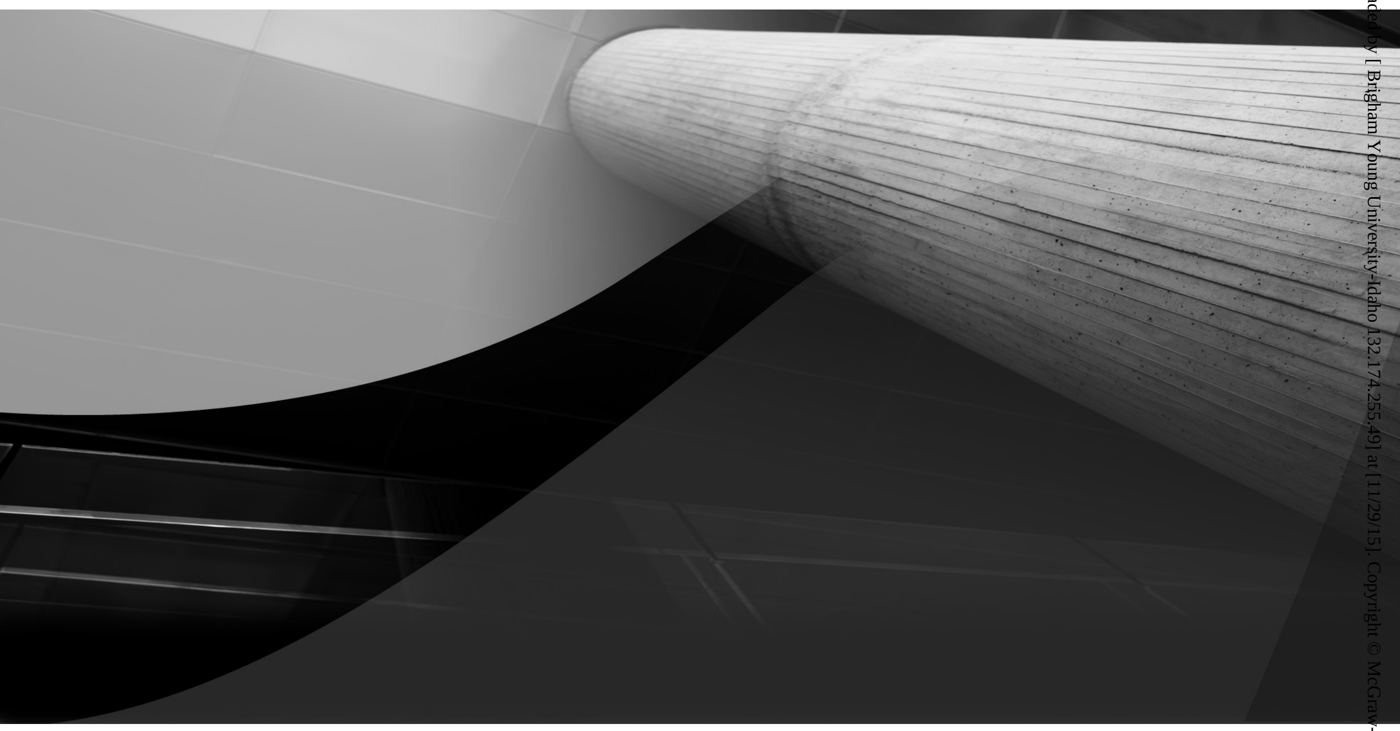

CHAPTER 7

Modifying Users and Structures

This chapter focuses on how you can change users, databases, tables, and indexes you've created in the database, and it explores how you can change your session to meet your needs in Oracle and MySQL. The chapter is organized by the following topics:

- Users
- Databases
- Sessions
- Tables
- Views
- Indexes

As explained in Chapter 6, users are synonymous with schemas in an Oracle database, and they're distinctly separate in a MySQL database. Databases are private work areas in both cases, and changes to them remain until you remove or change them again. Connections to your database management system are a session, and a session lasts only for the duration of your connection to a database or schema. Any changes to a session are lost when you break the connection by logging out or, in Oracle, by connecting as another user to another database. Tables are permanent structures unless you define them as temporary tables. Changes made to tables, like databases, last until you drop the table, undo the changes, or make new changes. As with tables, indexes exist as long as the table they reference exists, unless you drop or alter the table. Although users can modify index structures, content changes in the index occur only through changes in the referenced tables.

The following sections discuss how you can work with changing and removing users, databases, sessions, tables, and indexes. You'll learn what a developer needs to know to work with these structures in Oracle and MySQL. You won't find an exhaustive listing of all the things you can do with or to databases, because entire books are written on that, such as the *Oracle Database 11*g *DBA Handbook* (McGraw-Hill, 2008).

Users

The user and schema are inseparable in an Oracle DBMS. They share the same names, and the user generally holds all defined privileges on the schema. That means you must change a user to change a schema or database. Commands that change the database are actually changing accounts on the Oracle server. When you change a database in MySQL, you are changing only a private work area and not the database server.

The following sections cover the Oracle and MySQL commands used to change and remove users. Occasional cross-references exist in the text, so read the Oracle section before the MySQL section.

Oracle Users

Oracle users typically don't have privileges that let them change their user or schema properties unless they've been granted super user privileges. That means commands that change users or schema are run typically by the `SYSTEM` user. Dropping a user is a rare occasion in most Oracle databases, but changing properties of the user occurs routinely.

Oracle ALTER USER Statement

The prototype for the ALTER USER statement lets you change properties of a user, such as their role, profile, storage, password, and account status. The generic prototype for the ALTER USER statement is shown here:

```
ALTER USER user_name
[IDENTIFIED
{[BY current_password REPLACE new_password] |
 [EXTERNALLY AS 'certificate_name'] |
 [GLOBALLY AS 'directory_name']]}
[DEFAULT TABLESPACE tablespace_name]
[TEMPORARY TABLESPACE {tablespace_name | tablespace_group }]
[QUOTA {size_clause | UNLIMITED} ON tablespace_name]
[PROFILE profile_name]
[DEFAULT ROLE {role_name | ALL EXCEPT role_name | NONE}]
[PASSWORD EXPIRE]
[ACCOUNT {LOCK | UNLOCK}]
[{GRANT | REVOKE} CONNECT THROUGH
 {ENTERPRISE USERS |
  WITH {ROLE {role_name | ALL EXCEPT role_name} |
        NO ROLES} [AUTHENTICATION REQUIRED]}]
```

Using the ALTER USER statement, you can configure one of three different authentication types for a user: password, Secure Sockets Layer (SSL) certificate, and Lightweight Directory Access Protocol (LDAP) certificate. You also have the security options to expire a password (useful when terminating employees) and locking an account.

The other clauses let you change default or temporary tablespaces, or a quota, profile, role, or pass-through authentication, none of which occurs frequently. These changes are also seldom made by developers, because doing so is the DBA's responsibility.

If a user loses a password, you could change the user's password like this:

```
ALTER USER stanley IDENTIFIED stanley;
```

Then you'd need to let the user know the new password and hope somebody doesn't crack it before the user logs in to change it.

Better yet, you can go one more step by expiring the password after changing it. An expired password prompts the user for a new password when he or she logs in. You expire a password like this:

```
ALTER USER stanley PASSWORD EXPIRE;
```

The user would try to connect like this,

```
sqlplus stanley/stanley
```

and would then see the following messages and prompts at the command-line interface to SQL*Plus, where the user would enter a new password:

```
C:\data\oracle>sqlplus stanley/stanley
SQL*Plus: Release 11.1.0.7.0 - Production on Thu Jun 30 21:40:45 2011
Copyright (c) 1982, 2008, Oracle.  All rights reserved.
```

```
ERROR:
ORA-28001: the password has expired

Changing password for stanley
New password:
Retype new password:
Password changed
```

Assuming you've configured the database to use LDAP authentication and provided a wallet, you can also change a login to an LDAP validation, like this:

```
ALTER USER miles IDENTIFIED GLOBALLY AS 'CN=miles,O=apple,C=US'
```

A user can also change his/her password with the `ALTER USER` statement. The following syntax requires that the user know his/her current password:

```
ALTER USER stanley IDENTIFIED BY stanley REPLACE beatles1964;
```

The only problem with the preceding syntax is that it discloses the user's new password in plain text. Unfortunately, there's no way around that syntax limitation. More often than not, individual users remember their passwords and can change it like this:

```
SQL> password
Changing password for STANLEY
Old password:
New password:
Retype new password:
Password changed
```

The reason changes of tablespaces, roles, and profiles aren't done by developers is because they are managed by the super user accounts. These super user accounts are owned by DBAs who administer the physical resources of the database. Developers work closely with the system administrators to ensure that adequate disk space and processing resources are available for the database on any server.

A key DBA activity that you should know how to perform while working in your laptop or desktop development databases is locking and unlocking accounts, such as the `oe` sample or the legacy `scott` schemas. You would unlock the `oe` schema with this syntax:

```
ALTER USER oe ACCOUNT UNLOCK;
```

Alternatively, you can open an account and change the password with a single command like this:

```
ALTER USER scott ACCOUNT UNLOCK IDENTIFIED BY tiger;
```

After you're done with a sample schema like these, you can lock them away by typing this:

```
ALTER USER scott ACCOUNT LOCK;
```

You should never leave user schemas open (unlocked) when you're not actively using them. It's simply a best practice to lock unused schemas that you might need to reopen and likewise to drop obsolete schemas.

Oracle DROP Statement

DROP statements are Data Definition Language (DDL) commands that let you remove structures from the database. The DROP statement uses different prototypes depending on whether you're removing a user, table, object type, function, procedure, or sequence.

The DROP USER statement removes the user and his/her schema. Sometimes other schemas have dependencies on tables or other structures in a schema. You need to specify the CASCADE option when dependencies exist, which will cause the DROP statement to drop them in cascading fashion.

The prototype is small and easy:

```
DROP USER user_name [CASCADE]
```

You could drop the scott sample schema with this command:

```
DROP USER scott;
```

TIP
Try to drop the user without the CASCADE option first, and if it fails, figure out what the dependency is before you append CASCADE to the statement.

MySQL Users

Like Oracle users, MySQL users typically don't have permissions to change or drop other users unless they've been given administrator privileges on the server. This section contains subsections on changing and dropping a user.

MySQL SET PASSWORD Statement

There is no ALTER USER statement in MySQL. If you're asking how you or an administrator can change a user's password without an ALTER USER statement, there is a way. This section shows you the syntax.

A root super user can add a password or change a password with the following syntax:

```
SET PASSWORD FOR user_name = PASSWORD('password_string');
```

If the user name existed in the database with more than one hostname value, the syntax changes to this:

```
SET PASSWORD FOR 'user_name'@'host_string' = PASSWORD('new_password');
```

The host_string can have several possible values. It can be localhost for server-only connections, a '%' for anywhere, an IP address, or a DNS recorded machine name, domain, or subdomain address. You can find more on user connections in the "MySQL Databases" section of Chapter 6.

Individual users can also change their own password with the following variation on the SET PASSWORD command syntax:

```
mysql> SET PASSWORD = PASSWORD('new_password');
```

No other aspects of MySQL users are subject to change other than grants of privileges. That's because a MySQL user is distinct from a database, and a user is really only a name, encrypted password, and a host linked to a group of privileges. Some DBAs and developers update user information in the data catalog, which is fine when you're running the Community Edition. You should not alter the data dictionary without the guidance of Oracle Support for commercially licensed versions of MySQL.

MySQL DROP USER Statement

A `root` super user or other authorized administrative user can drop a user with the following syntax:

```
DROP USER user_name;
```

Users create tables, views, and routines in databases, so you can drop a user without any impact on the structures they've created. Although this approach differs from that of Oracle, you should find it easy to use and leverage.

Database

Changes to the database are typically the responsibility of the DBA. They involve allocation of space to support what resides in the database. Oracle uses the `ALTER DATABASE` statement to change the definition of the user's private work area. You can find more on setting these parameters in the *Oracle Database Administrator's Guide 11*g.

MySQL treats databases quite differently from Oracle, and although you can use the `ALTER DATABASE` statement in MySQL, its uses are relatively small—all you can do is change the collation or character set. Both of these settings belong to the DBA role.

Sessions

Changes to the session or connection are supported by the `ALTER SESSION` statement in an Oracle database, but the MySQL database has no equivalent. Two key `ALTER SESSION` statement commands should be at every developer's finger tips: one that enables tracing and another that accesses conditional compilation statements in PL/SQL.

Enable SQL Tracing

You need the `ALTER SESSION` privilege to work with SQL tracing commands. Without the `ALTER SESSION` privilege, you'd raise this type of error when you try to enable or disable SQL tracing with the `DBMS_SESSION` package:

```
BEGIN DBMS_SESSION.SESSION_TRACE_ENABLE(); END;
*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 269
ORA-06512: at line 1
```

The `ALTER SESSION` privilege can be granted by the `SYS` or `SYSTEM` user with this command:

```
GRANT ALTER SESSION TO user_name;
```

The newer and Oracle-recommended way to enable SQL tracing is to use the `DBMS_SESSION` or `DBMS_SYSTEM` packages. The `DBMS_SESSION` package is generally available to developers, while `DBMS_SYSTEM` is reserved for super users and those with DBA privileges.

DBA Insight

DBAs have much wider permissions than average users, and they have great knowledge about the internal workings of the database. DBAs can identify the security ID (SID) and serial number values for any active session.

With this information, DBAs can enable and disable tracing with the DBMS_SYSTEM package. To use it, they need to know how to identify your session's unique identifier and serial number.

A DBA would call the following to enable a user's session for tracing:

```
EXECUTE DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION(sid, serial#);
```

The reason that DBAs assume this role is because they're unwilling or prevented by policy to provide access to individual users. For example, a DBA may enable tracing on a session running the Oracle e-Business Suite in a schema where the developer lacks privileges. After confirming that the user has generated the trace file, the DBA will disable tracing because it consumes physical resources.

The DBMS_SESSION package lets you enable tracing, like this:

```
SQL> EXECUTE DBMS_SESSION.SESSION_TRACE_ENABLE();
```

After you've run the code for tracing purposes, you should disable SQL tracing with another call to the DBMS_SESSION package:

```
SQL> EXECUTE DBMS_SESSION.SESSION_TRACE_DISABLE();
```

There's also an older (legacy) way to enable and disable tracing that still works, notwithstanding Oracle's announcement that it has been deprecated. It requires that you hold the ALTER SESSION privilege. You enable tracing within your session with this command:

```
ALTER SESSION SET sql_trace = true;
```

You can disable it by setting sql_trace to false, like so:

```
ALTER SESSION SET sql_trace = false;
```

You've got the old and then the new techniques. The new technique creates a published method that lets Oracle hide, abstract, or encapsulate logic so that they can change keywords without changing command structures. My recommendation is that you go with Oracle's recommended syntax and move to the DBMS_SESSION package. Table 7-1 lists the tracing levels available in the Oracle database.

The default tracing level for the Oracle database is level 4, which means SQL parsing, executing, fetching, and bind variables are traced. Sometimes you need to get both waits and bind variables, and you can change your session to provide that level of tracing (as qualified in Table 7-1).

The following example changes the tracing level to the maximum setting (level 12):

```
ALTER SESSION
SET EVENTS = '10046 trace name context forever, level 12';
```

Level Flag	Tracing Level
0	No generated statistics
1	Standard trace output includes SQL parsing, executing, and fetching
2	Same as level 1
4	Same as level 1 plus information on bind variables
8	Same as level 1 plus information on waits
12	Same as level 1 plus information on bind variables and waits

TABLE 7-1. *SQL Tracing Levels*

After you've complete the tracing process, don't forget to return your session to the default. That's because the higher level of information incurs a performance cost when you re-enable tracing in the session. You can reset the session to the default with this command:

```
ALTER SYSTEM
SET EVENTS = '10046 trace name context forever, level 4'
```

This should be all you need on the database side, but you'll also need to understand how to use Oracle's `tkprof` command-line utility. `tkprof` formats the trace statistics into a readable (with practice) report. Please check Chapter 21 of the *Oracle Database Performance Tuning Guide* for more on the `tkprof` utility.

Enable Conditional Compilation

Oracle introduced *conditional compilation* in Oracle Database 10*g* Release 2. It allows you to embed debugging logic in your production programs that runs only when you want it to run. This is convenient, because a particular error might show up only when a specific customer is using your code, and there's no inexpensive way to track down the problem without performing a test at the customer's site. Traditionally, somebody would have to write a customer version of the stored program, ship it to the customer, and teach the customer how to test it. Conditional compilation eliminates that need.

A quick sample of how you embed a debugging message with conditional compilation is the following anonymous block PL/SQL program with a conditional `IF` block (you can find more on PL/SQL in Chapter 13):

```
SQL> SET SERVEROUTPUT ON SIZE UNLIMITED
SQL> BEGIN
  2    NULL; -- At least one statement without debug enabled.
  3    $IF $$DEBUG = 1 $THEN
  4      dbms_output.put_line('Debug Level 1 Enabled.');
  5    $END
  6  END;
  7  /
```

The first line sets a SQL*Plus environment variable that allows you to display output from PL/SQL block programs (check Chapter 2 for more details). Line 2 has a single `NULL;` statement, because there must be at least one statement in any block for it to parse (compile) and run. Lines 3 to 5 contain a block that runs only when you've enabled conditional compilation.

Naturally, this requires that you have the `ALTER SESSION` privilege, as discussed in the preceding section. You enable conditional compilation with this statement:

```
SQL> ALTER SESSION SET PLSQL_CCFLAGS = 'debug:1';
```

Now it will print the following message from inside the `IF` block:

```
Debug Level 1 Enabled.
```

As you might have guessed, there would be no recompilation of a stored program required to run the code in debug mode. Likewise, you can disable it by resetting the `PLSQL_CCFLAGS` value.

You've now learned the most critical things available for development through the `ALTER SESSION` statement. You can do many more things at the session level, and I suggest that you explore these.

Tables

Table definitions and constraints change over time for many reasons. These changes occur because developers discover more information about the business model, which requires changes to table definitions. This section examines how you can add, modify, and remove columns or constraints; rename tables, columns, or constraints; and drop tables.

First, however, you need to understand how table definitions are stored in the data catalogs. When you understand the rules of these structures and how they're stored, you can appreciate why SQL is able to let you change so much, so easily.

Data Catalog Table Definitions

The data catalog stores everything by the numbers, which happens to be through surrogate primary keys. The database also maintains a unique index on object names, which means that you can use a name only once per schema in an Oracle or MySQL database. This list of unique values is known as the schema's namespace.

Oracle Data Catalog

As discussed, Oracle maintains the data catalog in the `SYS` schema and provides access to administrative views, which are prefaced by `ALL_`, `DBA_`, and `USER_`. The `USER_` preface is available to every schema for objects that a user owns. The `ALL_` and `DBA_` prefixes give you access to objects owned by others, and only super or administrative users have access privileges to these views.

Many developer tools, such as the Oracle SQL Developer, Quest's Toad for Oracle, or Oracle CASE tools, can easily display information from the data catalog views. Sometimes you need to explore a database for specific information, and the fastest way would be to launch a few quick queries against the data catalog. The catalog view that lets you explore column definitions is the `USER_TAB_COLUMNS` view (short for table's columns).

The following query leverages some SQL*Plus formatting to let you find the definition of a specific table and display it in a single page format:

```
SQL> COLUMN column_id FORMAT 999 HEADING "Column|ID #"
SQL> COLUMN table_name FORMAT A12 HEADING "Table Name"
SQL> COLUMN column_name FORMAT A18 HEADING "Column Name"
SQL> COLUMN data_type FORMAT A10 HEADING "Data Type"
SQL> COLUMN csize FORMAT 999 HEADING "Column|Size"
SQL> SELECT   utc.column_id
  2  ,        utc.table_name
  3  ,        utc.column_name
  4  ,        utc.data_type
  5  ,        NVL(utc.data_length,utc.data_precision) AS csize
  6  FROM     user_tab_columns utc
  7  WHERE    utc.table_name = 'CONTACT';
```

The table name is in uppercase because Oracle maintains metadata text in an uppercase string. You could replace line 7 with the following line that uses the `UPPER` function to promote the text case before comparison, if you prefer to type table names and other metadata values in lowercase or mixed case:

```
  7  WHERE    utc.table_name = UPPER('contact');
```

The query displays the following:

```
Column                                                Column
  ID # Table Name   Column Name        Data Type    Size
------ ------------ ------------------ ---------- ------
     1 CONTACT      CONTACT_ID         NUMBER         22
     2 CONTACT      MEMBER_ID          NUMBER         22
     3 CONTACT      CONTACT_TYPE       NUMBER         22
     4 CONTACT      FIRST_NAME         VARCHAR2       20
     5 CONTACT      MIDDLE_NAME        VARCHAR2       20
     6 CONTACT      LAST_NAME          VARCHAR2       20
     7 CONTACT      CREATED_BY         NUMBER         22
     8 CONTACT      CREATION_DATE      DATE            7
     9 CONTACT      LAST_UPDATED_BY    NUMBER         22
    10 CONTACT      LAST_UPDATE_DATE   DATE            7
```

The `column_id` value identifies the position of columns for `INSERT` statements. The ordered list is set when you define a table with the `CREATE TABLE` statement or is reset when you modify it with an `ALTER TABLE` statement. Columns keep the position location when you change the columns' name or data type. Columns lose their position when you remove them from a table's definition, and when you add them back, they appear at the end of the positional list of values. There's no way to shift their position in an Oracle database without dropping and re-creating the table, which differs from how it's done in MySQL, where you can shift the position of columns with an `ALTER TABLE` statement.

Each column has a data type that defines its physical size. The foregoing example shows that number data types take up to 22 characters, the strings take 20 characters, and the dates take 7 characters. As you learned in Chapter 6, numbers can also have a specification (the values to the right of the decimal point) that fits within the maximum length (or precision) of the data type.

The USER_CONSTRAINTS and USER_CONS_COLUMNS views hold information about constraints. The USER_CONSTRAINTS view holds the descriptive information about the type of constraint, and the USER_CONS_COLUMNS view holds the list of columns participating in the constraint.

You would use a query like this to discover constraints and columns (formatting provided by the SQL*Plus commands):

```
SQL> COLUMN table_name FORMAT A12 HEADING "Table Name"
SQL> COLUMN constraint_name FORMAT A16 HEADING "Constraint|Name"
SQL> COLUMN position FORMAT A8 HEADING "Position"
SQL> COLUMN column_name FORMAT A18 HEADING "Column Name"
SQL> SELECT   ucc.table_name
  2  ,        ucc.constraint_name
  3  ,        uc.constraint_type ||':'|| ucc.position AS position
  4  ,        ucc.column_name
  5  FROM     user_constraints uc JOIN user_cons_columns ucc
  6  ON       uc.table_name = ucc.table_name
  7  AND      uc.constraint_name = ucc.constraint_name
  8  WHERE    ucc.table_name = 'CONTACT'
  9  ORDER BY ucc.constraint_name
 10  ,        ucc.position;
```

which would produce the following output:

```
               Constraint
Table Name     Name               Position Column Name
-------------- ------------------ -------- ------------------
CONTACT        PK_CONTACT_1       P:1      CONTACT_ID
CONTACT        UNIQUE_NAME        U:1      MEMBER_ID
CONTACT        UNIQUE_NAME        U:2      FIRST_NAME
CONTACT        UNIQUE_NAME        U:3      MIDDLE_NAME
CONTACT        UNIQUE_NAME        U:4      LAST_NAME
```

The first line of output reports a single column primary key, which is most often a surrogate primary key. You can tell that because a constraint_type column value represents the code (P) for a PRIMARY KEY constraint, as qualified in Table 7-2. In the query, the position column is the

Constraint Code	**Constraint Meaning**
C	Represents a CHECK or NOT NULL constraint
P	Represents a PRIMARY KEY constraint
R	Represents a FOREIGN KEY, which is really referential integrity between tables and why an R is the code value
U	Represents a UNIQUE constraint

TABLE 7-2. *Constraint Codes and Types*

concatenated result of the constraint type code and position number related to the column name. The remaining lines report a UNIQUE constraint that spans four columns, which is the natural key for the table. It's an imperfect third-normal form (3NF) key for the subject of the table, but it's an adequate natural key for our demonstration purposes.

The material in this section has described how you find definitions for tables and constraints. You'll find this information helpful when you need to change definitions or remove them from tables.

MySQL Data Catalog

MySQL has two data catalogs: an updatable catalog in the `mysql` database and the read-only catalog in the `information_schema` database. The read-only view is the user view of the database catalog.

The `columns` table contains information about columns in tables and is equivalent to the `user_tab_columns` view in Oracle. You can examine the data catalog by using a CASE tool, such as MySQL Workbench, Quest's Toad for MySQL, or other vendor products, or you can write quick queries.

You can inspect a table (`contact` in this case) with the following query:

```
SELECT   c.ordinal_position
,        c.table_name
,        c.column_name
,        c.data_type
,        IFNULL(c.character_maximum_length,c.numeric_precision) AS size
FROM     columns c
WHERE    c.table_name = 'CONTACT';
```

which would display the following:

```
+------------------+------------+------------------+-----------+------+
| ordinal_position | table_name | column_name      | data_type | SIZE |
+------------------+------------+------------------+-----------+------+
|                1 | contact    | contact_id       | int       |   10 |
|                2 | contact    | member_id        | int       |   10 |
|                3 | contact    | contact_type     | int       |   10 |
|                4 | contact    | first_name       | char      |   20 |
|                5 | contact    | middle_name      | char      |   20 |
|                6 | contact    | last_name        | char      |   20 |
|                7 | contact    | created_by       | int       |   10 |
|                8 | contact    | creation_date    | date      | NULL |
|                9 | contact    | last_updated_by  | int       |   10 |
|               10 | contact    | last_update_date | date      | NULL |
+------------------+------------+------------------+-----------+------+
```

The `ordinal_position` column records the order of the column set by the `CREATE TABLE` statement. Like Oracle, these positions remain the same when you change column names or data types, and a column gets the last `ordinal_position` number when a column is dropped and then re-added to the table. Unlike Oracle, MySQL lets you shift the position of columns with the `ALTER TABLE` statement.

The physical size of numeric and string columns are reported, but the size of dates isn't. You can check the data catalog of a table to monitor the impact of changes made by the `ALTER TABLE` statement.

Table-level constraints are in the `table_constraints` and `key_column_usage` views. Table-level constraints should have a name, but no syntax exists to name the `UNIQUE [KEY]` or `PRIMARY KEY` values. MySQL actually has rules for how these are stored in the data catalog.

Dynamic queries require that you set a session variable that matches the one in the query, because, unlike in Oracle, in MySQL, there are no substitution variables that prompt for inputs when you call scripts. The following script uses `@sv_table_name` as a session variable, and you'd set it this way before calling the script from a file:

```
mysql> SET @sv_table_name = 'rental_item';
```

The following query lets you find keys and constraints on MySQL tables, provided you're using the `information_schema` database:

```
SELECT   tc.constraint_name
,        tc.constraint_type
,        kcu.ordinal_position
,        kcu.column_name
FROM     table_constraints tc JOIN key_column_usage kcu
ON       tc.table_name = kcu.table_name
AND      tc.constraint_name = kcu.constraint_name
AND      tc.table_name LIKE @sv_table_name
ORDER BY tc.constraint_name
,        tc.constraint_type
,        kcu.ordinal_position;
```

This query only works "as is" if you're connected to the `information_schema` database. You could enable it to run without using the database by modifying the table declarations in the `FROM` clause, like this:

```
FROM     information_schema.table_constraints tc INNER JOIN
           information_schema.key_column_usage kcu
```

When you've previously set the `@sv_table_name` session variable with a value of `rental_item` (the `table_name`), you would see the following results in the sample database:

```
+----------------+----------------+------------------+-------------------+
| constraint_name | constraint_type | ordinal_position | column_name       |
+----------------+----------------+------------------+-------------------+
| natural_key     | UNIQUE          |                1 | rental_item_id    |
| natural_key     | UNIQUE          |                2 | rental_id         |
| natural_key     | UNIQUE          |                3 | item_id           |
| natural_key     | UNIQUE          |                4 | rental_item_type  |
| natural_key     | UNIQUE          |                5 | rental_item_price |
| PRIMARY         | PRIMARY KEY     |                1 | rental_item_id    |
| rental_item     | UNIQUE          |                1 | rental_item_id    |
| rental_item     | UNIQUE          |                2 | rental_id         |
| rental_item     | UNIQUE          |                3 | item_id           |
| rental_item     | UNIQUE          |                4 | rental_item_type  |
| rental_item     | UNIQUE          |                5 | rental_item_price |
+----------------+----------------+------------------+-------------------+
```

The result set appears to have three possible candidate keys: two unique constraints on the same set of columns and a primary key on what appears to be a surrogate key (a column based on a sequence value). It's likely that a mistake was made in this situation that led to deployment of two unique constraints.

There's also a hidden rule in this result set (that is, I couldn't find it in the documentation): the unique key in MySQL takes the table name as a constraint name. The `natural_key` constraint belongs to a unique index that was added to the table. You should have only one of these per table across all five columns, either a `UNIQUE KEY` or a `UNIQUE INDEX`. My suggestions would be the `UNIQUE KEY`, which should span the five columns.

A `UNIQUE INDEX` on only the four columns that make up the natural key is critical when you run the `LOAD DATA` statement, use `INSERT` statements with the `ON DUPLICATE KEY` clause, or when you run the `REPLACE INTO` statement. The `UNIQUE INDEX` should enclose a natural key and should guarantee that only one row contains those values. The presence of two matching unique constraints doesn't improve your searching speed, but it does slow down the inserting, updating, and deleting of data. You should delete one of these unique indexes, because it would speed your transaction and updates: any `INSERT`, `UPDATE`, or `DELETE` statement results in maintenance that changes the values in the index.

The output also shows a primary key based on a surrogate key. Notice that the constraint name is `PRIMARY` by default. The `PRIMARY` constraint is on the surrogate key column only, but the `UNIQUE rental_item` key includes the surrogate key column, and the surrogate key column leads the `UNIQUE KEY` to speed joins. This technique leverages five columns to speed the B-Tree resolution to any given row.

The following sections explore how you make changes to tables. Part of the following discussion refers to how those changes impact the ordering of columns in the data catalog.

Adding, Modifying, and Dropping Columns

Database tables help you normalize information that supports businesses, research, or engineering processes. In the real world, requirements change, and eventually modifying a table becomes necessary. Some of these changes are relatively trivial, such as changing a column name or data type. Some changes are less trivial, such as when some descriptive item (column) is overlooked, a column isn't large enough to hold a large number or string, or a column of data needs to be renamed or moved to another column. You can make any of these changes using the `ALTER TABLE` statement.

More involved changes occur in three situations:

- When you must change a column's data type when it already contains rows of data
- When you rename a column when existing SQL statements already use the older column
- When you shift the position of adjoining columns that share the same data type

When you need to change a column's data type and the column contains data, you need to develop a data migration plan. Small data migration might entail adding a new column, moving the contents from one column to the other, and then dropping the original column. Unless the database supports an implicit casting operation from the current to future data type, you will need a SQL statement to change the data type explicitly and put it into the new column.

Changing the name of a column in a table seems a trifling thing, but it's not insignificant when application software uses that column name in SQL statements. Any change to the column name will break the application software. You need to identify all dependent code artifacts and ensure that they're changed at the same time you make the changes to the column in the table. A full regression testing plan needs to occur when columns are renamed in tables that support working application software. You can start by querying the `ALL_`, `DBA_`, or `USER_DEPENDENCIES` views that preserve dependencies about tables and stored program units.

Shifting the position of columns can have two different types of impacts. One potential impact is that you break `INSERT` statements that don't use a column list before the `VALUES` clause or subquery. This happens when the columns have different data types, because the `INSERT` statements will fail and raise errors. The other potential impact is much worse, because it produces corrupt data. This happens when you change the position of two columns that share the same data type. This change doesn't break `INSERT` statements in an easily detectable way, as did the other scenario, because it simply inserts the data into the wrong columns. The fix is the same as when you change the positions of columns that have different data types, but the extent of the fix depends on when you notice the problem and how much corrupt data you need to sanitize (fix).

MySQL supports a `SET` clause in `INSERT` statements. This variation on traditional `INSERT` statements provides the closest thing to named notation that exists in SQL. Named notation using the `SET` clause pairs the column name directly with the value.

These risks should be managed by your company's release engineering team and should be subject to careful software configuration management of your code repository. You can leverage the `ALL_`, `DBA_`, and `USER_DEPENDENCIES` views in Oracle to check on dependencies in your software. Unfortunately, there is no equivalent view in MySQL.

Release Engineering

Release engineering is a component of software engineering, and it focuses on how you plan and control the creation and evolution of software—in other words, how you set, enforce, evaluate, and manage software standards. The more time you take to avoid problems, the less time you'll take fixing them.

During the course of normal product release cycles, release engineers manage multiple code branches and dependency trees. Good release engineers invest proactively in software configuration management, because they know it helps identify where problems exist. In a proactive model, you examine process and software dependencies to identify and manage risk exposure. You take corrective action before the event occurs in this approach, which requires you to set and enforce standards that prevent errors.

For example, in a database-centric application, you would check the impact of table definition changes before making them. That's because those types of changes can destabilize your application software. You flag all code modules that depend on existing table and view definitions so you can have them changed concurrently as part of the project. Identifying errors before regression testing by quality and assurance teams is less expensive than fixing these errors after deployment.

This type of good release engineering requires a code repository that tracks dependencies and prevents check-ins that would break other code modules. You need to understand the dependencies in your software process to create an effective process, and you need process automation to manage it. Like any good application that prevents a user from entering garbage data, release management should prevent dependency invalidation. Sometimes this means disallowing changes until you understand their full impact.

The SQL prototypes and mechanics of column maintenance differ between Oracle and MySQL. The next two sections qualify how you make these types of changes in the two databases.

Oracle Column and Constraint Maintenance

The `ALTER TABLE` statement allows you to add columns or constraints, to modify properties of columns and constraints, and to drop columns or constraints. A number of DBA type properties are excluded from the `ATLER TABLE` prototype, and the focus here is on those features that support relational tables.

Here's the basic prototype for the `ALTER TABLE` statement:

```
ALTER TABLE [schema_name.]table_name
[RENAME TO new_table_name]
[READ ONLY]
[READ WRITE]
[{NO PARALLEL | PARALLEL n}]
[ADD
  ({column_name data_type [SORT][DEFAULT value][ENCRYPT key] |
    virtual_column_name} data_type [GENERATED][ALWAYS] AS (expression)}
  ,{column_name data_type [SORT][DEFAULT value][ENCRYPT key] |
    virtual_column_name} data_type [GENERATED][ALWAYS] AS (expression)}
 [, ...])]
[MODIFY
  ({column_name data_type [SORT][DEFAULT value][ENCRYPT key] |
    virtual_column_name} data_type [GENERATED][ALWAYS] AS (expression)}
  ,{column_name data_type [SORT][DEFAULT value][ENCRYPT key] |
    virtual_column_name} data_type [GENERATED][ALWAYS] AS (expression)}
 [, ...])]
[DROP
  (column_name {CASCADE CONSTRAINTS | INVALIDATE} [CHECKPOINT n]
  ,column_name {CASCADE CONSTRAINTS | INVALIDATE} [CHECKPOINT n]
 [, ...])]
[ADD [CONSTRAINT constraint_name]
  {PRIMARY KEY (column_name [,column_name [, ...]) |
   UNIQUE (column_name [,column_name [, ...]) |
   CHECK (check_condition) |
   FOREIGN KEY (column_name [,column_name [, ...])
   REFERENCES table_name (column_name [,column_name [, ...])}]
[MODIFY data_type [SORT][DEFAULT value][ENCRYPT key] |
    virtual_column_name} data_type [GENERATED][ALWAYS] AS (expression)}
 [, ...])]
[RENAME COLUMN old_column_name TO new_column_name]
[RENAME CONSTRAINT old_constraint_name TO new_constraint_name]
```

The following sections provide working examples that add, modify, rename, and drop columns and constraints. Notice that you don't add a `NOT NULL` constraint to a column, but you modify the property of an existing column.

Adding Oracle Columns and Constraints This section shows you how to add columns and constraints. It also provides some guidance on when you can constrain a column as `NOT NULL`.

Here's how you add a new column to a table:

```
SQL> ALTER TABLE rental_item
  2    ADD (rental_item_price  NUMBER);
```

If the table contains no data, you could also add the column with a NOT NULL constraint, like this:

```
SQL> ALTER TABLE rental_item ADD
  2    (rental_item_price  NUMBER CONSTRAINT nn8_rental_item NOT NULL);
```

Adding a column with a NOT NULL constraint fails when rows are included in the table, because when you add the column, its values are empty in all the table rows. The attempt would raise the following error message:

```
ALTER TABLE rental_item
            *
ERROR at line 1:
ORA-01758: table must be empty to add mandatory (NOT NULL) column
```

You can disable the constraint until you've entered any missing values, and then you can re-enable the constraint. After you've added values to all rows of a nullable column, you can constrain the column to disallow null values. The syntax requires you to modify the column, like so:

```
SQL> ALTER TABLE rental_item MODIFY
  2   (rental_item_price  NUMBER CONSTRAINT nn8_rental_item NOT NULL)
  3  DISABLE CONSTRAINT nn8_rental_item;
```

You can also add more than one column at a time with the ALTER TABLE statement. The following would add two columns:

```
SQL> ALTER TABLE rental_item
  2    ADD (rental_item_price  NUMBER)
  3    ADD (rental_item_type   NUMBER);
```

Notice that no comma appears between the two ADD clauses; a comma *is* included when you perform the same statement in MySQL. There's no cute mnemonic to keep this straight, so you'll just have to remember it (probably after you've looked it up a dozen times or so).

That's it for columns. Now you'll see how to add the other four constraints that work with the ADD clause: PRIMARY KEY, CHECK, UNIQUE, and FOREIGN KEY. Note that, as demonstrated in the case of NOT NULL, you can raise errors with these statements when you already have data in a table and it fails to meet the rule of the constraint.

All the following examples work when tables are empty or conform to the constraint rules. After all, what would be the point of a database constraint that didn't constrain behaviors?

This example adds a PRIMARY KEY constraint to a surrogate key column. A primary key in this case restricts a single column's behavior. Here's the syntax:

```
SQL> ALTER TABLE calendar
  2    ADD PRIMARY KEY (calendar_id);
```

The alternative would be to add a `PRIMARY KEY` constraint on the natural key columns, like this:

```
SQL> ALTER TABLE calendar
  2    ADD PRIMARY KEY (month_name, start_date, end_date);
```

The `CHECK` constraint is very powerful in Oracle, because it lets you enforce a single rule or a complex set of rules. In the following example, you add the column and then an out-of-line constraint on the new column:

```
SQL> ALTER TABLE calendar
  2    ADD (month_type VARCHAR2(1))
  3    ADD CONSTRAINT ck_month_type
  4    CHECK(month_type = 'S' AND month_shortname = 'FEB'
  5      OR  month_type = 'M' AND month_shortname IN ('APR','JUN','SEP', 'NOV')
  6      OR  month_type = 'L')
```

The `CHECK` constraint verifies that a `month_type` value must correspond to a combination of its value and the value of the `month_shortname` column. Any month with less than 30 days holds an `S` (short), with 30 days holds an `M` (medium), and with 31 days holds an `L` (long).

The following `UNIQUE` constraint guarantees that no `start_date` and `end_date` combination can exist twice in a calendar table:

```
SQL> ALTER TABLE calendar
  2    ADD CONSTRAINT un_california UNIQUE (start_date, end_date);
```

A `FOREIGN KEY` constraint works with surrogate or natural keys by referencing the table and column or columns that are in its primary key. The next example sets the two foreign keys in a translation table between the `rental` and `item` tables:

```
SQL> ALTER TABLE rental_item
  2    ADD CONSTRAINT fk_rental_id_1
  3        FOREIGN KEY (rental_id) REFERENCES rentals (rental_id)
  4    ADD CONSTRAINT fk_rental_id_2
  5        FOREIGN KEY (item_id) REFERENCES items (item_id);
```

The next example sets a `FOREIGN KEY` on the natural key of the contact table, as shown earlier in this chapter. This references three natural columns and one foreign key column:

```
SQL> ALTER TABLE delegate
  2  ADD CONSTRAINT fk_natural_contact
  3      FOREIGN KEY ( member_id, first_name, middle_name, last_name )
  4      REFERENCES contact (member_id, first_name, middle_name, last_name);
```

The key concept is that you can add both column- and table-level (that is, multiple column) constraints with the `ALTER TABLE` statement. As shown, you can also add the column and then the constraint that goes with it.

Modifying Oracle Columns and Constraints Oracle lets you change column names, data types, and constraints with the `ALTER TABLE` statement. Although column names and data types change routinely during major software upgrades, as discussed, these changes can and do cause problems, because existing code can depend on the type or names of columns and encounter failures when they change unexpectedly.

The following examples demonstrate what you're likely to encounter when working with modifying tables. The first example lets you change the name of a column:

```
SQL> ALTER TABLE calendar
  2    RENAME COLUMN calendar_name TO full_month_name;
```

If you want to change the names of two or more columns in one `ALTER STATEMENT`, you would try something like this:

```
SQL> ALTER TABLE calendar
  2    RENAME COLUMN calendar_name TO full_month_name
  3    RENAME COLUMN calendar_short_name TO short_month_name;
```

But this would fail and raise an `ORA-23290` error:

```
ALTER TABLE calendar
*
ERROR at line 1:
ORA-23290: This operation may not be combined with any other operation
```

The failure occurs because you can't combine a `RENAME` clause with any other clause in an `ALTER TABLE` statement. It's simply disallowed with no more elaboration than that, as you can see in the *Oracle Database Administrator's Guide 11*g.

Data type changes are straightforward when the table contains no data, but you can't change the type when data exists in the column. A quick example attempts to change a `start_date` column using a `DATE` data type to using a `VARCHAR2` data type. The following syntax would work when no data is included in the column but fails when data exists:

```
SQL> ALTER TABLE calendar
  2    MODIFY (start_date VARCHAR2(9));
```

With data in the column, it raises this error message:

```
  MODIFY (start_date VARCHAR2(9))
          *
ERROR at line 2:
ORA-01439: column to be modified must be empty to change datatype
```

You would add a `NOT NULL` constraint to the `start_date` column with the following DML statement:

```
SQL> ALTER TABLE calendar
  2  MODIFY (start_date DATE NOT NULL );
```

The only problem with the foregoing statement is that it creates a `NOT NULL` constraint with a system-generated name. The best practice would assign the constraint a name like so:

```
SQL> ALTER TABLE calendar
  2  MODIFY (start_date DATE CONSTRAINT nn_calendar_1 NOT NULL);
```

Now you know how to rename columns and change column data types. The next section shows you how to drop columns and constraints.

Dropping Oracle Columns and Constraints You drop columns from tables rarely, but the syntax is easy. You would drop the following `short_month_name` column from the `calendar` table with this:

```
SQL> ALTER TABLE calendar
  2  DROP COLUMN short_month_name;
```

You would encounter a problem dropping a column when the column is involved in a table constraint (a constraint across two or more columns). For example, attempting to drop a `middle_name` column from the `contact` table fails when the column is referenced by a multiple column `UNIQUE` constraint (see example in the "Oracle Data Catalog" section earlier in this chapter). The statement would look like this:

```
SQL> ALTER TABLE contact
  2  DROP COLUMN middle_name;
```

and it would raise the following error message:

```
DROP COLUMN middle_name
            *
ERROR at line 2:
ORA-12991: column is referenced in a multi-column constraint
```

This is quite different from MySQL, where you can drop a column that's a member of a multiple-column `UNIQUE` constraint. My preference is Oracle's approach, in which the table can't be dropped until you decide what you're going to do about the `UNIQUE` constraint.

NOTE
Dropping columns when the table contains data can fragment the storage in physical files.

Dropping constraints is easy, because all you need to know is a constraint's name. A query to find the name is available in the "Oracle Data Catalog" section earlier in this chapter. The following drops the `unique_name` constraint from the `contact` table:

```
SQL> ALTER TABLE contact
  2  DROP CONSTRAINT unique_name;
```

This concludes the discussion about adding, modifying, and dropping columns and constraints in an Oracle database. The next section focuses on the same tasks for a MySQL database.

MySQL Column and Constraint Maintenance

This section shows you how to add, modify, and drop columns and constraints from MySQL tables. MySQL provides the same functionality for relational tables, and it also lets you reorder the position of columns in tables.

As with the Oracle prototype, some of the DBA-related options have been excluded here. The abbreviated prototype for the ALTER TABLE statement is shown here:

```
ALTER [{ONLINE | OFFLINE}] [IGNORE] TABLE table_name
ADD {[COLUMN] column_name data_type
      { NOT NULL |
       [CONSTRAINT [constraint_name]] PRIMARY KEY (column_list) |
       [CONSTRAINT [constraint_name]] FOREIGN KEY (column_list)
        REFERENCES table_name (column_list) |
       [CONSTRAINT [constraint_name]] UNIQUE (column_list)}
       [{FIRST | AFTER column_name }] |
     [COLUMN] column_name data_type
      { NOT NULL |
       [CONSTRAINT [constraint_name]] PRIMARY KEY (column_list) |
       [CONSTRAINT [constraint_name]] FOREIGN KEY (column_list)
        REFERENCES table_name (column_list) |
       [CONSTRAINT [constraint_name]] UNIQUE (column_list)} |
     {INDEX | KEY} index_name index_type (index_column_list)
      {KEY_BLOCK_SIZE [=] value |
       USING {BTREE | HASH} |
       WITH PARSER parser_name} |
     {[CONSTRAINT [constraint_name]] PRIMARY KEY (column_list) |
      [CONSTRAINT [constraint_name]] FOREIGN KEY (column_list)
       REFERENCES table_name (column_list) |
      [CONSTRAINT [constraint_name]] UNIQUE (column_list)} |
     ALTER [COLUMN] column_name {SET DEFAULT literal | DROP DEFAULT} |
     CHANGE [COLUMN] old_name new_name data_type
      { NOT NULL |
       [CONSTRAINT [constraint_name]] PRIMARY KEY (column_list) |
       [CONSTRAINT [constraint_name]] FOREIGN KEY (column_list)
        REFERENCES table_name (column_list) |
       [CONSTRAINT [constraint_name]] UNIQUE (column_list)}
       [{FIRST | AFTER column_name}] |
     MODIFY [COLUMN] column_name data_type
      { NOT NULL |
       [CONSTRAINT [constraint_name]] PRIMARY KEY (column_list) |
       [CONSTRAINT [constraint_name]] FOREIGN KEY (column_list)
        REFERENCES table_name (column_list) |
       [CONSTRAINT [constraint_name]] UNIQUE (column_list)}
       [{FIRST | AFTER column_name}] |
     DROP [COLUMN] column_name |
     DROP PRIMARY KEY |
     DROP {INDEX | KEY} index_name |
     DROP FOREIGN KEY foreign_key_name |
     DISABLE KEYS |
     ENABLE KEYS |
     RENAME [TO] new_table_name |
     ORDER BY column_name [, column_name], [...]
```

NOTE
You can find the full prototype in the MySQL Reference, Chapter 12.1.7. (Google and other search engines bring up MySQL 5.1, and you can get the more current document by replacing the "5.1" in the URL with "5.6" or "5.n" [n is current point release].)

Adding MySQL Columns and Constraints Like the equivalent Oracle section, this section shows you how to add columns and constraints. It highlights differences between creating columns and constraints in the two databases.

You can add a column with the `ALTER TABLE` statement, like this:

```
mysql> ALTER TABLE rental_item
    ->   ADD (rental_item_price INT UNSIGNED);
```

Other than the change in data type, the statement is a virtual mirror to the Oracle syntax to add a column. In MySQL, the parentheses surrounding the column name are optional. That means you could also run the statement without parentheses, like this:

```
mysql> ALTER TABLE rental_item
    ->   ADD rental_item_price INT UNSIGNED;
```

You can also add columns with inline constraints, as you can with Oracle. The differences between column- and table-level constraints are the same as the differences between using inline versus out-of-line constraints in the `CREATE TABLE` statement. For example, you can create the `rental_item_price` column with a `NOT NULL` constraint with this syntax:

```
mysql> ALTER TABLE rental_item
    ->   ADD (rental_item_price INT UNSIGNED NOT NULL);
```

As Chapter 6 explains, you can't name `NOT NULL` constraints in MySQL, because they're properties of tables, not separate constraints. Although Oracle disallows the creation of a `NOT NULL` constraint on a column with existing rows, MySQL allows this. MySQL prevents the error from occurring, because it automatically populates numeric columns with a zero, text columns with an empty string, and dates with a zero value (formatted as 0000-00-00).

TIP
Unlike Oracle, an empty string is not the same as a null value in MySQL.

The `ALTER TABLE` statement in MySQL also lets you add multiple columns. Unlike Oracle, MySQL requires commas between the `ADD` clauses, like so:

```
mysql> ALTER TABLE rental_item
    ->   ADD (rental_item_price DECIMAL(10,2))
    -> , ADD (rental_item_type INT UNSIGNED);
```

MySQL's process for adding columns differs from Oracle's in only two ways. You place a comma between `ADD` clauses, and you can assign a `NOT NULL` constraint to columns that have existing rows. MySQL supports PRIMARY KEY, NOT NULL, UNIQUE, and FOREIGN KEY constraints, but it doesn't support the CHECK constraint. Chapter 6 covers the ENUM and SET data types, which provide limited behaviors such as CHECK constraints. Foreign keys are not provided by the MySQL Server because they are a feature of database engines, such as the InnoDB engine.

Oracle requires you to modify a column to add a NOT NULL constraint, and MySQL is no different. However, MySQL provides two different syntax styles for performing the same task, and they're in the "Modifying MySQL Columns and Constraints" section, up next. Here's an example of a NOT NULL constraint:

```
mysql> ALTER TABLE calendar
    ->   MODIFY start_date DATE NOT NULL;
```

Primary keys can be assigned to surrogate (auto-incrementing columns) or to natural keys. You add a PRIMARY KEY constraint to a surrogate key column with the default constraint name like this:

```
mysql> ALTER TABLE calendar
    ->   ADD PRIMARY KEY (calendar_id);
```

Alternatively, you can name the PRIMARY KEY constraint, and add it this way:

```
mysql> ALTER TABLE calendar
    ->   ADD CONSTRAINT pk_calendar PRIMARY KEY (calendar_id);
```

A primary key can also be assigned to multiple columns with the following syntax:

```
mysql> ALTER TABLE rental_item
    ->   ADD CONSTRAINT pk_rental_item
    ->       PRIMARY KEY ( rental_item_id
    ->                   , rental_id
    ->                   , item_id
    ->                   , rental_item_price
    ->                   , rental_item_type );
```

You can use two syntax approaches to assign FOREIGN KEY constraints. One doesn't include a user-assigned name and the other does. You would create a FOREIGN KEY without naming it like this:

```
mysql> ALTER TABLE rental_item
    ->   ADD FOREIGN KEY (rental_id)
    ->       REFERENCES rental (rental_id);
```

Adding a user-assigned name is easy: you place it between the CONSTRAINT and FOREIGN KEY, as shown:

```
mysql> ALTER TABLE rental_item
    ->   ADD CONSTRAINT fk_rental_item_2 FOREIGN KEY (item_id)
    ->       REFERENCES item (item_id);
```

Finally, you can use the `ALTER TABLE` statement to add a `UNIQUE` constraint with the following syntax:

```
mysql> ALTER TABLE calendar
    ->   ADD CONSTRAINT unique_month
    ->       UNIQUE ( month_name
    ->              , month_shortname
    ->              , start_date
    ->              , end_date );
```

Naturally, like all the constraint examples, you can exclude the constraint name when you add a `UNIQUE` constraint with the `ALTER TABLE` statement. Here's the syntax:

```
mysql> ALTER TABLE calendar
    ->   ADD UNIQUE ( month_name
    ->              , month_shortname
    ->              , start_date
    ->              , end_date );
```

As a rule, naming constraints is the best practice. It makes finding your constraints much easier when you need to remove them in MySQL.

Modifying MySQL Columns and Constraints MySQL lets you change column names, data types, nullability, and positions in the list of columns. Changing any of these column properties poses a risk to existing code bases, as discussed earlier in the chapter. That being said, code evolves when the data model evolves to improve software products.

This section shows you how to evolve your table by changing its properties. The first thing to examine is changing the data type of a column. Changing a data type is easy when the table contains no data, but it's tricky when data is present. You can change types with data in the column only when the database can implicitly convert the data from the old to the new data type. Notice that Oracle handles the data type conversions the same way, but the set of data types that qualify for implicit conversion differ.

The following `calendar` table supports these examples on changing data type:

```
+-----------------+------------------+------+-----+---------+-------+
| Field           | Type             | Null | Key | Default | Extra |
+-----------------+------------------+------+-----+---------+-------+
| calendar_id     | int(10) unsigned | NO   | PRI | NULL    |       |
| month_name      | varchar(10)      | YES  | MUL | NULL    |       |
| month_shortname | varchar(3)       | YES  |     | NULL    |       |
| start_date      | date             | YES  |     | NULL    |       |
| end_date        | date             | YES  |     | NULL    |       |
+-----------------+------------------+------+-----+---------+-------+
```

Assuming there's no data in this table, the following changes the data type of the `start_date` column to a string:

```
mysql> ALTER TABLE calendar
    ->   MODIFY start_date VARCHAR(10) NULL;
```

Here's another syntax possibility to perform the same conversion of data types:

```
mysql> ALTER TABLE calendar
    ->   CHANGE start_date start_date VARCHAR(10) NULL;
```

Having changed the data type successfully, let's insert the following data:

```
+-------------+------------+----------------+------------+------------+
| calendar_id | month_name | month_shortname | start_date | end_date   |
+-------------+------------+----------------+------------+------------+
|           1 | January    | Jan             | 01-01-2011 | 2011-01-31 |
+-------------+------------+----------------+------------+------------+
```

The string in the `start_date` column is a noncompliant date format for MySQL. The database has no implicit method for changing the string into a date. The syntax to attempt a change from the `VARCHAR(10)` to a `DATE` data type is shown next:

```
mysql> ALTER TABLE calendar
    ->   MODIFY start_date DATE NOT NULL;
```

It would raise the following error:

```
ERROR 1292 (22007): Incorrect date value: '01-01-2011' for column
```

After updating the value in the `start_date` column to a 4-digit year, 2-digit month, and 2-digit day, the same `ALTER TABLE` statement changes the data type. This same syntax lets you change constraints by treating them like table properties.

You can also rename columns, like so:

```
mysql> ALTER TABLE calendar
    ->   CHANGE month_shortname short_month_name VARCHAR(3);
```

You can rename tables as well as columns with this syntax:

```
mysql> ALTER TABLE calendar
    ->   RENAME TO vcalendar;
```

Changing the nullability of a column is straightforward. You have two options. Here's the first:

```
mysql> ALTER TABLE calendar
    ->   CHANGE start_date start_date DATE NOT NULL
    -> , CHANGE end_date end_date DATE NOT NULL;
```

Or, with the `MODIFY` clause (an Oracle extension to the original MySQL `CHANGE` clause), you can make the `end_date` column null allowed:

```
mysql> ALTER TABLE calendar
    ->   MODIFY end_date DATE NULL;
```

Any column can have a `DEFAULT` value, and the `ALTER TABLE` statement lets you set it when unset, change it when set to another value, and remove it. This syntax sets or resets an integer column's `DEFAULT` value:

```
mysql> ALTER TABLE rental_item
    ->   ALTER COLUMN rental_item_price SET DEFAULT 1;
```

The ALTER COLUMN clause in an ALTER TABLE statement can confuse some, because they expect a MODIFY COLUMN clause. You can remove a DEFAULT column value with this syntax, which sets it to a null value:

```
mysql> ALTER TABLE rental_item
    ->   ALTER COLUMN rental_item_price DROP DEFAULT;
```

The ability to shift column positions is powerful and not available without dropping and re-creating a table in Oracle. You can do this when you want to position columns in a certain order. Consider, for example, the following table. Two new columns have been added as members of a natural primary key. They appear after the who-audit columns (the created by, creation date, last updated by, and last update date columns).

```
+-------------------+------------------+------+-----+---------+
| Field             | Type             | Null | Key | Default |
+-------------------+------------------+------+-----+---------+
| rental_item_id    | int(10) unsigned | NO   | PRI | NULL    |
| rental_id         | int(10) unsigned | NO   | PRI | NULL    |
| item_id           | int(10) unsigned | NO   | PRI | NULL    |
| created_by        | int(10) unsigned | NO   | MUL | NULL    |
| creation_date     | date             | NO   |     | NULL    |
| last_updated_by   | int(10) unsigned | NO   | MUL | NULL    |
| last_update_date  | date             | NO   |     | NULL    |
| rental_item_price | int(10) unsigned | NO   | PRI | NULL    |
| rental_item_type  | int(10) unsigned | NO   | PRI | NULL    |
+-------------------+------------------+------+-----+---------+
```

Because the columns are part of the primary key, it makes more sense to position them with the rest of the primary key. The following command puts them in a better position:

```
mysql> ALTER TABLE rental_item
    ->   MODIFY rental_item_price int unsigned AFTER item_id
    -> , MODIFY rental_item_type int unsigned AFTER rental_item_price;
```

and changes the table definition to this:

```
+-------------------+------------------+------+-----+---------+
| Field             | Type             | Null | Key | Default |
+-------------------+------------------+------+-----+---------+
| rental_item_id    | int(10) unsigned | NO   | PRI | NULL    |
| rental_id         | int(10) unsigned | NO   | PRI | NULL    |
| item_id           | int(10) unsigned | NO   | PRI | NULL    |
| rental_item_price | int(10) unsigned | NO   | PRI | 0       |
| rental_item_type  | int(10) unsigned | NO   | PRI | 0       |
| created_by        | int(10) unsigned | NO   | MUL | NULL    |
| creation_date     | date             | NO   |     | NULL    |
| last_updated_by   | int(10) unsigned | NO   | MUL | NULL    |
| last_update_date  | date             | NO   |     | NULL    |
+-------------------+------------------+------+-----+---------+
```

Notice that the five primary key columns are now shifted into positions 1 to 5. The preceding command also changed the new columns' DEFAULT values to 0. This occurs with many of the clauses in the ALTER TABLE statement. You can't reset the DEFAULT value in the same ALTER TABLE statement, but you can issue this follow-up statement to reset the DEFAULT values:

```
mysql> ALTER TABLE rental_item
    ->   ALTER rental_item_price DROP DEFAULT
    -> , ALTER rental_item_type DROP DEFAULT;
```

In this section, you've seen how to change column names, data types, nullability, and position in the list of columns. The ALTER TABLE statement lets you change one column at a time or a series of columns or constraints by delimiting statements with a comma.

Dropping MySQL Columns and Constraints MySQL lets you remove columns or constraints from tables. There's no recovery point when you drop a column, because it's a DDL statement, and no logging of the deleted data is performed. On the other hand, Oracle does support flashback technology, which provides a recovery point.

MySQL, actually the InnoDB engine, disallows dropping a primary or foreign key column unless you've first removed the constraint. MySQL lets you drop any unconstrained column and any column that's part of a non-unique or unique constraint.

In the "Adding MySQL Columns and Constraints" section earlier in the chapter, an ALTER TABLE statement added a unique_month constraint to the calendar table. Using the query logic from the "MySQL Data Catalog" section, you can see the definition of the unique constraint in the data catalog. It would look like this:

```
+-----------------+-----------------+------------------+-----------------+
| constraint_name | constraint_type | ordinal_position | column_name     |
+-----------------+-----------------+------------------+-----------------+
| unique_month    | UNIQUE          |                1 | month_full_name |
| unique_month    | UNIQUE          |                2 | short_name      |
| unique_month    | UNIQUE          |                3 | start_date      |
| unique_month    | UNIQUE          |                4 | end_date        |
+-----------------+-----------------+------------------+-----------------+
```

The following syntax lets you drop a short_name column from the calendar table, which is also part of the unique constraint:

```
ALTER TABLE calendar
  DROP COLUMN short_name;
```

If you DESCRIBE the table, the column is gone; requerying the data catalog shows you that the unique index has changed from a four-column to a three-column index. Here's the output from the new data catalog query:

```
+-----------------+-----------------+------------------+-----------------+
| constraint_name | constraint_type | ordinal_position | column_name     |
+-----------------+-----------------+------------------+-----------------+
| unique_month    | UNIQUE          |                1 | month_full_name |
| unique_month    | UNIQUE          |                2 | start_date      |
| unique_month    | UNIQUE          |                3 | end_date        |
+-----------------+-----------------+------------------+-----------------+
```

The short_name column is no longer a member of the unique_month index, and likewise the column is no longer part of the calendar table.

The following lets you remove the unique index from the calendar table:

```
mysql> ALTER TABLE calendar
    ->   DROP INDEX unique_month;
```

Although dropping a constraint doesn't remove the data, it deletes a business rule. Some business rules have small impacts on data integrity, but others have potentially large impacts. In general, don't drop anything until you're sure you don't need it.

Dropping Tables

You use the DROP TABLE statement to remove tables from the database. The DROP TABLE statement can fail when other tables or views have referential integrity (foreign key) dependencies on the table or view. The prototypes are shown in the Oracle and MySQL subsections.

Dropping Oracle Tables

Oracle lets you drop only a single table with a DROP TABLE statement. You can drop tables when they contain data or when they're empty. This statement also drops global temporary tables. You can set aside referential integrity by using the CASCADE CONSTRAINTS clause.

The prototype for the DROP TABLE statement is shown here:

```
DROP TABLE [schema_name.]table_name [CASCADE CONSTRAINTS] [PURGE];
```

A DROP TABLE statement against a table that has foreign keys referencing it raises an exception:

```
DROP TABLE parent
           *
ERROR at line 1:
ORA-02449: unique/primary keys in table referenced by foreign keys
```

The CASCADE CONSTRAINTS clause removes dependencies from other tables, such as FOREIGN KEY constraints that reference the table. It does not remove the data from the other table's previous foreign key column, which is important if you plan on re-creating the table and re-importing data. Any re-import of data would need to ensure that primary key values would map to existing foreign key values.

The PURGE keyword is required when you're dropping a partitioned table. The PURGE keyword starts a series of subtransactions that drop the all partitions of the tables. The first successful subtransaction marks the table as UNUSABLE in the data catalog. This flag in the data catalog ensures that only a DROP TABLE statement works against the remnants of the table. If you encounter a problem trying to access a table, you can query the status column value to see if it's unusable. The UNUSABLE column is in the ALL_, DBA_, or USER_TABLES, USER_PART_TABLES, USER_ALL_TABLES, USER_OBJECT_TABLES, or USER_OBJECTS administrative views.

Dropping MySQL Tables

In MySQL, unlike Oracle, the DROP TABLE statement doesn't manage referential integrity issues with a CASCADE clause. MySQL does let you check whether a table exists before you attempt to drop it (IF EXISTS clause), and it lets you use a single DROP TABLE statement to remove multiple tables. It would be great to see these two features in Oracle.

NOTE
A quick reminder on foreign key features: Foreign keys aren't native components in the MySQL Server; they're implementation elements of MySQL engines. Users deploying referential integrity use the InnoDB engine.

Here's the prototype for the MySQL `DROP` statement:

```
DROP [TEMPORARY] TABLE [IF EXISTS] table_name [, table_name [, ...]]
[{RESTRICT | CASCADE}];
```

NOTE
The `CASCADE` keyword does not change the behavior in MySQL but is permitted to make porting easier.

While working with the InnoDB engine and referential integrity, you can disable dependencies by setting the following session variable:

```
SET FOREIGN_KEY_CHECKS = 0;
```

This command disables referential integrity checking for the InnoDB engine. It's handy in a re-runnable script that drops and creates tables. Just make sure you re-enable that checking at the end of your script, like this:

```
SET FOREIGN_KEY_CHECKS = 1;
```

The `TEMPORARY` keyword in the `DROP` statement doesn't stop any transactions currently working with `TEMPORARY` tables, because it simply waits on their completion. Because temporary tables are visible only within the scope of a session, the `TEMPORARY` keyword doesn't check access rights to the table.

Indexes

As mentioned in Chapter 6, *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.

There are many reasons for fixing indexes in Oracle and MySQL, and the following is a list of some of the major reasons for altering an index:

- Rebuilding or coalescing an existing index
- Deallocating unused space or allocating new space
- Enabling and specifying the degree of parallelism for storing the index
- Changing storage parameters to improve index performance
- Enabling or disabling logging
- Enabling or disabling key compression
- Marking the index as unusable

- Making the index invisible
- Renaming the index
- Starting or stopping index usage monitoring

The next two sections review some basics of using indexes from a developer's perspective. Clearly, storage and parallel optimization belong to the DBAs, because they know the critical resources of CPUs, memory, and disk space.

Oracle Index Maintenance

The Oracle `ALTER INDEX` statement lets you manage indexes, and `DROP INDEX` lets you remove indexes. You can also use the `ALTER TABLE` statement to enable primary key column(s) to use indexes.

The prototype for the `ALTER INDEX` statement, minus the DBA options, is shown here:

```
ALTER INDEX [schema_name.]index_name [COMPILE] |
[{ENABLE | DISABLE}] |
[UNUSABLE] |
[REBUILD [{PARTITION partition_clause |
          SUBPARTITION subpartition_clause |
          [{REVERSE | NOREVERSE}]}]] |
[{VISIBLE | INVISIBLE}] |
[RENAME TO new_index_name] |
[COALESCE] |
[{MONITORING | NOMONITORING} USAGE] |
```

The first thing developers want to do when they've discovered poor throughput in a query is disable the index to see what impact it has on their code. I'll show you how to do that, but it's generally better done by modifying the query so that it doesn't run the index by concatenating an empty string to a string or by adding a 0 to a number or date.

You can enable an index when it's necessary:

```
ALTER INDEX nk_rental_item ENABLE;
```

Or you can disable it:

```
ALTER INDEX nk_rental_item DISABLE;
```

Sometimes you want to mark an index to rebuild it. You do that with the `UNUSABLE` keyword:

```
ALTER INDEX nk_rental_item UNUSABLE;
```

You can rebuild an index, provided it isn't partitioned, with this:

```
ALTER INDEX nk_rental_item REBUILD;
```

If you don't have the space to rebuild an index online, you can try offline rebuilding, or *coalescing* the index. Coalescing is like defragmenting a disk. When you coalesce an index it reorganizes the data and maintains fully free blocks, which eliminate the cost of releasing and reallocating blocks. Many DBAs choose to coalesce indexes because of the speed, absence of locking, and minimal incremental disk space requirements.

This syntax coalesces an index:

```
ALTER INDEX nk_rental_item COALESCE;
```

The idea of visibility or invisibility might seem odd, but the `VISIBLE` and `INVISIBLE` keywords make an index visible or invisible to the Oracle cost-based optimizer for queries. DML statement, such as `INSERT`, `UPDATE`, and `DELETE` statements, maintain an invisible index, but queries don't use it. As a rule, from Oracle 11*g* forward, you want the optimizer to see indexes. You can discover whether an index is invisible by checking the visibility column in the `ALL_`, `DBA_`, or `USER_INDEXES` view.

TIP
Setting an index to `INVISIBLE` is without merit when the DBA has set the `OPTIMIZER_USE_INVISIBLE_INDEXES` parameter to true, because it makes all invisible indexes visible.

You make an index visible with this:

```
ALTER INDEX nk_rental_item VISIBLE;
```

Renaming an index is something to consider if you originally chose a poorly descriptive index name. Here's the syntax:

```
ALTER INDEX nk_rental_item RENAME TO naturalkey_rental_item;
```

You can use the `ALTER TABLE` statement to let a table's primary key column use an existing index, like so:

```
ALTER TABLE rental_item ENABLE PRIMARY KEY USING nk_rental_item;
```

Don't forget that some views in the data catalog let you see the indexes you've already created.

MySQL Index Maintenance

Although index maintenance isn't as diverse in MySQL as it is in Oracle, you still have some options. MySQL uses the `ALTER TABLE` statement to perform index maintenance, such as adding or dropping an index.

Here's the syntax to add a non-unique index:

```
mysql> ALTER TABLE calendar
    ->   ADD INDEX unique_month ( calendar_month_name
    ->                          , start_date
    ->                          , end_date ) USING BTREE;
```

A unique index requires only one additional word, as shown here:

```
mysql> ALTER TABLE calendar
    ->   ADD UNIQUE INDEX unique_month ( calendar_month_name
    ->                                 , start_date
    ->                                 , end_date ) USING BTREE;
```

You can remove the index with the DROP INDEX clause, like so:

```
mysql> ALTER TABLE calendar
    ->   DROP INDEX unique_month;
```

You cannot disable a unique index, but you can turn off a non-unique index while you perform some updates to a table. This lets you fix data and regenerate indexes. Here's the syntax:

```
mysql> ALTER TABLE calendar DISABLE KEYS;
```

Don't forget to re-enable the non-unique index after you're done performing the table maintenance:

```
mysql> ALTER TABLE calendar ENABLE KEYS;
```

Views

View creation is covered in Chapter 11. Views are stored queries and Chapter 11 shows you how they work. The syntax for removing a view from Oracle or MySQL is very similar and is shown in the following sections.

Oracle Drop Views

Oracle doesn't provide syntax for conditionally dropping tables or views as does MySQL. You can write a PL/SQL block that verifies whether the view exists, by querying the USER_OBJECTS or USER_VIEWS administrative view.

Here's the prototype for dropping a view:

```
DROP VIEW [schema_name.]view_name CASCADE CONSTRAINTS;
```

The CASCADE CONSTRAINTS is required when other tables have foreign key dependencies on the view. Without the clause, an error is raised and dropping the view is disallowed. Oracle supports dropping only a single object with the DROP VIEW statement.

MySQL Drop Views

MySQL lets you remove views. The IF EXISTS clause lets you check whether the view exists before you try to remove it and allows you to write conditional DROP statements.

Here's the prototype for a removing a view:

```
DROP VIEW view_name [, view_name2 [, ...]]  IF EXISTS;
```

Notes are generated as warning messages when views are present, but when the view doesn't exist, an error is suppressed and not raised. The statement also lets you remove a comma-separated list of views, which isn't possible with the Oracle syntax.

Summary

The chapter explored how you can maintain structures in the database. It explored the various syntax combinations that support modifying, such as users, databases, sessions, tables, views, and indexes. The chapter compared maintenance methods between the Oracle and MySQL database.

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 *user* in Oracle is synonymous with a *schema*.
2. **True** ☐ **False** ☐ In an Oracle database, a user can change his or her password with an `ALTER USER` statement.
3. **True** ☐ **False** ☐ You must know a user's password to change their password as a super user, such as `SYSTEM`, in an Oracle database.
4. **True** ☐ **False** ☐ In a MySQL database, a user can change his or her password with an `ALTER USER` statement.
5. **True** ☐ **False** ☐ You can add a `NOT NULL` constraint to an existing column when it contains null values.
6. **True** ☐ **False** ☐ You can lock and unlock accounts in MySQL.
7. **True** ☐ **False** ☐ Oracle has deprecated tracing using the `DBMS_SYSTEM` package and replaced it with the `ALTER SESSION` statement.
8. **True** ☐ **False** ☐ Oracle lets you embed logic in your stored functions and procedures that is run only when you enable specialized session variables.
9. **True** ☐ **False** ☐ It's always best to define a `UNIQUE KEY` and `INDEX` when they work with the same columns in a MySQL database.
10. **True** ☐ **False** ☐ You can modify a column's data type when it contains data in an Oracle database, but not a MySQL database.
11. Which of the following is not a valid constraint flag in Oracle?
    - **A.** A `P`
    - **B.** A `N`
    - **C.** A `C`
    - **D.** A `R`
    - **E.** A `U`
12. Which rule applies in an Oracle database to changing the data type of a column?
    - **A.** Any column can be changed when the table is empty.
    - **B.** All columns can be changed except those that are members of a primary key, whether the table contains data or not.
    - **C.** All variable length string columns (`VARCHAR2`) can be changed to a `DATE` data type column regardless of the format mask involved when the table is full.
    - **D.** All non-key columns can be changed when the table is full.
    - **E.** None of the above

13. Which of the following isn't a keyword in an Oracle `ALTER TABLE` statement?

A. An `ALTER` clause

B. A `CHANGE` clause

C. A `MODIFY` clause

D. A `DROP` clause

E. An `ADD` clause

14. Which of the following isn't a keyword in a MySQL `ALTER TABLE` statement?

A. A `CHANGE` clause

B. A `MODIFY` clause

C. A `DROP` clause

D. A `DISABLE` clause

E. A `READ ONLY` clause

15. Which of the following constraints can't be added to a table in MySQL?

A. The `PRIMARY KEY` constraint

B. The `FOREIGN KEY` constraint

C. The `CHECK` constraint

D. The `UNIQUE` constraint

E. The `NOT NULL` constraint