET` data types that mimic a `CHECK` constraint in MySQL.

Oracle supports naming or using a system-generated name for a `PRIMARY KEY` constraint. The following syntax demonstrates how to define a `PRIMARY KEY` constraint with a system-generated constraint name:

```
SQL> CREATE TABLE sample
  2  ( sample_id    NUMBER PRIMARY KEY
  3  , sample_text  VARCHAR2(30));
```

You can name the constraint by using this syntax:

```
SQL> CREATE TABLE sample
  2  ( sample_id    NUMBER CONSTRAINT pk_sample PRIMARY KEY
  3  , sample_text  VARCHAR2(30));
```

A `FOREIGN KEY` constraint works as a column constraint much like the `PRIMARY KEY` constraint but it has a dependency. You can define only a single column `FOREIGN KEY` that references (points to) a primary key constrained column. The primary key column can be in the same or a different table. The more common pattern is a different table.

For the following example, assume a `SYSTEM_USER` table has a `SYSTEM_USER_ID` primary key constrained column. The example builds on the previous `SAMPLE` table by adding a `CREATED_BY` column, which should contain a value from the primary key column of the `SYSTEM_USER` table. Here's the syntax for an in-line `FOREIGN KEY` constraint:

```
SQL> CREATE TABLE sample
  2  ( sample_id    NUMBER CONSTRAINT pk_sample PRIMARY KEY
  3  , sample_text  VARCHAR2(30)
  4  , created_by   NUMBER REFERENCES system_user(system_user_id));
```

Line 4 adds a `FOREIGN KEY` constraint to the `SAMPLE` table with a system-generated name. Notice that there is no use of the foreign key as a label when creating an in-line `FOREIGN KEY` constraint. The `FOREIGN KEY` has two formal parameters: the name of a table followed by the column name in parentheses.

Here's the alternative syntax that names the FOREIGN KEY constraint:

```
SQL> CREATE TABLE sample
  2  ( sample_id     NUMBER CONSTRAINT pk_sample PRIMARY KEY
  3  , sample_text   VARCHAR2(30)
  4  , created_by    NUMBER
  5    CONSTRAINT fk_sample_01 REFERENCES system_user(system_user_id));
```

A CHECK constraint also works as a column constraint. The following syntax shows how to constrain a one-character column to contain a gender value.

```
SQL> CREATE TABLE club_member
  2  ( club_member_id  NUMBER PRIMARY KEY
  3  , first_name      VARCHAR2(30)
  4  , last_name       VARCHAR2(30)
  5  , gender          CHAR(1) DEFAULT 'F' CHECK (gender ('M', 'F')));
```

This syntax creates the CHECK constraint with a system-generated constraint name. It assigns a default value of an 'F' to the gender column when an INSERT statement excludes the column during an insert operation. Excluding the column differs from inserting a null value. A null value in the VALUES clause of an INSERT statement or SET clause of an UPDATE statement overrides the default value and inserts a null value. As an alternative, you could place a NOT NULL constraint on the column to prevent a null value insertion or update, but that removes any value of the DEFAULT phrase.

You can add a constraint name by shifting the constraint from line 5 to a new line 6, as shown here:

```
  5  , gender          CHAR(1) DEFAULT 'F'
  6    CONSTRAINT ck_club_member_01 CHECK (gender IN ('M', 'F')));
```

A UNIQUE constraint typically spans multiple columns but can be applied to a single column. The following example shows how you would define an in-line UNIQUE constraint with a system generated name that includes an automotive VIN (vehicle identification number):

```
SQL> CREATE TABLE vehicle
  2  ( vehicle_id    NUMBER
  3  , vin           VARCHAR2(17) UNIQUE
  4  , make          VARCHAR2(30)
  5  , model         VARCHAR2(30)
  6  , year          VARCHAR2(4));
```

Changing the definition to include a user-defined constraint name, you would modify line 3 to this:

```
  3  , vin           VARCHAR2(17) CONSTRAINT un_vehicle_01 UNIQUE
```

UNIQUE constraints automatically create indexes. This is important only when you're trying to drop an index and can't. Then, you can discover the implicitly created indexes by querying the ALL_, DBA_, or USER_INDEXES and USER_IND_COLUMNS views. Joining the two views gives you all the columns in a unique index. Note that an index created by a UNIQUE constraint can't be dropped independently of the table.

There is also a REF constraint for objects. Although many don't store objects in tables, those that do need the syntax. A REF constraint is more limiting than the nested tables solution, because it disallows nested tables within nested tables—at least, it disallows them when they're collections of scalar data types.

The following example leverages the discussion from the "Nested Table Data Types" section earlier. You create an ADDRESS_TYPES object with a flattened set of columns in place of the list of possible street addresses. Here's the syntax:

```
CREATE OR REPLACE TYPE address_type AS OBJECT
( address_id       NUMBER
, street_address_1 VARCHAR2(20)
, street_address_2 VARCHAR2(20)
, street_address_3 VARCHAR2(20)
, city             VARCHAR2(30)
, state            VARCHAR2(2)
, postal_code      VARCHAR2(10));
```

You then create a table by using the definition of the object type, like this:

```
CREATE TABLE address_list OF address_type;
```

You create a table that uses the collection stored in the ADDRESS_LIST table with the following syntax:

```
SQL> CREATE TABLE employee
  2  ( employee_id    NUMBER
  3  , first_name     VARCHAR2(20)
  4  , middle_name    VARCHAR2(20)
  5  , last_name      VARCHAR2(20)
  6  , home_address   REF address_type SCOPE IS address_list);
```

Line 6 contains the column that uses a reference constraint to find the values for the HOME_ADDRESS column. The SCOPE keyword points to the reference table that uses the object type. The advantage of this approach is that the referenced table can have data inserted and deleted without touching the EMPLOYEE table.

Table Constraints Table constraints can apply to a single column or multiple columns. That's because table constraints are defined by their position in the CREATE TABLE statement syntax. They occur after the last column definition. However, that really makes them out-of-line constraints. Table constraints apply against multiple columns or impose a unique constraint across rows.

While seldom written as an out-of-line constraint, you can write a single column PRIMARY KEY that way. The reason this is uncommon is that a single PRIMARY KEY constraint imposes both a NOT NULL and UNIQUE column constraint, and they're the only constraints that would displace an in-line PRIMARY KEY. The syntax requires that you provide the column name as an argument to the constraint. You have two options, as with in-line constraints: one uses a system-generated constraint name and the other uses a user-defined constraint name.

Here's an out-of-line PRIMARY KEY constraint with a system-generated constraint name:

```
, PRIMARY KEY(system_user_id)
```

And here's the same constraint with a user-defined name:

```
, CONSTRAINT pk_system_user PRIMARY KEY(system_user_id)
```

A multiple column `PRIMARY KEY` column would look like this:

```
, CONSTRAINT pk_contact PRIMARY KEY(first_name, last_name)
```

Check constraints often occur as out-of-line constraints because the columns involved can hold other in-line constraints, such as a `NOT NULL` constraint. A `CHECK` constraint with a system-generated constraint name looks like this:

```
, CHECK(salary > 0 AND salary < 50000)
```

Adding a user-defined name, the constraint looks like this:

```
, CONSTRAINT ck_employee_01 CHECK(salary > 0 AND salary < 50000)
```

A table constraint on multiple columns would look like the following:

```
, CONSTRAINT ck_employee_02 CHECK
  ((salary BETWEEN      0 AND  49999.99 AND employee_class = 'NON-EXEMPT') OR
(salary BETWEEN  50000 AND 249999.99 AND employee_class = 'EXEMPT')
OR (salary BETWEEN 250000 AND 999999.99 AND employee_class = 'EXECUTIVE'))
```

You create a `UNIQUE` constraint as an out-of-line constraint in two cases. The first case occurs when you've applied a `NOT NULL` in-line constraint and you want to create a `UNIQUE` single-column constraint. The second is when you want to create a multiple-column `UNIQUE` constraint.

Here's the syntax for a single-column `UNIQUE` constraint:

```
, UNIQUE(common_lookup_id)
```

The multiple-column syntax isn't much different for a `UNIQUE` constraint. The only difference is a comma-delimited list of column names in lieu of a single column. This syntax example includes a user-defined constraint name:

```
, CONSTRAINT un_lookup
  UNIQUE(common_lookup_table, common_lookup_column, common_lookup_type)
```

The most popular table-level constraint is a `FOREIGN KEY` constraint. You must provide the `FOREIGN KEY` phrase in an out-of-line constraint, unlike the in-line version that starts with the `REFERENCES` keyword. Here's an example of a `FOREIGN KEY` constraint without a user-defined constraint name:

```
, FOREIGN KEY(system_id) REFERENCES common_lookup(common_lookup_id)
```

The `SYSTEM_ID` column name identifies the column in the table that becomes constrained by the `FOREIGN KEY`. The `REFERENCES` clause identifies the table and column inside the parentheses where the constraint looks to find the list of primary key values.

You would add a `CONSTRAINT` keyword and a user-defined constraint name when you name a `FOREIGN KEY` constraint. The syntax would look like this:

```
, CONSTRAINT fk_system_01 FOREIGN KEY(system_id)
  REFERENCES common_lookup(common_lookup_id)
```

Like a `UNIQUE` constraint, you can provide a single-column reference or a comma-delimited list of columns. Single-column `FOREIGN KEY` constraints generally refer to surrogate keys that are generated values from a sequence. Multiple-column `FOREIGN KEY` values relate to the multiple-column *natural key* of the table.

This concludes how you define constraints in the `CREATE TABLE` statement syntax. You also have the option of adding, dropping, disabling, or enabling constraints with the `ALTER TABLE` statement. Chapter 7 covers how you use the `ALTER TABLE` statement. The next section shows you the syntax to create externally organized tables.

Externally Organized Tables

Oracle lets you define externally organized tables. Externally organized tables appear like ordinary tables in the database but are structures that read-only or read and write files from the operating system. Read-only files can be comma-separated files (CSVs) or position-specific files. Read and write files are stored in an Oracle Data Pump proprietary format. However, only non-proprietary file formats are known as *flat files*.

Oracle SQL*Loader lets you read these flat files with a `SELECT` statement from what appear as standard tables. Oracle Data Pump lets you read with a `SELECT` statement like Oracle SQL*Loader. Oracle Data Pump also lets you write with an `INSERT` statement. The write creates a proprietary formatted file, and the read extracts the data from the file.

Two key preparation steps are required whether you're working with externally organized read-only or read-write files. These steps help you create virtual directories and grant database privileges to read from and write to them. The first subsection shows you those preparation steps, and the next two show you how to work with read-only and read-write files.

Virtual Directories Virtual directories are structures in the Oracle database, and they're stored in the data catalog. They map virtual directory names to physical operating system directories. Virtual directories make a few assumptions, which can become critical fail points. For the database grants to work successfully, the physical directories must be accessible to the operating system user who installed the Oracle Server. That means the operating system user should have read and write privileges to the related physical directories.

As the `SYS`, `SYSTEM`, or authorized administrator account, you create a virtual directory with the following syntax:

```
SQL> CREATE DIRECTORY upload AS 'C:\Data\Upload';
```

After you create the virtual directory, you must grant permissions to read from and write to the directory. This is true whether you're deploying a read-only file or read-write file because both types of files typically write error, discard, and log files to the same directory.

```
SQL> GRANT READ, WRITE ON DIRECTORY upload TO importer;
```

After creating a virtual directory, you find the mapping of virtual directories to operating system directories in the `DBA_DIRECTORIES`. Only a `SYS` or `SYSTEM` super user can gain access to this conceptual view. Unlike many other administrative views, there is no USER_ DIRECTORIES view.

For reference, virtual directories are also used for BFILE data types. Web developers need to know the list of virtual directories and their associated physical directories. They need that information to ensure their programs place the uploaded files where they belong.

Oracle SQL*Loader Files After the preparation steps, you can define an externally organized table that uses a read-only file. Line 6 sets the TYPE value as Oracle SQL*Loader and line 7 sets the DEFAULT DIRECTORY as the virtual directory name you created previously:

```
SQL> CREATE TABLE CHARACTER
  2  ( character_id NUMBER
  3  , first_name VARCHAR2(20)
  4  , last_name VARCHAR2(20))
  5    ORGANIZATION EXTERNAL
  6    ( TYPE oracle_loader
  7      DEFAULT DIRECTORY upload
  8      ACCESS PARAMETERS
  9      ( RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
 10        BADFILE     'UPLOAD':'character.bad'
 11        DISCARDFILE 'UPLOAD':'character.dis'
 12        LOGFILE     'UPLOAD':'character.log'
 13        FIELDS TERMINATED BY ','
 14        OPTIONALLY ENCLOSED BY "'"
 15        MISSING FIELD VALUES ARE NULL )
 16      LOCATION ('character.csv'))
 17  REJECT LIMIT UNLIMITED;
```

Lines 10 through 12 set the virtual directory and log files for any read from the externally organized table. Logs are written with each SELECT statement against the character table when data fails to conform to the definition. After the log file setup, the delimiters define how to read the data in the external file. Line 13 sets the delimiter, FIELD TERMINATED BY, as a comma. Line 14 sets the optional delimiter, OPTIONALLY ENCLOSED BY, as a single quote mark or apostrophe—this is important when you have a comma in a string.

The character file reads a file that follows this format:

```
1,'Indiana','Jones'
2,'Ravenwood','Marion'
3,'Marcus','Brody'
4,'Rene','Belloq'
```

Sometimes, you don't want to use CSV files because you've received position-specific files. That's the case frequently when the information comes from mainframe exports. You can create a position-specific table with the following syntax:

```
SQL> CREATE TABLE grocery
  2  ( grocery_id  NUMBER
  3  , item_name   VARCHAR2(20)
  4  , item_amount NUMBER(4,2))
  5    ORGANIZATION EXTERNAL
  6    ( TYPE oracle_loader
  7      DEFAULT DIRECTORY upload
  8      ACCESS PARAMETERS
  9      ( RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
 10        BADFILE 'UPLOAD':'grocery.bad'
 11        LOGFILE 'UPLOAD':'grocery.log'
 12        FIELDS
 13        MISSING FIELD VALUES ARE NULL
 14        ( grocery_id  CHAR(3)
 15        , item_name   CHAR(20)
 16        , item_amount CHAR(4)))
 17      LOCATION ('grocery.csv'))
 18  REJECT LIMIT UNLIMITED;
```

The major difference between the CSV-enabled table and a positionally organized external table is the source signature on lines 14 through 16. The CHAR data type specifies fixed-length strings, which can be implicitly cast to number data types. When a SELECT statement reads the external source, it casts the values from fixed-length strings to their designated numeric and variable-length string data types.

An alternative position-specific syntax replaces lines 14 through 16 with exact positional references, like this:

```
 14        ( grocery_id  POSITION(1:3)
 15        , item_name   POSITION(4:23)
 16        , item_amount POSITION(24:27)))
```

The casting issue works the same way because POSITION(1:3) expects to find a fixed-length string. The value in the flat file can be cast successfully only when it is a number.

The grocery table reads values from a flat file, like this:

```
1  Apple               1.49
2  Orange              2
```

These are the preferred solutions when importing large amounts of data. Many data imports include values that belong in multiple tables. Import sources that include data for multiple tables are composite import files. Most import source files generally ignore or exclude surrogate key values because they'll change in the new database. Importing the data is important, but taking from the externally managed table into the normalized business model can be tricky. The MERGE statement lets you import data, and based on some logic you can determine whether it's new or existing information. The MERGE statement then lets you insert new information or update rows of existing data, and the MERGE statement is covered in depth in Chapter 12.

Oracle Data Pump Files Oracle Data Pump lets you read and write data in a proprietary format. It is most often used for backup and recovery. You have import files for reading proprietary formatted files and export tables for saving data in a proprietary format.

The next example requires you to create a new download virtual directory and grant the directory read and write permissions. The following creates a table that exports data to an Oracle Data Pump–formatted file:

```
SQL> CREATE TABLE item_export
  2  ORGANIZATION EXTERNAL
  3  ( TYPE oracle_datapump
  4    DEFAULT DIRECTORY download
  5    LOCATION ('item_export.dmp')
  6  ) AS
  7  SELECT   item_id
  8  ,        item_barcode
  9  ,        item_type
 10  ,        item_title
 11  ,        item_subtitle
 12  ,        item_rating
 13  ,        item_rating_agency
 14  ,        item_release_date
 15  ,        created_by
 16  ,        creation_date
 17  ,        last_updated_by
 18  ,        last_update_date
 19  FROM   item;
```

The exporting process with externally organized tables has only one very noticeable problem: It throws a nasty error when the file already exists, like so:

```
CREATE TABLE item_export
*
ERROR at line 1:
ORA-29913: error IN executing ODCIEXTTABLEOPEN callout
ORA-29400: DATA cartridge error
KUP-11012: file item_export.dmp IN C:\DATA\Download already EXISTS
```

My advice on this type of process is that you create an operating system script, a Java application, or a web solution that checks for the existence of the file before inserting data into the `item_export` table. Alternatively, you can create a set of utilities in Java libraries. You deploy the libraries inside the database, and wrap them with PL/SQL function definitions. They can clean up the file system for you by calling them before you query the table. You can check Chapter 15 of *Oracle Database 11*g *PL/SQL Programming* for details on writing and deploying Java libraries on the Oracle database.

NOTE
*Java libraries work only in the Standard or Enterprise Editions of Oracle Database 11*g.

Reversing the process and importing from the external file source isn't as complex. There are a few modifications to the `CREATE TABLE` statement. Here's a sample:

```
SQL> CREATE TABLE item_import
  2  ( item_id             NUMBER
  3  , item_barcode        VARCHAR2(20)
  4  , item_type           NUMBER
  5  , item_title          VARCHAR2(60)
  6  , item_subtitle       VARCHAR2(60)
  7  , item_rating         VARCHAR2(8)
  8  , item_rating_agency  VARCHAR2(4)
, item_release_date   DATE
, created_by          NUMBER
, creation_date       DATE
, last_updated_by     NUMBER
, last_update_date    DATE)
ORGANIZATION EXTERNAL
( TYPE oracle_datapump
  DEFAULT DIRECTORY upload
  LOCATION ('item_export.dmp'));
```

Notice that the table definition mirrors the source file. This means you must know the source before you can define the external table `CREATE TABLE` statement.

Partitioned Tables

Partitioning is the process of breaking up a data source into a series of data sources. Partitioned tables are faster to access and transact against. Partitioning data becomes necessary as the amount of data grows in any table. It speeds the search to find rows and insert, update, or delete rows.

Oracle Database 11*g* supports four types of table partitioning: list, range, hash, and composite partitioning.

List Partitioning A list partition works by identifying a column that contains a value, such as a `STATE` column in an `ADDRESS` table. Partitioning clauses follow the list of columns and constraints.

A list partition could use a `STATE` column, like the following (the complete example is avoided to conserve space, and the three dots represent the balance of partitions not shown):

```
CREATE TABLE franchise
( franchise_id    NUMBER CONSTRAINT pk_franchise PRIMARY KEY
, franchise_name  VARCHAR(20)
, city            VARCHAR(20)
, state           VARCHAR(20))
PARTITION BY LIST(state)
( PARTITION offshore VALUES('Alaska', 'Hawaii')
, PARTITION west VALUES('California', 'Oregon', 'Washington')
, PARTITION desert VALUES ('Arizona','New Mexico')
, PARTITION rockies VALUES ('Colorado', 'Idaho', 'Montana', 'Wyoming')
, ... );
```

This can be used with other values such as ZIP codes with great effect, but the maintenance of list partitioning can be considered costly. Cost occurs when the list of values changes over time. Infrequent change means low cost, while frequent change means high costs. In the latter case, you should consider other partitioning strategies. Although an Oracle database supports partitioning on a variable-length string, MySQL performs list partitioning only on integer columns.

Range Partitioning Range partitioning is very helpful on any column that contains a continuous metric, such as dates or time. It works by stating a minimum set that is less than a certain value, and then a group of sets of higher values until you reach the top most set of values. This type of partition helps you improve performance by letting you search ranges rather than complete data sets. Range partitioning is also available in MySQL.

A range example based on dates could look like this:

```
PARTITION BY RANGE(rental_date)
( PARTITION rental_jan2011
  VALUES LESS THAN TO_DATE('31-JAN-11','DD-MON-YY')
, PARTITION rental_feb2011
  VALUES LESS THAN TO_DATE('28-FEB-11','DD-MON-YY')
, PARTITION rental_mar2011
  VALUES LESS THAN TO_DATE('31-MAR-11','DD-MON-YY')
, ... );
```

The problem with this type of partitioning, however, is that the new months require constant management. Many North American businesses simply add partitions for all months in the year as an annual maintenance task during the holidays in November or December. Companies that opt for bigger range increments reap search and access benefits from range partitioning, while minimizing ongoing maintenance expenses.

Hash Partitioning Hash partitioning is much easier to implement than list or range partitioning. Many DBAs favor it because it avoids the manual maintenance of list and range partitioning. Oracle Database 11*g* documentation recommends that you implement a hash for the following reasons:

- There is no concrete knowledge about how much data maps to a partitioning range.
- The sizes of partitions are unknown at the outset and difficult to balance as data is added to the database.
- A range partition might cluster data in an ineffective way.

This next statement creates eight partitions and stores them respectively in one of the eight tablespaces. The hash partition manages nodes and attempts to balance the distribution of rows across the nodes.

```
PARTITION BY HASH(store)
PARTITIONS 8
STORE IN (tablespace1, tablespace2, tablespace3, tablespace4
         ,tablespace5, tablespace6, tablespace7, tablespace8);
```

As you can imagine the maintenance for this type of partitioning is low. Some DBAs choose this method to get an initial sizing before adopting a list or range partitioning plan. Maximizing the physical resources of the machine ultimately rests with the DBAs who manage the system. Developers need to stand ready to assist DBAs with analysis and syntax support.

Composite Partitioning Composite partitioning requires a partition and subpartition. The composites are combinations of two types of partitioning—typically, list and range partitioning, or range and hash composite partitioning. Which of these you should choose depends on a few considerations. List and range composite partitioning is done for historical information and is well suited for data warehouses. This method lets you partition on unordered or unrelated column values.

A composite partition like this uses the range as the partition and the list as the subpartition, like the following:

```
PARTITION BY RANGE (rental_date)
 SUBPARTITION BY LIST (state)
 (PARTITION FQ1_1999 VALUES LESS THAN (TO_DATE('1-APR-2011','DD-MON-YYYY'))
  (SUBPARTITION offshore VALUES('Alaska', 'Hawaii')
  , SUBPARTITION west VALUES('California', 'Oregon', 'Washington')
  , SUBPARTITION desert VALUES ('Arizona','New Mexico')
  , SUBPARTITION rockies VALUES ('Colorado', 'Idaho', 'Montana', 'Wyoming')
  , ... )
,(PARTITION FQ2_1999 VALUES LESS THAN (TO_DATE('1-APR-2011','DD-MON-YYYY'))
  (SUBPARTITION offshore VALUES('Alaska', 'Hawaii')
  , SUBPARTITION west VALUES('California', 'Oregon', 'Washington')
  , SUBPARTITION desert VALUES ('Arizona','New Mexico')
  , SUBPARTITION rockies VALUES ('Colorado', 'Idaho', 'Montana', 'Wyoming')
  , ... )
, ... )
```

Range and hash composite partitioning is done for historical information when you also need to stripe data. *Striping* is the process of creating an attribute in a table that acts as a natural subtype or separator of data. Users typically view data sets of one subtype, which means organizing the data by stripes (subtypes) can speed access based on user access patterns.

Range is typically the partition and the hash is the subpartition in this composite partitioning schema. The syntax for this type of partition is shown next:

```
PARTITION BY RANGE (rental_date)
 SUBPARTITION BY HASH(store)
  SUBPARTITIONS 8 STORE IN (tablespace1, tablespace2, tablespace3
                           ,tablespace4, tablespace5, tablespace6
                           ,tablespace7, tablespace8)
   ( PARTITION rental_jan2011
     VALUES LESS THAN TO_DATE('31-JAN-11','DD-MON-YY')
   , PARTITION rental_feb2011
     VALUES LESS THAN TO_DATE('28-FEB-11','DD-MON-YY')
   , PARTITION rental_mar2011
     VALUES LESS THAN TO_DATE('31-MAR-11','DD-MON-YY')
   , ... )
```

NOTE
Developers need to understand techniques, but DBAs often have major decision-making authority in partitioning. Partitioning effectively requires an understanding of the underlying choices made by DBAs in organizing the database.

MySQL Tables

Tables in the MySQL database are slightly different from tables in an Oracle database in three ways. One key difference is that MySQL tables support automatic numbering and the sequence is a property of the table. Another key difference is that MySQL lets you define tables to work with any supported engine, while Oracle has only one engine by default. The last key difference is that MySQL tables don't support nested tables. Other differences are driven by the supported data types. MySQL supports only scalar data types, which include large object types. Large objects are large character or binary arrays and don't require special library handling like the Oracle database counterparts. MySQL supports only two composite data types: `ENUM` and `SET`.

Like the Oracle section, this section covers generalized syntax and how to define the following:

- Scalar data type columns
- `ENUM` and `SET` data type columns
- Column and table constraints
- Partitioned tables

Matching the organization of the Oracle sections, at least those that are supported by MySQL, this section uses small single tables to show how you can define scalar data types. Column and table constraints work more or less like those in Oracle when you define tables that use the default InnoDB engine (see Chapter 1).

TIP
You should use the default InnoDB engine, because it's the only engine that is fully transactional. Opting to use another database engine, such as MyISAM, would disallow the use of foreign key constraints.

MySQL provides three options for creating tables. One is a temporary table that acts much like the temporary table in Oracle, because it lasts only for the scope of the session. Another pseudo-temporary table exists when you opt to define a table using the `ENGINE = MEMORY` clause. Tables defined using the Memory engine are available during the runtime instance of the database and discarded when the database is stopped. Memory engine tables aren't transactional and can't be included in an ACID-compliant transaction with InnoDB or MyISAM stored tables. The last option lets you create a table that becomes permanent—at least until you remove it from the database.

The general and basic prototype for the `CREATE TABLE` statement is:

```
CREATE [TEMPORARY] TABLE [database_name.] table_name
( column_name data_type [NOT NULL]
   [{DEFAULT value | AUTO_INCREMENT | UNIQUE [KEY] | [PRIMARY] KEY}]
   [COMMENT 'comment_text'] | [REFERENCE table_name(column_list)]
   [[CONSTRAINT] constraint_name constraint_type]
, column_name data_type [NOT NULL]
   [{DEFAULT value | AUTO_INCREMENT | UNIQUE [KEY] | [PRIMARY] KEY}]
   [COMMENT 'comment_text'] | [REFERENCE table_name(column_list)]
   [[CONSTRAINT] constraint_name constraint_type]
,[...]
,[CONSTRAINT constraint_name constraint_type(column_list)
  [REFERENCES table_name(column_list)]]
,[...]) AUTO_INCREMENT=start_with_value ENGINE=engine_name;
```

You would create a table like this, where the ellipses represents columns and constraints:

```
CREATE TABLE table_name ( ... );
```

You create a temporary table using this syntax:

```
CREATE TEMPORARY TABLE table_name ( ... );
```

The following syntax lets you create an *in memory* table:

```
CREATE TABLE table_name ( ... ) ENGINE = MEMORY;
```

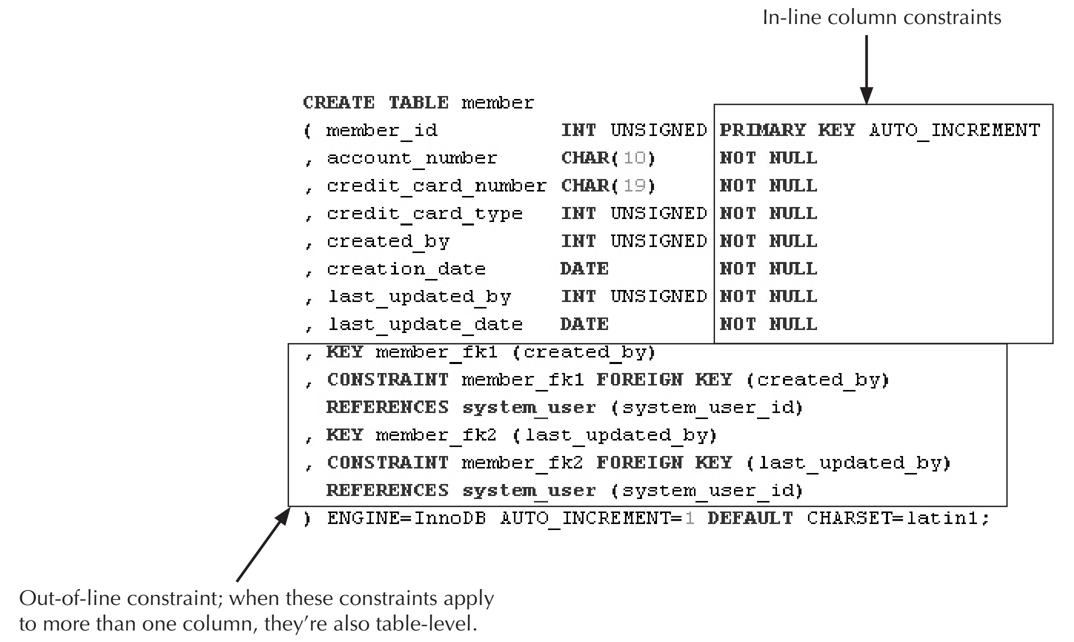
You would set auto incrementing to start at 1001 with this type of syntax:

```
CREATE TABLE table_name ( ... ) AUTO_INCREMENT=1001 ENGINE = MEMORY;
```

The following `SHOW` command lets you examine the catalog's definition of a table:

```
SHOW create table table_name;
```

Figure 6-3 shows you how to create a permanent table. It defines multiple columns and implements in-line and out-of-line database constraints. The figure is annotated to help you visualize possibilities. It does exclude `DEFAULT` values for columns, which are limited to numeric and string literal values in MySQL.


In-line column constraints
Out-of-line constraint; when these constraints apply to more than one column, they're also table-level.

```
CREATE TABLE member
( member_id           INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
, account_number      CHAR(10)     NOT NULL
, credit_card_number  CHAR(19)     NOT NULL
, credit_card_type    INT UNSIGNED NOT NULL
, created_by          INT UNSIGNED NOT NULL
, creation_date       DATE         NOT NULL
, last_updated_by     INT UNSIGNED NOT NULL
, last_update_date    DATE         NOT NULL
, KEY member_fk1 (created_by)
, CONSTRAINT member_fk1 FOREIGN KEY (created_by)
  REFERENCES system_user (system_user_id)
, KEY member_fk2 (last_updated_by)
, CONSTRAINT member_fk2 FOREIGN KEY (last_updated_by)
  REFERENCES system_user (system_user_id)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=latin1;
```

FIGURE 6-3. *MySQL CREATE TABLE statement*

As you see in Figure 6-3, the definition of columns can include in-line constraints, and they precede the definition of out-of-line constraints. The `AUTO_INCREMENT` clause sets an internal sequence as a property of the table. Unless you set the sequence value with an `AUTO_INCREMENT` clause to something other than 1, the sequence always starts with 1.

Unsigned integers are used as the data type for the primary and foreign key columns because surrogate primary keys generally start with 1 and increment by 1. These surrogate keys are automatic numbering sequences.

TIP
MySQL trainers frequently recommend in the MySQL DBA course that you start surrogate keys with unsigned integers, because they take less space (4 bytes) than doubles (8 bytes). You can perform table maintenance to change the integer to a double before you reach the 4.2 billion limitation of an integer data type.

Note that the in-line `NOT NULL` constraints lack user-defined names. That's because MySQL treats `NOT NULL` constraints as table properties, and you can't assign these constraints names. The lack of constraint names is a weakness in the implementation of `NOT NULL` constraints. At least, it's a weakness when you're examining constraints from the Oracle DBA's perspective.

The `KEY` references map to out-of-line `CONSTRAINT` names. This `KEY` line is unnecessary when creating `FOREIGN KEY` constraints. It does, however, find its way into some coding examples on the Internet. They're shown in case you run into them, but I suggest that you leave them out of your code. The best practice is to leave out the `KEY` line for foreign key columns when creating tables.

The next section discusses data types. You can use any of these when you define columns in MySQL tables.

Scalar Data Type Columns

Scalar data types in MySQL include numeric, date, character, and strings. Character data types are fixed length strings and covered in the "Strings Data Type" section. Several possibilities are available in each of these data type groups, and most are qualified in their respective sections.

The largest variation between MySQL and Oracle is that MySQL stores large character strings as `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, and `LONGTEXT` data types, and large binary strings as `TINYBLOB`, `BLOB`, `MEDIUMBLOB`, and `LONGBLOB` data types. Oracle treats these large strings as specialized large objects, which requires separate PL/SQL libraries to manage them. There are no specialized SQL/PSM libraries to manage these large character and binary strings in MySQL.

The following sections cover numeric, Boolean, date, character, and string data types. Boolean data types don't formally exist in SQL for either database. Oracle does support a Boolean data type in their PL/SQL language. This section tells you how to work with integers to support Boolean-like usage in columns as a work around to their absence from SQL. Large character and binary strings are discussed in the "Strings Data Type" section.

Numeric Data Types MySQL supports numeric data types with two groups: decimal and integer data types. Decimal data types support fixed-point and floating-point numbers. Integers support various sizes of integers.

Defining business data frequently involves numbers with decimal points. These are known as *numeric* data types because they support real numbers. The maximum number of digits allowed for this type of number is 65. The maximum number of digits that can appear to the right of the decimal place is 30.

You can define these types of fixed decimal point numbers with a `DEC`, `DECIMAL`, `NUMERIC`, or `FIXED` data type label. The labels are synonymous. With any of these data types, you define a 65-digit integer when you exclude the comma and the number of decimal points.

The following `CREATE TABLE` statement shows examples of defining real number data types:

```
mysql> CREATE TABLE number_table
    -> ( var1  DEC(65,2)
    -> , var2  DECIMAL(65,2)
    -> , var3  NUMERIC(65,2)
    -> , var4  FIXED(65,2));
```

After creating the table, you can describe the table and see that these labels are pseudonyms (aliases) for the `DECIMAL` data type. I recommend that you use the `DECIMAL` label, because it's exportable to other databases, and the others are no more than aliases. You would see the following by describing the table:

```
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| var1  | decimal(65,2) | YES  |     | NULL    |       |
| var2  | decimal(65,2) | YES  |     | NULL    |       |
| var3  | decimal(65,2) | YES  |     | NULL    |       |
| var4  | decimal(65,2) | YES  |     | NULL    |       |
+-------+---------------+------+-----+---------+-------+
```

The total number of digits in a decimal data type is the *precision* (or more plainly the *width*) and the number of possible digits to the right of the decimal point is the *scale*. The value of the scale can never be greater than the value of the precision. These data types create a fixed decimal point storage type. Any attempt to define a column data type with a scale that is larger than the precision throws an ERROR 1427. This error informs you that `FLOAT(M,D)`, `DOUBLE(M,D)`, or `DECIMAL(M,D)`, where `M` must be greater than or equal to `D`.

Sometimes you want to define a data type that accepts numbers with varying decimal sizes. This type of number is a floating-point number. You have two options for floating-point data types: `FLOAT` and `DOUBLE`. The alias for `DOUBLE` is `DOUBLE PRECISION`. The `FLOAT` data type is smaller than the `DOUBLE` data type. According to MySQL documentation, these return approximate numeric data values, and testing finds that the approximation varies depending on the number of digits that appear before and after the decimal point.

You would declare columns of these floating-point numbers like this:

```
CREATE TABLE floating
( var1  FLOAT
, var2  DOUBLE );
```

Alternative syntax would include a precision and scale value. The maximum scale value is 30, but the scale can't be more than the precision value. The following values appear to represent the maximum values for precision and scale:

```
CREATE TABLE floating
( var1  FLOAT(23,23)
, var2  DOUBLE(53,30));
```

Testing on the release found odd results when using a FLOAT data type and a precision above 23. If you opt for these approximate numeric values, please test thoroughly. Sometimes the choice of a floating-point number can be replaced with a DECIMAL data type because you can set more decimal places than you need with the scale parameter and the values are accurate rather than approximate. Choose the DOUBLE when you have a very precise decimal value.

Integers in the MySQL database are signed or unsigned. This is more common in C and C++ programming languages than it is in C# or Java programming. The idea of signed integers is that half the values are negative numbers, half the values minus one are positive numbers, and the last number is a zero. Unsigned integers start with zero and give you twice the number of positive numbers plus one. Table 6-2 qualifies the possible ranges of integer data types.

The TINYINT has two aliases, the BOOL and BOOLEAN data types. When you define a BOOLEAN you get a zero for false and one for true. INT has an INTEGER alias, and BIGINT has a SERIAL alias.

These data types have the same pattern of assignment when you create tables. You provide the integer type and an UNSIGNED keyword when you want zero to be the maximum positive number. A negative to positive number is the default when you only specify the data type of integer, which makes it an unsigned integer.

You would create a table with signed and unsigned integers like this:

```
mysql> CREATE TABLE integer_sample
    -> ( positive_spectrum  INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
    -> , full_spectrum      INT NOT NULL
    -> , fill_full_spectrum INT ZEROFILL
    -> , fill_pos_spectrum  INT UNSIGNED ZEROFILL);
```

Two of the columns use an unsigned integer and the positive_spectrum column uses an in-line PRIMARY KEY constraint. The primary key column designates itself as a surrogate key

Type	Subtype	Storage (Bytes)	Minimum Value	Maximum Value
TINYINT	Signed	1	–128	127
TINYINT	Unsigned	1	0	255
SMALLINT	Signed	2	–32768	32,767
SMALLINT	Unsigned	2	0	65,535
MEDIUMINT	Signed	3	–8,388,608	8,388,607
MEDIUMINT	Unsigned	3	0	16,777,215
INT	Signed	4	–2,147,483,648	2,147,483,647
INT	Unsigned	4	0	4,294,967,295
BIGINT	Signed	8	-9.2×10^{-18}	9.2×10^{18}
BIGINT	Unsigned	8	0	1.8×10^{19}

TABLE 6-2. *MySQL Integer Table*

column by invoking automatic numbering with the `AUTO_INCREMENT` keyword. The other two columns use a signed integer. The last two columns include an optional `ZEROFILL` keyword. Columns with `ZEROFILL` display leading zeros for integers.

Boolean Data Types There is no Boolean data type in MySQL. Some developers are surprised by the lack of a Boolean data type because MySQL Workbench provides one. Actually, MySQL Workbench uses Boolean as an alias for the `TINYINT` data type. It's more convenient in MySQL to implement this because two constants are defined in the MySQL Monitor to handle Boolean operations: `TRUE` and `FALSE`. You can query these constants like this:

```
SELECT true, false;
```

Here's the return from MySQL Monitor:

```
+------+-------+
| TRUE | FALSE |
+------+-------+
|    1 |     0 |
+------+-------+
```

As mentioned, the range of a `TINYINT` is 256 values. The range can be either –128 to 127, or zero to 255. The constants don't change the range of the data type.

Date Data Types Like an Oracle database, dates and timestamps are both date data types. They are typically scalar double-precision numbers linked to an epoch. An *epoch* is a slice of time typically measured in decades. Integer values are mapped to days in the epoch, and the fractions of days, such as hours, minutes, and seconds, are expressed in decimal equivalents.

Table 6-3 shows the available date, date-time, and timestamp data types in MySQL. You should verify what type of data you may need to store for the business rule before choosing one of these data types.

There's only one data type for a date, and it's `DATE`. The `DATE` data type takes a date in a four-digit year, hyphen, two-digit month, hyphen, and two-digit day. The format would look like this, `yyyy-mm-dd`, or like this, `yyyymmdd`, without the optional hyphens. You can also use a two-digit year, which presents challenges unless you know how and why it works. That format is `yy-mm-dd` or `yymmdd`, and it assumes two digit years belong to the last forty or next sixty years. The pattern for inserting values into a data type is known as a *format mask*. This format mask is more like the one in Perl than Oracle's default `DATE` format masks.

Data Type	Zero Fill Format Mask
DATE	'0000-00-00'
DATETIME	'0000-00-00 00:00:00'
TIMESTAMP	'0000-00-00 00:00:00'
TIME	'00:00:00'
YEAR	'0000'

TABLE 6-3. *MySQL Date and Time Data Types*

The range of dates in the epoch is 1000-01-01 to 9999-12-31. It's unlikely you'll need to worry about a business date falling outside the range of possibilities.

NOTE
The epoch for dates is much larger than the epoch for timestamps in MySQL.

You can assign a date-time or timestamp value to a `DATE` column, but the assignment raises a warning message advising you that the assignment loses precision. This warning occurs most frequently when developers assign the date-time result from the `NOW()` function. This is like assigning a value from a `DOUBLE` number to a `BIGINT`, where the decimal portion of the original number is truncated (dropped).

MySQL's `NOW()` function is very much like the `SYSDATE` function in Oracle. They both return the current date-time value. Oracle's `DATE` data type doesn't raise a warning because it's not a `DATE` but a timestamp. Truncated time-stamps are dates in Oracle, but their storage is as a date-time value.

The following `date_sample` table demonstrates defining a column with a `DATE` data type:

```
CREATE TABLE sample_date
( var1  DATE
, var2  DATE DEFAULT '2011-01-01');
```

The default day isn't very helpful, because ideally you want the current day as a default. At present, MySQL doesn't support a call to the `UTC_DATE()` function as a default value. Default values must be numeric or string literals (ouch).

You cannot assign a current date value, but you can change the `DATE` data type to a `TIMESTAMP` data type and assign the current `TIMESTAMP` value. You can designate only one `TIMESTAMP` column in any table. A `TIMESTAMP` column uses a `TIMESTAMP` data type and has a `DEFAULT` or `ON UPDATE CURRENT_TIMESTAMP` event linked to it. That means multiple columns in a table can use a `TIMESTAMP` data type, provided there is no `TIMESTAMP` column. You can define other date-time columns using the `DATETIME` data type.

The syntax for an `updated` `TIMESTAMP` column that's always current with the last `INSERT` or `UPDATE` to the row would look like this:

```
CREATE TABLE sample
( sample_id  INT UNSIGNED AUTO_INCREMENT PRIMARY KEY
, created    DATETIME
, updated    TIMESTAMP NOT NULL
                       DEFAULT CURRENT_TIMESTAMP
                       ON UPDATE CURRENT_TIMESTAMP);
```

The `updated` column in the `sample` table gets the `CURRENT_TIMESTAMP` with an `INSERT` statement and gets an updated `CURRENT_TIMESTAMP` with any `UPDATE` statement. If you attempted to define a table with two columns that use a `TIMESTAMP` data type, and one is a `TIMESTAMP` column, you would raise the following error (reformatted to fit the page):

```
ERROR 1293 (HY000): Incorrect table definition; there can be only
one TIMESTAMP column with CURRENT_TIMESTAMP in DEFAULT or
ON UPDATE clause
```

When You Want It All

Sometimes for a *who-audit* component, you need to know when a row was *created* and *last updated*, and by whom. That is more difficult in MySQL because of the single `TIMESTAMP` column limit. The workaround requires you to implement an *on insert* trigger that assigns the `CURRENT_TIME` to `created` column, while leaving the `updated` column as the single `TIMESTAMP` column in the table.

Assuming the `sample` table definition from the "Date Data Types" section earlier in the chapter, you can write an *on insert* trigger to populate the `created` column with the current timestamp value. The trigger runs only on insertion of the row, while the event handler runs with each `UPDATE` statement. The *on insert* trigger (based on material in Chapters 14 and 15) would look like this in MySQL:

```
-- Replace value from the insert statement.
CREATE TRIGGER sample_t
BEFORE INSERT ON sample
FOR EACH ROW
BEGIN
  SET new.created = current_timestamp();
END;
$$
```

Only one statement is processed in this trigger. It resets the `created` element of the *new* pseudo structure (tied to the table definition in the data catalog). This type of table would let you insert all null values, like this:

```
INSERT INTO sample VALUES (null, null, null);
```

A query against the table would show this:

```
+-----------+---------------------+---------------------+
| sample_id | created             | updated             |
+-----------+---------------------+---------------------+
|         1 | 2011-08-10 12:49:56 | 2011-08-10 12:49:56 |
+-----------+---------------------+---------------------+
```

Contrary to popular opinion, it appears you *can* have it all—well, at least in this one little use case. You implement two set statements with the `CURRENT_DATE` function when the `created` and `updated` *who-audit* columns are `DATE` data types.

Although not supported by MySQL, the Oracle database does support an equivalent call to the `SYSDATE` function, which is a current date-time value. Oracle also supports any of the built-in functions as default values for tables.

Date-times, timestamps, and time data types contain a date and the decimal value that maps to a time. The format for date-times is `yyyy-mm-dd hh:mm:ss`, and the data type is `DATETIME`. In this case, hyphens and colons aren't optional in the format mask for `DATETIME` columns. Although this is a more restrictive format mask than the `DATE` mask, at least there's only one way to do it.

The range of date-times in the epoch is 1000-01-01 00:00:00 to 9999-12-31 00:00:00. This mimics the range of DATE columns, and in many cases the DATE and DATETIME values meet general business requirements.

The TIMESTAMP data type has the same format mask as DATETIME but it works in a completely different epoch. The epoch for a TIMESTAMP is January 1, 1970 to December 31, 2037. That's the end of time for many current computing epochs. The migration effort when the end of epoch approaches 2037 might rival the "Year 2000" conversion from two- to four-year dates.

TIP
Use the DATETIME*, because that way the expiration of the epoch won't cause you a headache, and you can write your own* ON INSERT *or an* ON UPDATE *trigger to capture the current date-time value. Chapter 15 contains examples of triggers.*

TIME is another data type that presents interesting challenges. It measures the interval between 838 hours, 59 minutes, and 59 seconds ago to 838 hours, 59 minutes, and 59 seconds in the future. The format mask for a TIME column is hh:mm:ss or hhmmss, which is similar to the time formatting in the DATETIME and TIMESTAMP data types. The good news here is that the colons are optional, and, better yet, you can insert this as a number or as string when you don't include the colons.

NOTE
The TIMESTAMP *data type is often chosen because it implements an* ON UPDATE *current timestamp trigger.*

Here's a sample table that uses these data types:

```
CREATE TABLE timing
( var1  DATETIME
, var2  TIMESTAMP
, var3  TIME
, var4  YEAR);
```

NOTE
As long as you remember the formatting, masks, date-times, timestamps, and times are fairly straightforward. The best way to keep things simple is to use the DATETIME *not the* TIMESTAMP *data type.*

The YEAR data type is fairly simple. You can define a YEAR column as a two-digit or four-digit year, and when you don't specify which you want, the default is a four-digit year.

Here's an example defining YEAR columns:

```
CREATE TABLE timing
( var1  YEAR
, var2  YEAR(2)
, var3  YEAR(4));
```

If you described the table, you would see the following:

```
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| var1  | year(4) | YES  |     | NULL    |       |
| var2  | year(2) | YES  |     | NULL    |       |
| var3  | year(4) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
```

Before you deploy these data types, make sure you understand the dates, date-times, times, and years data types. Choose an epoch that is consistent and best for your company, and remember that the `TIMESTAMP` epoch is short-lived and, where possible, use the `DATETIME` data type.

Strings Data Types Strings are straightforward because they're one of two things: a bunch of characters grouped together in some meaningful way, or a stream of binary characters that make a picture or document of some sort. The MySQL database stores character and binary strings in a similar way but in different data types.

Character strings are stored in fixed-width or variable-width data types. The `CHAR` data type stores fixed-width data, and the `VARCHAR` data type stores variable-width data types. There's also a `CHARACTER VARYING` alias that maps to the `VARCHAR` data type, but you should avoid it as legacy syntax. A `VARBINARY` data type stores binary strings. `CHAR` can hold only 255 characters of data, and `VARCHAR` and `VARBINARY` can have up to 65,535 characters of data.

Beyond 65,535 characters, you have large character and binary strings that are stored as special data types. They can hold large character strings or binary data streams. The families of these data types also start small, which can be misleading. They are the large string data types—one for characters and the other for binary characters. Character data types are `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, and `LONGTEXT`. Binary character data types are `TINYBLOB`, `BLOB`, `MEDIUMBLOB`, and `LONGBLOB`.

String data types also support optional settings for the character set and collation. Character sets are important because they designate the available symbols, which may be western European alphabets, Arabic Abjad (a script for Arabic, Persian, and Urdu), or pictograms such as Chinese and Japanese. Collation sets how a language is organized for sorting purposes, such as alphabetizing in English. Unless you override the settings chosen at installation, MySQL uses the defaults for the instance. Character sets and collation are typically matched, but you can override this paring. You can find available character sets by using the following MySQL Monitor command:

```
SHOW CHARACTER SET LIKE 'latin%';
```

This should return the following:

```
+---------+-----------------------------+-------------------+--------+
| Charset | Description                 | Default collation | Maxlen |
+---------+-----------------------------+-------------------+--------+
| latin1  | cp1252 West European        | latin1_swedish_ci |      1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1 |
| latin5  | ISO 8859-9 Turkish          | latin5_turkish_ci |      1 |
| latin7  | ISO 8859-13 Baltic          | latin7_general_ci |      1 |
+---------+-----------------------------+-------------------+--------+
```

National character and variable character types let you use broader character sets, such as Unicode.

The CHAR, CHARACTER, NCHAR, and NATIONAL CHARACTER data types support a *fixed-width* character string of an empty string to a 255 character string. Any of these types allocate space for the entire width of characters, whether they're required or not. The downside to this data type is that you consume space for empty space.

You can store binary, ASCII, or Unicode text in these data types. Optional settings let you override the default character set and collation.

The basic syntax to create columns of these data types is shown here:

```
CREATE TABLE fixed_width
( var1  CHARACTER(10)
, var2  CHAR(10)      BINARY
, var3  CHAR(10)      ASCII
, var4  CHAR(10)      UNICODE
, var5  CHAR(10)      CHARACTER SET utf16 COLLATE utf16_general_ci
, var6  NCHAR(10)
, var7  NATIONAL CHARACTER(10));
```

The BINARY, ASCII, and UNICODE options exist independent of the character set and collate settings. You can't use them on the same column. Also, these don't work with a national character data type, like NCHAR or NATIONAL CHARACTER.

In addition to fixed-width strings, *variable-width* strings provide a maximum length for strings in a column, and they consume space only for the values inserted. They're the preferred solution for most strings, with a maximum length of 65,535 bytes.

Like its fixed-width peers, variable-width strings have multiple keywords that can be used to define these types as columns in a table. VARCHAR and CHARACTER VARYING store strings in the default character set for the database and use the pared default collation. NVARCHAR and NATIONAL VARCHAR data types are available for Unicode strings. You also have a binary option for the VARCHAR and CHARACTER VARYING data types, but you should generally opt for the VARBINARY data type for binary strings.

The following example shows how to define columns of these data types:

```
CREATE TABLE variable_width
( var1  VARCHAR(10)
, var2  VARCHAR(10)            BINARY
, var3  CHARACTER VARYING(10)
, var4  CHARACTER VARYING(10)  BINARY
, var5  NVARCHAR(10)
, var6  NATIONAL VARCHAR(10)
, var7  VARBINARY(10));
```

The VARBINARY does support your setting an override character set and collation. The syntax for a VARBINARY follows the examples provided in the discussion of fixed-width strings for character set and collation.

As a rule, most developers use VARCHAR or NVARCHAR data types. You should consider revising older scripts that use the other types to the more common data types.

Large strings should mean large strings, but TINYTEXT and TINYBLOB have a maximum size of 255 characters. As you can imagine, they're rarely used. TEXT and BLOB data types are equivalent to the VARCHAR and VARBINARY data types with a maximum value of 65,535 bytes. MEDIUMTEXT and MEDIUMBLOB represent what are generally large strings, up to 16,777,215 bytes. LONGTEXT and LONGBLOB data types store up to 4 GB. When you use these types in the database,

MySQL Check Constraint Syntax Mystery

At the time of writing, MySQL Server's documentation lists CHECK constraints as valid syntax but doesn't let you use them. You can find the discussion about this valid-but-invalid syntax in MySQL Bug #3464, which was logged in April 2004 and last updated April 2010.

they need to be loaded by segments. As a rule of thumb, multiples of the operating system block size (typically 8 KB) are best, such as 8,192 bytes, 32,768 bytes, and so forth.

These data types don't specify size because they're implemented as variable-length character or binary byte streams (a more formal and complex word for strings). As a result, this example defines only two of the types:

```
CREATE TABLE large_strings
( var1  BLOB
, var2  TEXT);
```

The nice thing about BLOB and TEXT data types is that a query with a * for all columns won't raise a display error as it would in an Oracle database. That makes things simpler in one way and more complex in another. All of the strings will roll across your console when you query either of these large stream data types.

ENUM and SET Data Type Columns

The MySQL database supports two complex data types: ENUM and SET. The ENUM stands for enumeration of possible values and SET represents a set of inclusive values. You could make the argument that they're both like nested tables, but they are, in fact, different. You can't use an INSERT or UPDATE statement to change the list of values in either the ENUM or SET data type columns. You can use only the ALTER TABLE statement (see Chapter 7 for that statement syntax) to change the list of values. There's also an occasional opinion that the ENUM and SET data types act like specialized CHECK constraints, because they limit the values in a column to a list of possible values.

The ENUM and SET data types maintain a list of possible values in an array that is densely populated. Densely populated arrays use a sequential numeric index to identify items in the list. MySQL's index for the ENUM and SET data types is a zero-based integer. Both the ENUM and SET data types support overriding the character set and collation for the column. This behavior is consistent with other strings in MySQL and beneficial when supporting a multiple language installation.

The difference between the two data types is that an ENUM data type allows you to pick only one value from the list for each column value. The SET data type lets you pick one to many of the values. The ENUM data type may contain up to 65,535 values, whereas the SET data type can contain only 64 values.

They are defined by a similar rule, a comma-delimited list. The next two subsections provide examples of the ENUM and SET data types.

ENUM Data Type The ENUM type contains a list of values. The column in any row can reference only one of the values from the list.

The CREATE TABLE statement shows how you would define ENUM columns:

```
CREATE TABLE indexed_list
( var1  ENUM('Female','Male')
, var2  ENUM('Female','Male') DEFAULT 'Male'
, var3  ENUM('Female','Male') NOT NULL);
```

These three definitions have very different behaviors. var1 can hold an index value that points to a case-insensitive match to the strings Female or Male, or a null value. It doesn't matter how they're inserted or updated, because MySQL performs a case-insensitive match and then stores the index value from the match found. var2 can hold the same three values, but the DEFAULT value is inserted when you exclude the column from an override INSERT statement (see Chapter 8 for details). The only way to insert a null value in var1 or var2 is to explicitly insert or update the column with a null value. var3 must hold only an index value to a member of the enumeration list. That's because it uses an in-line constraint to prevent a null insertion or update.

NOTE
The worst default case I've ever seen of the misuse of override statements was in a hospital system that assumed a child was available for adoption unless the clerk entered an override character. Needless to say, a lot of parents were upset.

SET Data Type As discussed, the SET type contains a list of 64 values. The column in any row can reference only zero, one, or all of the values from the list. It can reference zero SET members only when the column is *nullable*. A NOT NULL constrained column must contain at least one value from the list.

You define a column with the SET data type the same way. Only the insert or update syntax differs, because you can enter multiple values. For those who browse the syntax, here's a quick example that should catch your eye:

```
CREATE TABLE inclusive_set
( var1  SET('Ham','Pepperoni','Pineapple','Salami','Sausage'));
```

As mentioned, any insert or update to the column performs a case-insensitive match of the inserted or updated value against the list, and then stores the index values that match in the SET column. This means that queries find the index and return the case-sensitive value from the SET data type.

Column and Table Constraints

Column and table constraints work in MySQL databases more or less the same way they do in an Oracle database. That's because the syntax follows the ANSI SQL standards. You have in-line and out-of-line constraints in the CREATE TABLE statement, and the out-of-line constraints can be column- or table-level constraints.

The biggest difference between the two is that MySQL doesn't allow you to enter constraint names for in-line PRIMARY KEY or NOT NULL constraints. You can name a PRIMARY KEY constraint when you define it as an out-of-line constraint. NOT NULL constraints can't be given a user-defined name because they're treated as column properties inside the table.

MySQL doesn't support a CHECK constraint as the Oracle database does. This means you must use ON INSERT or ON UPDATE database triggers to gain some of the same behaviors. The ENUM and SET data types do constrain the list of possible values, and thereby provide a limited alternative to an Oracle CHECK constraint. It appears that the MySQL team plans to implement a CHECK constraint because it's defined in the documentation.

The following subsections cover column and table constraints. Out-of-line column constraints are included in the table constraints subsection because they're defined after the list of columns in a table.

Column Constraints Column-level constraints apply only to a single column. Single column primary keys are typically *surrogate* keys (or artificial sequences used to simplify joins and externally identify rows). Foreign keys are single columns when they refer to surrogate primary keys. Unfortunately, you can't define an in-line FOREIGN KEY constraint in MySQL. FOREIGN KEY constraints are InnoDB engine features, not generic features of MySQL. NOT NULL constraints are always column-level constraints, because they make values mandatory in columns.

PRIMARY KEY constraints for surrogate keys also constrain a column to contain unique values in the scope of all rows in a table. Because surrogate keys are supported by sequences, it's important to note that there's a difference between how Oracle and MySQL databases implement sequences. Unlike Oracle, MySQL doesn't support independent sequence structures. MySQL implements sequences as table properties, just as it treats NOT NULL constraints.

The following example shows how to implement in-line PRIMARY KEY and NOT NULL constraints:

```
CREATE TABLE primary_inline
( primary_inline_id  INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
, optional           VARCHAR(10)
, mandatory          VARCHAR(10)  NOT NULL);
```

The important aspect of an unsigned integer is that the maximum value can be twice that of a signed integer. Surrogate primary keys should start with 1 and have the potential to grow as large as necessary, which means their columns should always be unsigned integers or doubles. The rule of thumb creates them as integers until they approach the maximum value, and then you should change them to a double data type.

The PRIMARY KEY clause follows the data type. It imposes two constraints on the column: a NOT NULL and a UNIQUE constraint. The next AUTO_INCREMENT keyword creates a sequence as a table property that starts at 1 and increments by 1. The optional column is unconstrained and the mandatory column is NOT NULL constrained.

You can't define an in-line FOREIGN KEY constraint. They must be defined as table-level constraints even when they refer only to a single column.

UNIQUE constraints that refer to a single column can be defined as in-line constraints. The following demonstrates an in-line UNIQUE constraint:

```
CREATE TABLE unique_table
( unqiue_table_id  INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
, unique_column    VARCHAR(10) UNIQUE);
```

MySQL supports only PRIMARY KEY, NOT NULL, and UNIQUE constraints in-line. As a last word on in-line constraints, remember that FOREIGN KEY constraints are available only when you use the InnoDB engine, and they can't be defined in-line.

Table Constraints You define table constraints after the last column in a CREATE TABLE statement. MySQL lets you define table-level PRIMARY KEY, FOREIGN KEY, and UNIQUE constraints.

The first example shows how to define out-of-line PRIMARY KEY and UNIQUE constraints:

```
CREATE TABLE outofline
( outofline_id  INT UNSIGNED
, subject_item  VARCHAR(10)
, CONSTRAINT pk_outofline PRIMARY KEY (outofline_id)
, CONSTRAINT un_outofline UNIQUE (subject_item));
```

The examples are out-of-line constraints that work on a single column. Any real table-level constraints would contain a comma-delimited list of column names.

An out-of-line FOREIGN KEY constraint is defined like this:

```
CREATE TABLE dependent
( dependent_id  INT UNSIGNED
, outofline_id  INT UNSIGNED
, CONSTRAINT fk_dependent
  FOREIGN KEY (outofline_id) REFERENCES outofline(outofline_id));
```

Table-level FOREIGN KEY constraints are slightly more complex than the PRIMARY KEY and UNIQUE constraints because they have two comma-delimited lists of column names. One column list occurs in the parentheses after the FOREIGN KEY phrase, and the other list of columns occurs after the table name in the REFERENCES clause. The parentheses after the FOREIGN KEY phrase contains columns defined in the current table. The parentheses after the REFERENCES phrase contain columns defined in the current table when the current table's name precedes the parentheses; otherwise, it stores a list of columns found in the table that precedes the list of column names. As mentioned, MySQL FOREIGN KEY constraints work only when both the primary and foreign key holding tables are defined with the InnoDB engine.

Partitioned Tables

Like the Oracle database, MySQL supports partitioning of tables. It supports range, list, hash, and key partitioning. Range partitioning lets you partition based on column values that fall within given ranges. List partitioning lets you partition based on columns matching one of a set of discrete values. Hash partitioning lets you partition based on the return value from a user-defined expression (the result from a stored SQL/PSM function). Key partitioning performs like hash partitioning, but it lets a user select one or more columns from the set of columns in a table; a hash manages the selection process for you. A hash is a method of organizing keys to types of data, and hashes speed access to read and change data in tables.

Each of the following subsections discusses one of the supported forms of partitioning in MySQL. Naturally, there are differences between Oracle and MySQL implementations.

Range Partitioning Range partitioning works only with an integer value or an expression that resolves to an integer against the primary key column. The limitation of the integer drives the necessity of choosing an integer column for range partitioning. You can't define a range-partitioned table with a `PRIMARY KEY` constraint unless the primary key becomes your partitioning key, like the one below.

```
CREATE TABLE ordering
( ordering_id      INT UNSIGNED AUTO_INCREMENT
, item_id          INT UNSIGNED
, rental_amount    DECIMAL(15,2)
, rental_date      DATE
, index idx (ordering_id))
PARTITION BY RANGE(item_id)
( PARTITION jan2011 VALUES LESS THAN (10000)
, PARTITION feb2011 VALUES LESS THAN (20000)
, PARTITION mar2011 VALUES LESS THAN (30000));
```

Range partitioning is best suited to large tables that you want to break into smaller pieces based on the integer column. You can also use stored functions that return integers as the partitioning key instead of the numeric literals shown. Few other options are available in MySQL.

List Partitioning Like an Oracle list partition, a MySQL list partition works by identifying a column that contains an integer value, the `franchise_number` in the following example. Partitioning clauses follow the list of columns and constraints and require a partitioning key to be in the primary key or indexed.

The following list partition works with literal numeric values. MySQL uses the `IN` keyword for list partitions while Oracle doesn't. Note that there's no primary key designated and an index is on the auto-incrementing surrogate key column. A complete example is provided to avoid confusion on how to index the partitioning key:

```
CREATE TABLE franchise
( franchise_id     INT UNSIGNED AUTO_INCREMENT
, franchise_number INT UNSIGNED
, franchise_name   VARCHAR(20)
, city             VARCHAR(20)
, state            VARCHAR(20)
, index idx (franchise_id))
PARTITION BY LIST(franchise_number)
( PARTITION offshore VALUES IN (49,50)
, PARTITION west VALUES IN (34,45,48)
, PARTITION desert VALUES IN (46,47)
, PARTITION rockies VALUES IN (38,41,42,44));
```

The inclusion of a `PRIMARY KEY` constraint on the `franchise_id` column would trigger an ERROR 1503 when the partitioning key isn't the primary key. The reason for the error message is that a primary key implicitly creates a unique index, and that index would conflict with the partitioning by list instruction. The use of a non-unique `idx` index on the `franchise_id` column is required when you want to partition on a non-primary key column.

Columns Partitioning Columns partitioning is a *new* variant of range and list partitioning. It is included in MySQL 5.5 and forward. Both range and list partitioning work on an integer-based column (using `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT` [alias `INTEGER`], and `BIGINT`). Columns partitioning extends those models by expanding the possible data types for the partitioning column to include CHAR, VARCHAR, BINARY, and VARBINARY string data types, and DATE, DATETIME, or TIMESTAMP data types. You still can't use other number data types such as DECIMAL and FLOAT. The TIMESTAMP data type is also available only in range partitions with the caveat that you use a UNIX_TIMESTAMP function, according to MySQL Bug 42849.

Hash Partitioning Hash partitions ensure an even distribution of rows across a predetermined number of partitions. It is probably the easiest way to partition a table quickly to test the result of partitioning on a large table. You should base hash partitions on a surrogate or natural primary key.

The following provides a modified example of the ordering table:

```
CREATE TABLE ordering
( ordering_id      INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
, item_id          INT UNSIGNED
, rental_amount    DECIMAL(15,2)
, rental_date      DATE)
PARTITION BY HASH(ordering_id) PARTITIONS 8;
```

This is the partitioning type that benefits from a PRIMARY KEY constraint because it automatically creates a unique index that can be used by the hash. A non-unique index such as the list partitioning example doesn't work for a hash partition.

Key Partitioning Key partitioning is valuable because you can partition on columns that aren't integers. It performs along the line of hash partitioning, except the MySQL Server uses its own hashing expression.

```
CREATE TABLE orders_list
( order_list_id     INT UNSIGNED AUTO_INCREMENT
, customer_surname  VARCHAR(30)
, store_id          INT UNSIGNED
, salesperson_id    INT UNSIGNED
, order_date        DATE
, index idx (order_list_id))
PARTITION BY KEY (order_date) PARTITIONS 8;
```

This is the only alternative when you want to partition by date ranges. Like the hash partition, it's easy to deploy. The only consideration is the number of slices that you want to make of the data in the table.

Subpartitioning The concept of subpartitioning is also known as composite partitioning. You can subpartition range or list partitions with a hash, linear hash, or linear key.

A slight change to the previously created ordering table is required to demonstrate composite partitioning: we'll add a store_id column to the table definition. The following is an example of a range partition subpartitioned by a hash:

```
CREATE TABLE ordering
( ordering_id      INT UNSIGNED AUTO_INCREMENT
, item_id          INT UNSIGNED
, store_id         INT UNSIGNED
, rental_amount    DECIMAL(15,2)
, rental_date      DATE
, index idx (ordering_id))
PARTITION BY RANGE(item_id)
  SUBPARTITION BY HASH(store_id) SUBPARTITIONS 4
( PARTITION jan2011 VALUES LESS THAN (10000)
, PARTITION feb2011 VALUES LESS THAN (20000)
, PARTITION mar2011 VALUES LESS THAN (30000));
```

Composite partitioning is non-trivial and might require some experimentation to achieve optimal results. Plan on making a few tests of different scenarios before you deploy a solution.

Indexes

Indexes apply against tables. You can index ordinary data, such as scalar columns. Alternatively, you can use a full text index to search a large text column or a spatial index to search specialized spatial columns. These specialized columns support graphics and geometry. Full text indexes use custom parsers that are specific to languages and provide such things as lexical stemming. Both Oracle and MySQL support spatial objects.

Indexes typically resolve by using a B-Tree access path to data, which makes searching data faster. On the downside, indexes need to be refreshed as you insert new data and update existing data. You also have several options for indexes, such as hashes, hash clusters, bitmaps, key-compressed, and function-based indexes.

Unique indexes can work with one or several columns. Naturally, they work with unique data sets. Unique data sets have the highest cardinality possible. Unique indexes work best with multiple columns, because the combination of values increases the effectiveness of B-Tree and bitmap indexes. This is done by reducing the number of levels from the top of the B-Tree to the data, which means the index finds rows quickly. Although it is possible to craft many unique indexes on tables with many columns, you should remember that every unique index places a load on the server for all `INSERT`, `UPDATE`, and `DELETE` statements.

It is also possible to define a non-unique index. These are useful when they have a high cardinality, which means there's a lot of diversity between rows of data. A rule of thumb on non-unique indexes is that the distribution of data should ensure that no single set of column values exceeds 5 percent of the total data set. Following this rule, you shouldn't implement a non-unique index on a gender column that has two possibilities, for example. Likewise, a pair of columns on gender and college class standing wouldn't be a good choice for a non-unique index, because the distribution would approach 12.5 percent. A distribution of 12.5 percent would be a low cardinality value because of the similarity of the data. A non-unique index against low cardinality data would present more overhead than benefit.

Oracle Indexes

You can create a unique or non-unique index in an Oracle database. These are separate structures from the table, and you create them independently of the table in most cases. The exceptions to that creation rule occur when you define a `PRIMARY KEY` or `UNIQUE` constraint on a table, because those constraints implicitly add unique indexes. Unique indexes created by `PRIMARY KEY` and `UNIQUE` constraints can't be dropped from a database unless you alter the table and remove the constraints. You can alter many things about indexes as well as rebuild them with the `ALTER` statement, which is covered in Chapter 8. For reference, all table indexes are dropped from the database when you drop a table.

Indexes work best when they include multiple columns, but you can define an index to work on a single column. An example of a multiple column index that's critical in good design is an index that starts with the surrogate key column and includes all columns in the natural key. The reason for such an index is twofold: The natural key component contains the values that let you search for data based on business aspects of the model. The leading surrogate key lets joins resolve faster with the multiple column index.

Full text indexes work against CLOB data type columns. They let you perform searches using approximate matching and proximity matching. These types of indexes improve performance for advanced text processing such as *stemming*, where you match the root-word stem against actual occurrences of the word in use.

Spatial indexes work with Oracle object types that are specialized PL/SQL program units stored inside the database, such as stored procedures. When you define columns that use these user-defined types, the columns hold a constructor with arguments. These types of objects are known as *collapsed* objects, and you bring them to life with the TREAT function.

You create a unique index with syntax like this:

```
CREATE UNIQUE INDEX index_name
ON table_name (column_name1, column_name2, ...);
```

Oracle indexes use B-Tree by default and also support storage clauses, which are very important on large tables. Like table data, index data should be stored in as much contiguous space as possible. You organize data into contiguous space by ensuring the proper sizing of tablespaces and extents allocated to tables and indexes.

The alternative to a unique index is a *bitmapped* index. Bitmapped indexes work with data that has a low cardinality—in other words, a non-unique set of values, such as a gender column. Some restrictions apply to bitmap indexes. They can't coexist with unique indexes or domain indexes. Likewise, you can't specify bitmap indexes with a global partition index, and you can't create a secondary bitmap index on an index-organized table unless the index-organized table has a mapping table associated with it.

You create a bitmap index with syntax like this:

```
CREATE BITMAP INDEX index_name
ON table_name (column_name1, column_name2, ...);
```

A non-unique index has virtually the same syntax possibilities. All you do is drop the UNIQUE keyword, like so:

```
CREATE INDEX index_name
ON table_name (column_name1, column_name2, ...);
```

Remember that UNIQUE indexes are almost always important, but too many on a single table can slow transaction throughput. Non-unique indexes have less value but present the same demands for updates as UNIQUE indexes. Use non-unique indexes sparingly.

MySQL Indexes

MySQL supports unique and non-unique indexes against single or multiple columns. It also supports full-text searches against large strings (stored in TEXT columns), and spatial indexes against the POINT and GEOMETRY data types. These specialized types and more are supported by the Spatial Extension provided in four engines: MyISAM, InnoDB, NDB, and Archive.

Like Oracle indexes, MySQL indexes work best when they're unique or non-unique against high-cardinality data. They work poorly when placed against low-cardinality data.

You define a unique index with the following syntax:

```
CREATE UNIQUE INDEX index_name
ON table_name(column_name1, column_name2, ...) USING BTREE;
```

In lieu of the `UNIQUE` keyword, you can use `FULLTEXT` and `SPATIAL`. The absence of a qualifier creates a non-unique index. The `USING` subclause is the default, and you can replace `BTREE` with a `HASH` index type. MySQL also supports clauses for storage for the block size of the key and a `WITH PARSER` clause for text-based indexes.

A non-unique index, also known as a *key*, uses this syntax:

```
CREATE INDEX index_name
ON table_name(column_name1, column_name2, ...) USING BTREE;
```

You can also create indexes inside the `CREATE TABLE` statement. The syntax for that type of unique index is shown here:

```
, UNIQUE INDEX (column_name)
```

A non-unique index can be created by using just the `INDEX` keyword or the `KEY` keyword, like so:

```
, KEY (column_name)
```

This concludes the index section, but you might want to review the subpartitioning information earlier in the chapter. There you'll find that a non-unique key is the critical component for creating partitioned tables on non-unique columns.

Summary

This chapter covered how you create data structures, such as databases, users, tables, and indexes. It presented the syntax necessary to create these structures in Oracle and MySQL, and discussed similarities and differences between the databases.

Not all things in Oracle map directly to MySQL, and vice versa. It's important where possible to implement concepts that are more portable than product-specific. Likewise, it's also important that you not overlook opportunities presented by product-specific features.

Mastery Check

The mastery check is a series of true or false and multiple choice questions that let you confirm how well you understand the material in the chapter. You may check the Appendix for answers to these questions.

1. **True** ☐ **False** ☐ A column can be made mandatory by using an in-line `NOT NULL` constraint in Oracle and MySQL databases.
2. **True** ☐ **False** ☐ A `CHECK` constraint can be implemented in a MySQL database.

3. **True** ☐ **False** ☐ A `FOREIGN KEY` constraint can be implemented against only one column.
4. **True** ☐ **False** ☐ You can create tables with nested tables in an Oracle database.
5. **True** ☐ **False** ☐ You can use an `UNSIGNED INT` data type when creating a table in an Oracle database.
6. **True** ☐ **False** ☐ Creating a unique index inside the `CREATE TABLE` statement is possible in MySQL.
7. **True** ☐ **False** ☐ A range partition uses the `VALUES IN` clause in an Oracle database.
8. **True** ☐ **False** ☐ An `ENUM` data type allows only one of a list of values to be stored in a column.
9. **True** ☐ **False** ☐ MySQL supports composite partitioning.
10. **True** ☐ **False** ☐ A foreign key works whether the data type is a signed or unsigned integer.
11. Which of the following data types isn't supported in a MySQL database?
    - **A.** `BLOB`
    - **B.** `MEDIUMCLOB`
    - **C.** `TEXT`
    - **D.** `DOUBLE`
    - **E.** `FLOAT`
12. Which of the following data types isn't supported in an Oracle database?
    - **A.** `CLOB`
    - **B.** `BLOB`
    - **C.** `NCHAR`
    - **D.** `VARYING CHARACTER`
    - **E.** `CHAR`
13. Which of the following aren't date-time data types (multiple possible answers)?
    - **A.** `DATE` in Oracle
    - **B.** `DATE` in MySQL
    - **C.** `TIMESTAMP` in either database
    - **D.** `TIME` in MySQL
    - **E.** `YEAR` in MySQL
14. Which of the following syntax operations are prohibited in MySQL?
    - **A.** Defining an in-line `NOT NULL` constraint
    - **B.** Defining an in-line `FOREIGN KEY` constraint
    - **C.** Defining an in-line `UNIQUE` constraint
    - **D.** Defining an in-line `PRIMARY KEY` constraint
    - **E.** Defining an out-of-line `FOREIGN KEY` constraint

15. Which data type is supported by list and range partitioning in MySQL?

A. `FLOAT`

B. `CHAR`

C. `VARCHAR`

D. `DATE`

E. `INT`