Class Note: Chapter 7

Building a Basic Relational Schema

(Updated May 17, 2016)

[The "class note" is the typical material I would prepare for my face-to-face class. Since this is an Internet based class, I am sharing the notes with everyone assuming you are in the class.]

Every database application is built upon a set of related database objects that store the application's data and allow the application to function. This chapter introduces Oracle database objects, such as tables, and discusses the logical concepts of database objects. Discussions of data storage (storage parameters, partitioning, and so on) will come in subsequent chapters of this course. This chapter's topics include:

- Schemas
- Tables
- Integrity constraints
- Views
- Sequences
- Synonyms
- Indexes

Chapter Prerequisites

To practice the hands on exercises in this chapter, you need to start SQL*Plus and run the following command script at SQL> prompt:

```
location\\Sql\chap07.sql
```

Where *location* is the file directory where you expanded the supplemental files downloaded from course web site. For example, after starting SQL*Plus and connecting as SCOTT, you can run this chapter's SQL command script using the SQL*Plus command @, as in the following example (assuming that your chap07.sql file is in C:\temp\\Sql).

```
SQL> @C:\temp\\Sql\chap07.sql;
```

Once the script completes successfully, leave the current SQL*Plus session open and use it to perform this chapter's exercises in the order that they appear.

7.1. Schemas

It is easier to solve most problems in life when you are organized and have a well designed plan to achieve your goal. If you are unorganized, you will most certainly realize your goals less efficiently, if at all. Designing an information management system that uses Oracle is no different.

Databases organize related objects within a database schema. For example, it is typical to organize within a single database schema all of the tables and other database objects necessary to support an application. This way, it is clear that the purpose of a certain table or other database object is to support the corresponding application system. Figure 7-1 illustrates the idea of an application schema.

7.1.1. Schemas, an Entirely Logical Concept

It's important to understand that schemas do not physically organize the storage of objects. Rather, schemas logically organize related database objects. In other words, the logical organization of database objects within schemas is purely for the benefit of organization and has absolutely nothing to do with the physical storage of database objects.

The logical organization that schemas offer can have practical benefits. For example, consider an Oracle database with two schemas, S1 and S2. Each schema can have a table called T1. Even though the two tables share the same name, they are uniquely identifiable because they are within different database schemas. Using standard dot notation, the complete names for the different tables would be S1.T1 and S2.T1.

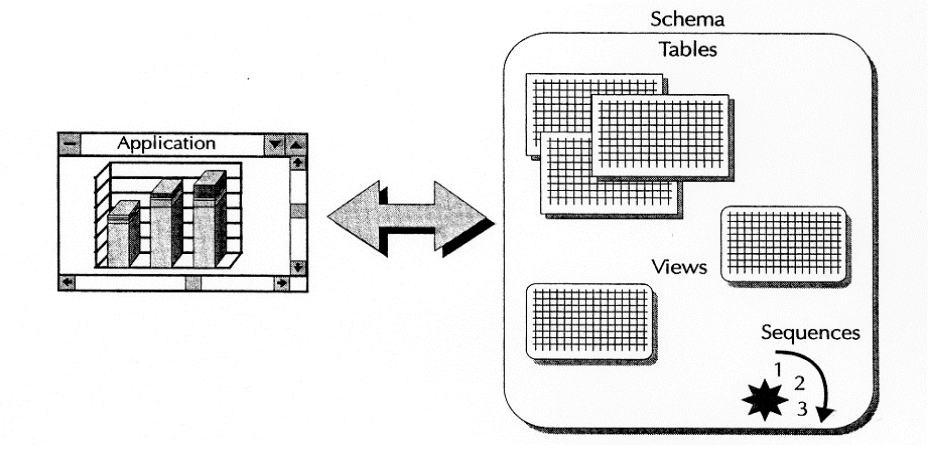

FIGURE 7-1 . *A schema is a logical organization of related database objects*

If the idea of logical versus physical organization is confusing to you, consider how operating systems organize files on disk. The layout of folders and files in a graphical file management utility, such as the Microsoft Windows Explorer, does not necessarily correspond to the physical location of the folders and files on a particular disk drive. File folders represent the logical organization of operating system files. The

underlying operating system decides where to physically store the blocks for each operating system file, independent of the logical organization of encompassing folders.

Subsequent chapters of this book explain more about how Oracle can physically organize the storage of database objects using physical storage structures.

7.1.2. The Correlation of Schemas and Database User Accounts

With Oracle, the concept of a database schema is directly tied to the concept of a database user. That is, a schema in an Oracle database has a “one to one” correspondence with a user account such that a user and the associated schema have the same name. As a result, people who work with Oracle often blur the distinction between users and schemas, commonly saying things like "the user SCOTT owns the EMP and DEPT tables" rather than "the schema SCOTT contains the EMP and DEPT tables." Although these two sentences are more or less equivalent, understand that there might be a clear distinction between users and schemas with relational database implementations other than Oracle. Therefore, while the separation between users and schemas might seem trivial for Oracle, the distinction can be very important if you plan to work with other database systems.

NOTE
The scripts that you executed to support the practice exercises of this chapter and previous chapters create new database users/schemas (practice03, practice04, and so on) that contain similar sets of tables and other database objects (PARTS, CUSTOMERS, and so on).

7.2. Database Tables

Tables are the basic data structure in any relational database. A *table* is nothing more than an organized collection of *records*, or *rows*, that all have the same *attributes*, or *columns*. Figure 7-2 illustrates a typical CUSTOMERS table in a relational database.

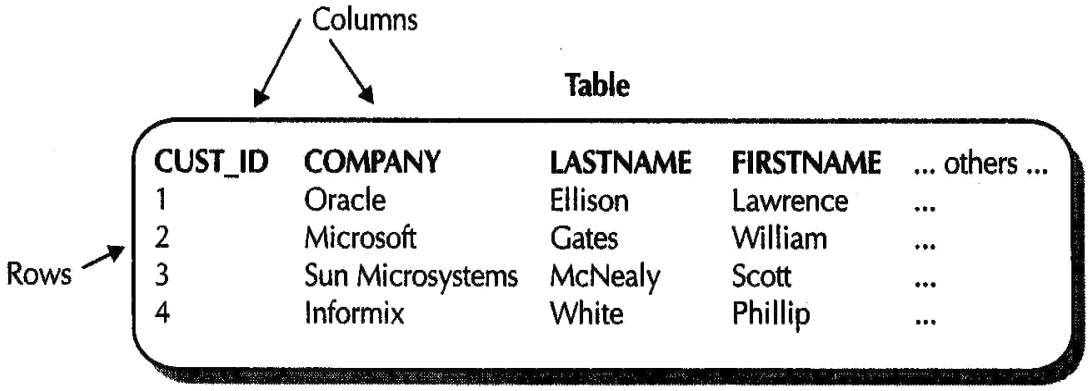

FIGURE 7-2. *A table is a set of records with the same attributes*

Each customer record in the example CUSTOMERS table has the same attributes, including an ID, a company name, a last name, a first name, and so on. When you create tables, the two primary things that you must consider are the following:

- The table's columns, which describe the table's structure
- The table's integrity constraints, which describe the data that is acceptable within the table

The following sections explain more about columns and integrity constraints.

7.2.1. Columns and Datatypes

When you create a table for an Oracle database, you establish the structure of the table by identifying the columns that describe the table's attributes. Furthermore, every column in a table has a datatype, which describes the basic type of data that is acceptable in the column, much like when you declare the datatype of a variable in a PL/SQL or Java program. For example, the ID column in the CUSTOMERS table uses the basic Oracle datatype NUMBER because the column stores ID numbers. Oracle supports many fundamental datatypes that you can use when creating a relational database table and its columns. Table 7-1 and the following sections describe the most commonly used Oracle datatypes.

Datatype	**Description**
CHAR(*size*)	Stores fixed length character strings up to 2,000 bytes
VARCHAR2(*size*)	Stores variable length character strings up to 4,000 bytes
NUMBER(*precision*, *scale*)	Stores any type of number
DATE	Stores dates and times
CLOB	Stores single byte character large objects (CLOBs) up to 40 gigabytes

TABLE 7-1. *The Most Commonly Used Oracle Datatypes*

7.2.1.1. CHAR and VARCHAR2: Oracle's Character Datatypes

Oracle's CHAR and VARCHAR2 are the datatypes most commonly used for columns that store character strings. The Oracle datatype CHAR is appropriate for columns that store fixed length character strings, such as two letter USA state codes.

Alternatively, the Oracle datatype VARCHAR2 is useful for columns that store variable-length character strings, such as names and addresses. The primary difference between these character datatypes relates to how Oracle stores strings shorter than the maximum length of a column.

- When a string in a CHAR column is less than the column's size, Oracle pads (appends) the end of the string with blank spaces to create a string that matches the column's size.
- When a string in a VARCHAR2 column is less than the column's maximum size, Oracle stores only the string and does not pad the string with blanks.

Thus, when the strings in a column vary in length, Oracle can store them more efficiently in a VARCHAR2 column than in a CHAR column. Oracle also uses different techniques for comparing CHAR and VARCHAR2 strings to one another so that comparison expressions evaluate as expected.

7.2.1.2. NUMBER: Oracle's Numeric Datatype

To declare columns that accept numbers, you can use Oracle's NUMBER datatype. Rather than having several numeric datatypes, Oracle's NUMBER datatype supports the storage of all types of numbers, including integers, floating point numbers, real numbers, and so on. You can limit the domain of acceptable numbers in a column by specifying a precision and a scale for a NUMBER column.

7.2.1.3. DATE: Oracle's Time Related Datatype

When you declare a table column with the DATE datatype, the column can store all types of time related information, including dates and associated times.

7.2.1.4. CLOBs, BLOBs, and More: Oracle's Multimedia Datatypes

Because databases are secure, fast, and safe storage areas for data, they are often employed as data repositories for multimedia applications. To support such content rich applications, Oracle supports several different large object (LOB) datatypes that can store unstructured information, such as text documents, static images, video, audio, and more.

- A *CLOB* column stores character objects, such as documents.
- A *BLOB* column stores large binary objects, such as graphics, video clips, or sound files.
- A *BFILE* column stores file pointers to LOBS managed by file systems external to the database. For example, a BFILE column might be a list of filename references for photos stored on a CD ROM.

The following section explains several other important LOB characteristics, comparing LOBS with some older Oracle large object datatypes.

7.2.1.5. Contrasting LOBs with Older Oracle Large Object Datatypes

For backward compatibility, Oracle continues to support older Oracle datatypes designed for large objects, such as *LONG* and *LONG RAW*. However, Oracle's newer LOB datatypes have several advantages over the older Oracle large datatypes.

- A table can have multiple CLOB, BLOB, and BFILE columns. In contrast, a table can have only one LONG or LONG RAW column.
- A table stores only small locators (pointers) for the LOBs in a column, rather than the actual large objects themselves. In contrast, a table stores data for a LONG column within the table itself.
- A LOB column can have storage characteristics independent from those of the encompassing table, making it easier to address the large disk requirements typically associated with LOBS. For example, it's possible to separate the storage of primary table data and related LOBS in different physical locations (for example, disk drives). In contrast, a table physically stores the data for a LONG column in the same storage area that contains all other table data.
- Applications can efficiently access and manipulate pieces of a LOB. In contrast, applications must access an entire LONG field as an atomic (indivisible) piece of data.

Before migrating or designing new multimedia applications for Oracle, consider the advantages of Oracle's newer LOB datatypes versus older large object datatypes.

7.2.1.6. Oracle's National Language Support Character Datatypes

Oracle's National Language Support (NLS) features allow databases to store and manipulate character data in many languages. Some languages have character sets that require several bytes for each character. The special Oracle datatypes NCHAR, NVARCHAR2, and NCLOB are datatypes that are counterparts to the CHAR, VARCHAR2, and CLOB datatypes, respectively.

7.2.1.7. ANSI Datatypes and Others

Oracle also supports the specification of Oracle datatypes using other standard datatypes. For example, Table 7-2 lists the ANSI/ISO (American National Standards Institute/International Organization for Standardization) standard datatypes that Oracle supports.

7.2.1.8. Default Column Values

When you declare a column for a table, you can also declare a corresponding default column value. Oracle uses the default value of a column when an application inserts a new row into the table but omits a value for the column. For example, you

might indicate that the default value for the ORDERDATE column of the ORDERS table be the current system time when an application creates a new order.

NOTE
Unless you indicate otherwise, the initial default value for a column is null (an absence of value).

This ANSI/ISO datatype	converts to this Oracle dat
CHARACTER CHAR	CHAR
CHARACTER VARYING CHAR VARYING	VARCHAR2
NATIONAL CHARACTER NATIONAL CHAR NCHAR	NCHAR
NATIONAL CHARACTER VARYING NATIONAL CHAR VARYING NCHAR VARYING	NVARCHAR2
NUMERIC DECIMAL INTEGER INT SMALLINT FLOAT DOUBLE PRECISION REAL	NUMBER

TABLE 7-2. *Oracle Supports the Specification of Oracle Datatypes Using ANSI/ISO Standard Datatypes*

7.3. Creating and Managing Tables

Now that you understand that the structure of a table is defined by its columns and that each column in a table has a datatype, it's time to learn the basics of creating and managing the structure of tables in an Oracle database. The following practice exercises introduce the SQL commands CREATE TABLE and ALTER TABLE.

EXERCISE 7.1: Creating a Table

You create a table using the SQL command CREATE TABLE. For the purposes of this simple exercise, the basic syntax for creating a relational database table with the CREATE TABLE command is as follows:

```
CREATE TABLE [schema.] table
 ( column datatype [DEFAULT expression]
 [, column datatype [DEFAULT expression] ]
 [ ... other columns ... ]
)
```

Using your current SQL*Plus session, enter the following command to create the familiar PARTS table in this lesson's practice schema.

```
CREATE TABLE parts(
 id INTEGER,
 description VARCHAR2(250),
 unitprice NUMBER(10, 2),
 onhand INTEGER,
 reorder INTEGER
);
```

Your current schema (practice07) now has a new table, PARTS, that you can query, insert records into, and so on. Notice that the PARTS table has five columns.

- The statement declares the ID, ONHAND, and REORDER columns with the ANSI/ISO datatype INTEGER, which Oracle automatically converts to the Oracle datatype NUMBER with 38 digits of precision.
- The statement declares the DESCRIPTION column with the Oracle datatype VARCHAR2 to accept variable length strings up to 250 bytes in length.
- The statement declares the UNITPRICE column with the Oracle datatype NUMBER to hold numbers up to ten digits of precision and to round numbers after two digits to the right of the decimal place.

Before continuing, create the familiar CUSTOMERS table using the following CREATE TABLE statement:

```
CREATE TABLE customers (
 id INTEGER,
 lastname VARCHAR2(100),
 firstname VARCHAR2(50),
 companyname VARCHAR2(100),
 street VARCHAR2(100),
 city VARCHAR2(100),
 state VARCHAR2(50),
 zipcode NUMBER(10),
 phone VARCHAR2(30),
 fax VARCHAR2(30),
 email VARCHAR2(100)
);
```

When you are designing the tables in a database schema, sometimes it can be tricky to choose the correct datatype for a column. For example, consider the ZIPCODE column in the CUSTOMERS table of the previous example, declared with the NUMBER datatype. Consider what will happen when you insert a customer record with the ZIPCODE "01003" Oracle is going to store this number as "1003", certainly not what you intended. Furthermore, consider what would happen if you insert a customer record with a zip code and an extension such as "91222 0299"Oracle is going to evaluate this numeric expression and store the resulting number "90923". These two simple examples illustrate that the selection of a column's datatype is certainly an important consideration, and is not to be taken lightly. In the next practice exercise, you'll learn how to change the datatype of the ZIPCODE column to store postal codes correctly. To complete this exercise, create the SALESREPS table with the following CREATE TABLE statement.

```
CREATE TABLE salesreps (
 id INTEGER,
 lastname VARCHAR2(100),
 firstname VARCHAR2(50),
 commission NUMBER(38)
);
```

NOTE
The examples in this section introduce the basics of the CREATE TABLE command. Subsequent exercises in this and other chapters demonstrate more advanced clauses and parameters of the CREATE TABLE command.

EXERCISE 7.2: Altering and Adding Columns in a Table

After you create a table, you can alter its structure using the SQL command ALTER TABLE. For example, you might want to change the datatype of a column, change a column's default column value, or add an entirely new column altogether. For the purposes of this simple exercise, the basic syntax of the ALTER TABLE command for adding or modifying a column in a relational database table is as follows:

```
ALTER TABLE [schema.]table
[ ADD column datatype [DEFAULT expression] ]
[ MODIFY column [datatype] [DEFAULT expression] ]
```

For example, enter the following ALTER TABLE statement, which modifies the datatype of the ZIPCODE column in the CUSTOMERS table that you created in Exercise 7.1.

```
ALTER TABLE customers
MODIFY zipcode VARCHAR2(50);
```

NOTE
You can change the datatype of a column, the precision or scale of a NUMBER column, or the size of a CHAR or VARCHAR2 column only

when the table does not contain any rows, or when the target column is null for every record in the table.

Suppose that you realize the CUSTOMERS table must be able to track each customer's sales representative. Enter the following ALTER TABLE statement, which adds the column S_ID to record the ID of a customer's sales representative.

```
ALTER TABLE customers
ADD s_id INTEGER;
```

NOTE
The examples in this section introduce the basics of the ALTER TABLE command. Subsequent exercises in this chapter and others demonstrate more advanced clauses and parameters of the ALTER TABLE command.

7.4. Data Integrity and Integrity Constraints

Data integrity is a fundamental principle of the relational database model. Saying that a database has integrity is another way of saying that the database contains only accurate and acceptable information. For obvious reasons, data integrity is a desirable attribute for a database.

To a small degree, a column's datatype establishes a more limited domain of acceptable values for the column it limits the type of data that the column can store. For example, a DATE column can contain valid dates and times, but not numbers or character strings. But while simple column datatypes are useful for enforcing a basic level of data integrity, there are typically more complex integrity rules that must be enforced in a relational database. In fact, the relational database model, itself, outlines several inherent data integrity rules that a relational database management system (RDBMS) must uphold. The next few sections describe these common integrity rules and related issues.

7.4.1. Domain Integrity, Nulls, and Complex Domains

Domain integrity defines the domain of acceptable values for a column. For example, you might have a rule that a customer record is not valid unless the customer's state abbreviation code is one of the fifty or so USA state codes.

Besides using column datatypes, Oracle supports two types of integrity constraints that allow you to further limit the domain of a column:

- A column can have a not null constraint to eliminate the possibility of nulls (absent values) in the column.
- You can use a check constraint to declare a complex domain integrity rule as part of a table. A check constraint commonly contains an explicit list of the acceptable values for a column. For example, "M" and "F" in a column

that contains gender information; "AL", "AK", ... "WY" in a column that contains USA state codes; and so on.

7.4.2. Entity Integrity, Primary Keys, and Alternate Keys

Entity integrity ensures that every row in a table is unique. As a result, entity integrity eliminates the possibility of duplicate records in the table and makes every row in the table uniquely identifiable.

The primary key of a table ensures its entity integrity. A primary key is a column that uniquely identifies each row in a table. Typically, tables in a relational database use ID type columns as primary keys. For example, a customer table might include an ID column to uniquely identify the customer records within. This way, even if two customers, say John Smith and his son John Smith (Jr.), have the same name, address, phone number, and so on, they have distinct ID numbers that make them different.

A table's primary key is sometimes a composite key; that is, it is composed of more than one column. For example, the primary key in a typical line item table of an order entry system might have a composite primary key that consists of the ORDER ID and ITEM ID columns. In this example of a composite primary key, many line item records can have the same line item ID (1, 2, 3, ...), but no two line item records can have the same order ID and line item ID combination (order ID 1, line item IDs 1,2,3, ...; order ID 2, line item IDs 1,2,3, ...; and so on).

Optionally, a table might require secondary levels of entity integrity. Alternate keys are columns or sets of columns that do not contain duplicate values within them. For example, the EMAIL column in an employee table might be made an alternate key to guarantee that all employees have unique e-mail addresses.

7.4.3. Referential Integrity, Foreign Keys, and Referential Actions

Referential integrity, sometimes called relation integrity, establishes the relationships among different columns and tables in a database. Referential integrity ensures that each column value in a foreign key of a child (or detail) table matches a value in the primary or an alternate key of a related parent (or master) table. For example, a row in the CUSTOMERS (child) table is not valid unless the customer's S_ ID field refers to a valid sales representative ID in the SALESREPS (parent) table. When the parent and child table are the same, this is called self-referential integrity. Figure 7-3 illustrates the terminology and concepts related to referential integrity.

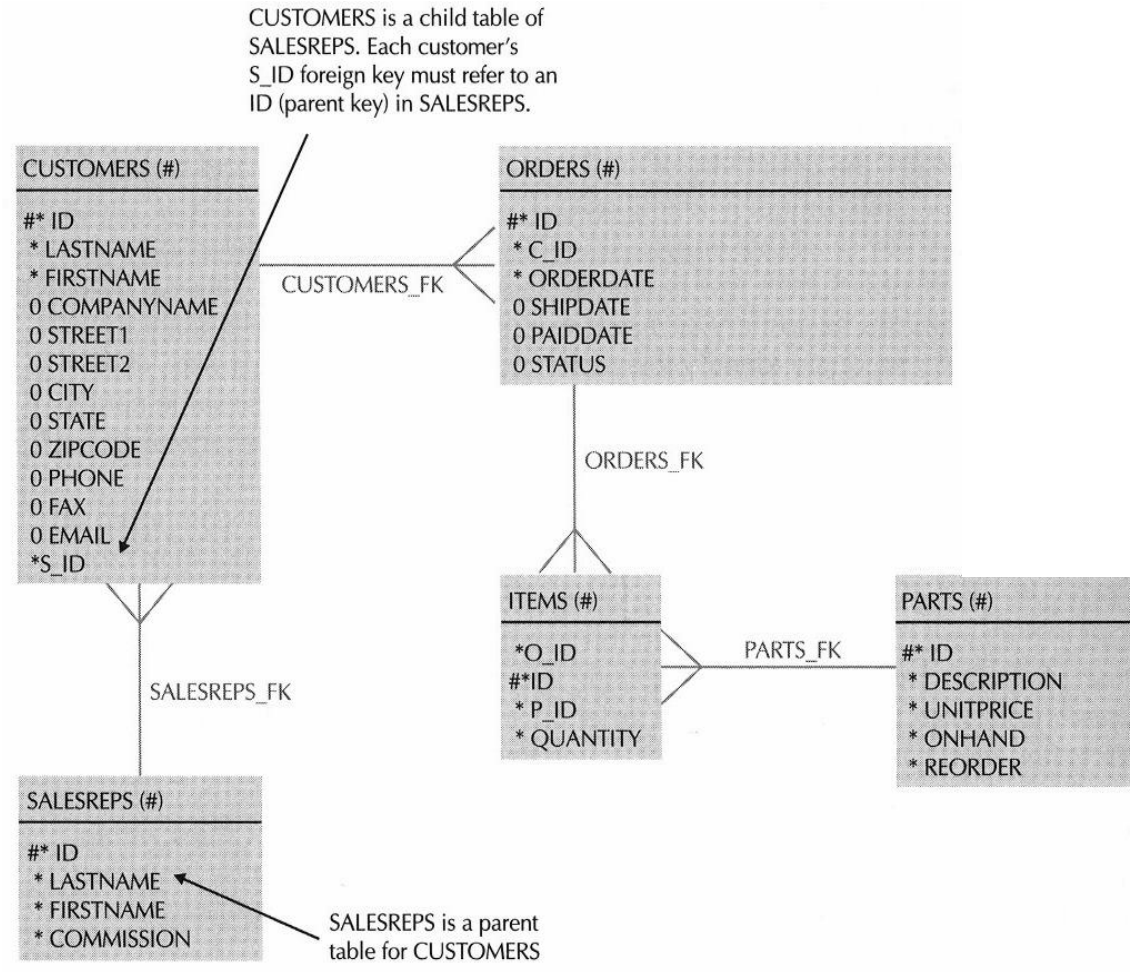

FIGURE 7-3. *Referential integrity describes the relationships among columns and tables in a relational database*

REFERENTIAL ACTIONS: Referential integrity ensures that each value in a foreign key always has a matching parent key value. To guarantee referential integrity, an RDBMS must also be able to address database operations that manipulate parent keys. For example, when a user deletes a sales order, what happens to the dependent line items for that order? Referential actions describe what will be done in cases where an application updates or deletes a parent key that has dependent child records.

The relational database model describes several referential actions:

- **Update/Delete Restrict** The RDBMS does not allow an application to update a parent key or delete a parent row that has one or more dependent child records. For example, you cannot delete a sales order from the ORDERS table if it has associated line items in the ITEMS table.
- **Delete Cascade** When an application deletes a row from the parent table, the RDBMS cascades the delete by deleting all dependent records in a child table. For example, when you delete an order from the ORDERS table,

the RDBMS automatically removes all corresponding line items from the ITEMS table.

- **Update Cascade** When an application updates a parent key, the RDBMS cascades the update to the dependent foreign keys. For example, when you change an order's ID in the ORDERS table, the RDBMS would automatically update the order ID of all corresponding line item records in the ITEMS table. This referential action is rarely useful, because applications typically do not allow users to update key values.
- **Update/Delete Set Null** When an application updates or deletes a parent key, all dependent keys are set to null.
- **Update/Delete Set Default** When an application updates or deletes a parent key, all dependent keys are set to a meaningful default value.

By default, Oracle enforces the Update/Delete Restrict referential actions for all referential integrity constraints. Optionally, Oracle can perform the Delete Cascade or Delete Set Null referential action for a referential integrity constraint.

7.4.4. When Does Oracle Enforce Integrity Constraint Rules?

Oracle can enforce an integrity constraint at two different times:

- By default, Oracle enforces all integrity constraints immediately after an application submits a SQL statement to insert, update, or delete rows in a table. When a statement causes a data integrity violation, Oracle automatically rolls back the effects of the statement.
- Optionally, Oracle can delay the enforcement of a deferrable integrity constraint until just before the commit of a transaction. When you commit a transaction and the transaction has modified table data such that it does not conform to all integrity constraints, Oracle automatically rolls back the entire transaction (that is, the effects of all statements in the transaction).

Typical database applications should choose to immediately check data integrity as each SQL statement is executed. However, certain applications, such as large data loads, might need to update many tables and temporarily violate integrity rules until just before the end of the transaction.

7.4.5. Creating and Managing Integrity Constraints

You can create integrity constraints for a table when you create the table, or subsequently by altering the table. The next few practice exercises teach you how to use the SQL commands CREATE TABLE and ALTER TABLE to create not null, check, primary key, unique, and referential integrity constraints.

EXERCISE 7.3: Creating a Table with Integrity Constraints

One way to declare integrity constraints for a table is to do so when you create the table. To create a table with integrity constraints, use the CONSTRAINT clause of the CREATE TABLE command. The following syntax listing is a partial listing of the options available with the CONSTRAINT clause of the CREATE TABLE command.

```
CREATE TABLE [schema.]table (
{ column datatype [DEFAULT expression]
 [CONSTRAINT constraint]
 { [NOT] NULL
 / (UNIQUE|PRIMARY KEY)
 / REFERENCES [schema.]table [(column)] [ON DELETE CASCADE]
 / CHECK (condition) }}
 / another constraint specification }}
/  [CONSTRAINT constraint]
 { {UNIQUE|PRIMARY KEY) (column [, column] ... )
 / FOREIGN KEY (column [, column] ... )
REFERENCES [schema.]table [(column [, column] ... )]
 [ON DELETE {CASCADEISET NULL}]
 / CHECK (condition) ]
[ ,... other columns/constraints or constraints ]
```

Notice that you can declare an integrity constraint along with a column, or you can declare an integrity constraint separate from a specific column declaration. In general, you can always choose either option to create a constraint, except in the following situations:

- To declare a column with a not null constraint, you must do so as part of the column declaration.
- To declare a composite primary key, unique, or referential integrity constraint, you must declare the constraint separate from a specific column declaration.

Enter the following CREATE TABLE statement to create the familiar ORDERS table with some integrity constraints.

```
CREATE TABLE orders (
 id INTEGER
 CONSTRAINT orders_pk PRIMARY KEY,
 orderdate DATE DEFAULT SYSDATE
 NOT NULL,
 shipdate DATE,
 paiddate DATE,
 status CHAR(1) DEFAULT 'F'
 CONSTRAINT status_ck
 CHECK (status IN ('F','B'))
);
```

This statement creates the ORDERS table and declares three integrity constraints as part of column declarations (see the bold CONSTRAINT clauses above).

- The ID column is the ORDERS table's primary key--every record must have an ID (a null is implicitly disallowed) that is unique from all others. The statement names the primary key constraint ORDERS_PK.
- The statement declares the ORDERDATE column as not null. Because the statement does not explicitly name the not null constraint, Oracle generates a unique name for the constraint. Later in this chapter, you'll learn how to reveal information about schema objects and integrity constraints, including generated constraint names.
- The STATUS_CK check constraint ensures that the STATUS field value is F or B for every record in the ORDERS table. Because the statement does not declare the STATUS column with a not null constraint, the STATUS column can also contain nulls.

The next section provides more examples of integrity constraint declarations with the ALTER TABLE command, including how to declare unique and referential integrity constraints.

EXERCISE 7.4: Adding a Not Null Constraint to an Existing Column

After you create a table, you might need to add (or remove) a not null integrity constraint to (or from) an existing column you can do so by using the following syntax of the ALTER TABLE command:

```
ALTER TABLE [schema.] table
MODIFY column [NOT] NULL
```

For example, enter the following command to add a not null integrity constraint to the DESCRIPTION column of the PARTS table:

```
ALTER TABLE parts
MODIFY description NOT NULL;
```

NOTE
To subsequently remove the not null constraint from the STATUS column, you would use the previous statement but omit the NOT keyword.

EXERCISE 7.5: Adding Primary Key and Unique Constraints to a Table

You can also declare an integrity constraint after you create a table using the ALTER TABLE command, as follows:

```
ALTER TABLE [schema.] table
ADD [CONSTRAINT constraint]
 {{UNIQUE|PRIMARY KEY} (column [, column] ... )
 / FOREIGN KEY (column [, column] ... )
    REFERENCES [schema.]table [(column [, column] ... )]
    [ON DELETE {CASCADE|SET NULL}]
/CHECK (condition) }
```

For example, the PARTS and CUSTOMERS tables that you created in Exercise 7.1 do not have primary keys--enter the following commands to add primary key constraints for these tables.

```
ALTER TABLE parts
ADD CONSTRAINT parts_pk PRIMARY KEY (id);

ALTER TABLE customers
ADD CONSTRAINT customers_pk PRIMARY KEY (id);
```

Enter the following command to add a composite unique constraint to the CUSTOMERS table that prevents duplicate LASTNAME/FIRSTNAME combinations. Because the statement does not explicitly name the unique constraint, Oracle generates a unique system identifier for the new constraint.

```
ALTER TABLE customers
ADD UNIQUE (lastname, firstname);
```

EXERCISE 7.6: Adding Referential Constraints to a Table

You can also use the syntax of the ALTER TABLE command in the previous exercise to add a referential integrity constraint to a table. For example, enter the following statement to add a referential integrity constraint to the CUSTOMERS table that ensures each customer record's SID refers to an ID in the SALESREPS table.

```
ALTER TABLE customers
  ADD CONSTRAINT salesreps_fk
    FOREIGN KEY (s_id) REFERENCES salesreps (id);
```

The previous statement should return the following error number and message:

```
ORA 02270: no matching unique or primary key for this column-list
```

Why? Remember that when you declare a referential integrity constraint for a table, the foreign key must refer to a primary key or unique key in a table. Because the SALESREPS table does not have a primary key, the preceding statement returns an error.

To remedy this situation, first add the primary key to the SALESREPS table, and then reissue the previous statement to add the referential integrity constraint to the CUSTOMERS table.

```
ALTER TABLE salesreps
 ADD CONSTRAINT salesreps_pk PRIMARY KEY (id);

ALTER TABLE customers
 ADD CONSTRAINT salesreps_fk
  FOREIGN KEY (s_id) REFERENCES salesreps (id);
```

Notice that the specification of the SALESREPS_FK referential integrity constraint does not specify a referential action for deletes. By this omission, the referential integrity constraint enforces the delete restrict referential action. A subsequent exercise in this chapter demonstrates how to declare a referential integrity constraint with the delete cascade referential action.

EXERCISE 7.7: Adding a Column with Constraints

When you add a column to a table, you can also add a constraint to the table at the same time, using the following syntax of the ALTER TABLE command:

```
ALTER TABLE [schema.]table
 ADD (
  column [datatype] [DEFAULT expression]
  [ [CONSTRAINT constraint]
  { NOT NULL
  / {UNIQUEIPRIMARY KEY}
  / REFERENCES table [(column)]
      [ON DELETE {CASCADE|SET NULL}]
  / CHECK (condition) } ]
  [ another constraint specification ]
  [,other columns and their constraints...]
)
```

For example, each record in the ORDERS table needs a field to keep track of the ID of the customer that places the order. To add this column, enter the following statement, which adds the C_ID column to the ORDERS table, along with a not null and referential integrity constraint.

```
ALTER TABLE orders
 ADD c_id INTEGER
  CONSTRAINT c_id_nn NOT NULL
  CONSTRAINT customers_fk
REFERENCES customers (id);
```

The combination of the not null and referential integrity constraints in this exercise ensure that each record in the ORDERS table must have a C_ID (customer ID) that refers to an ID in the CUSTOMERS table.

EXERCISE 7.8: Declaring a Referential Constraint with a Delete Action

We need one more table to complete the table specifications in our very simple practice schema enter the following CREATE TABLE statement, which builds the ITEMS table, along with several integrity constraints (highlighted in bold).

```
CREATE TABLE items(
 o_id INTEGER
  CONSTRAINT orders_fk
  REFERENCES orders ON DELETE CASCADE,
 id INTEGER,
```

```
 p_id INTEGER
  CONSTRAINT parts_fk
  REFERENCES parts,
quantity  INTEGER DEFAULT 1
CONSTRAINT quantity_nn NOT NULL,
CONSTRAINT items_pk
PRIMARY KEY (o_id, id)
);
```

The following list describes the integrity constraints declared for the ITEMS table.

- The ORDERS_FK referential integrity constraint ensures that the O_ID field of each record in the ITEMS table refers to an ID in the ORDERS table. This referential integrity constraint also specifies the delete cascade referential action whenever a transaction deletes a record in the ORDERS table, Oracle will automatically cascade the delete by deleting the associated records in the ITEMS table.
- The QUANTITY_NN not null constraint prevents nulls from being entered in the QUANTITY column.
- The PARTS_FK referential integrity constraint ensures that the P_ID field of each record in the ITEMS table refers to an ID in the PARTS table.
- The ITEMS_PK primary key constraint is a composite primary key. This constraint ensures that neither the O_ID nor ID columns contain nulls, and that each record's O_ID/ID combination is unique from all others in the ITEMS table.

EXERCISE 7.9: Testing an Integrity Constraint

At this point, we've got all of our tables built. The statements in this exercise have you insert some rows into various tables to confirm that the integrity constraints we created in the previous exercises actually enforce our business rules.

First, observe what happens when you enter the following statements, which insert three new sales representatives into the SALESREPS table.

```
INSERT INTO salesreps
(id, lastname, firstname, commission)
VALUES (1,'Pratt','Nick',5);

INSERT INTO salesreps
(id, lastname, firstname, commission)
VALUES (2, 'Jonah','Suzanne',5);

INSERT INTO salesreps
(id, lastname, firstname, commission)
VALUES (2,'Greenberg','Bara',5);
```

The first and second INSERT statements should execute without error. However, when you attempt the third INSERT statement, Oracle will return the following error number and message:

```
ORA 00001: unique constraint (PRACTICE07.SALESREPS PK) violated
```

The primary key constraint of the SALESREPS table prohibits two records from having the same ID. In this example, the third INSERT statement attempts to insert a new record with an ID number of 2, which is already in use by another record. If you rewrite the third INSERT statement with a different ID, the row will insert without error.

```
INSERT INTO salesreps
(id, lastname, firstname, commission)
VALUES (3,'Greenberg','Bara',5);
```

You can permanently commit your current transaction by issuing a COMMIT statement.

```
COMMIT;
```

Now, let's test a referential integrity constraint and see what happens. Enter the following statements, which insert some records into the CUSTOMERS table.

```
INSERT INTO customers
( id, lastname, firstname, companyname, street,
city, state, zipcode, phone, fax, email, s_id)
VALUES (1,'Joy','Harold','McDonald Co.',
'4458 Stafford St.','Baltimore','MD','21209',
'410 983 5789',NULL,'harold_joy@mcdonald.com',3);

INSERT INTO customers
( id, lastname, firstname, companyname,
street, city, state, zipcode, phone, fax, email, s_id)
VALUES (2,'Musial','Bill','Car Audio Center',
'12 Donna Lane','Reno','NV','89501','775 859 2121',
'775 859 2121','musial@car audio.net',5);
```

The first INSERT statement should execute without error, provided that you successfully executed the previous INSERT statement in this exercise (inserting the record for the sales representative with an ID of 3). However, when you attempt to execute the second INSERT statement, Oracle will return the following error number and message.

```
 ORA 02291: integrity constraint (PRACTICE07.SALESREPS_FK)
violated
 - parent key not found
```

The SALESREPS_FK referential integrity constraint in the CUSTOMERS table does not permit a customer record with an S_ID that fails to match an ID in the SALESREPS table. In this case, the INSERT statement attempts to insert a record that

refers to a sales representative with an ID of 5, which does not exist. The following rewrite of the second INSERT statement should succeed without error:

```
INSERT INTO customers
(id, lastname, firstname, companyname, street,
city, state, zipcode, phone, fax, email, s_id)
VALUES (2,'Musial','Bil1','Car Audio Center',
'12 Donna Lane','Reno','NV','89501','775-859-2121',
'775-859-2121','musial@car-audio.net',1);

COMMIT;
```

EXERCISE 7.10: Declaring and Using a Deferrable Constraint

All of the constraints that you specified in Exercises 7.3 through 7.8 are immediately enforced as each SQL statement is executed. The previous exercise demonstrates this immediate constraint enforcement when you attempt to insert a row that does not have a unique primary key value into the PARTS table, Oracle immediately enforces the constraint by rolling back the INSERT statement and returning an error.

You can also create a deferrable constraint, if your application logic requires. If you do so, upon beginning a new transaction, you can instruct Oracle to defer the enforcement of selected or all deferrable constraints until you commit the transaction. To create a deferrable constraint, include the optional keyword DEFERRABLE when you specify the constraint.

To demonstrate deferrable constraints, let's make the STATUS _CK check constraint of the ORDERS table a deferrable constraint. This would permit a sales representative to defer the decision as to whether a new order should be backordered because of a lack of inventory for a particular part being ordered. First, you have to drop the existing check constraint, as follows:

```
ALTER TABLE orders
DROP CONSTRAINT status_ck;
```

Next, enter the following command to recreate the STATUS CK check constraint as a deferrable constraint:

```
ALTER TABLE orders
ADD CONSTRAINT status_ck
CHECK (status IN ('F','B')) DEFERRABLE;
```

Now let's test the deferrable constraint. To defer the enforcement of a deferrable constraint, you start a transaction with the SQL command SET CONSTRAINTS, which has the following syntax:

```
SET CONSTRAINT[S]
{  [schema.]constraint [,[schema.]constraint] ...
```

```
 |ALL }
{  IMMEDIATE | DEFERRED }
```

Notice that the SET CONSTRAINTS command lets you explicitly set the enforcement of specific or all constraints. To defer the enforcement of our STATUS CK constraint, start the new transaction with the following statement:

```
SET CONSTRAINTS status_ck DEFERRED;
```

Next, enter the following statement to insert a record into the ORDERS table that does not meet the condition of the STATUS_CK check constraint the STATUS code for the order is "U" rather than "B" or "F".

```
INSERT INTO orders
 (id, c_id, orderdate, shipdate, paiddate, status)
VALUES (1,1,'18 JUN 99','18 JUN 99','30 JUN 99','U');
```

Now, commit the transaction with a COMMIT statement, to see what happens. Oracle should return the following error messages:

```
COMMIT;

ORA 02091: transaction rolled back
ORA 02290: check constraint (PRACTICE07.STATUS_CK) violated
```

When you commit the transaction, Oracle enforces the rule of the STATUS CK deferrable constraint and notices that the new row in the ORDERS table does not comply with the associated business rule. Therefore, Oracle rolls back all of the statements in the current transaction.

7.5. Views

Once you define the tables in a database, you can start to focus on other things that enhance the usability of the application schema. You can start by defining views of the tables in your schema. A view is a database object that presents table data. Why and how would you use views to present table data?

- You can use a simple view to expose all rows and columns in a table, but hide the name of the underlying table for security purposes. For example, you might create a view called CUST that presents all customer records in the CUSTOMERS table.
- You can use a view to protect the security of specific table data by exposing only a subset of the rows and/or columns in a table. For example, you might create a view called CUST_CA that presents only the LASTNAME, FIRSTNAME, and PHONE columns in the CUSTOMERS table for customers that reside in the state of California.
- You can use a view to simplify application coding. A complex view might join the data of related parent and child tables to make it appear as though a

different table exists in the database. For example, you might create a view called ORDER ITEMS that joins related records in the ORDERS and ITEMS tables.

- You can use a view to present derived data that is not actually stored in a table. For example, you might create a view of the ITEMS table with a column called TOTAL that calculates the line total for each record.

As you can see from this list, views provide a flexible means of presenting the table data in a database. In fact, you can create a view of any data that you can represent with a SQL query. That's because a view is really just a query that Oracle stores as a schema object. When an application uses a view to do something, Oracle derives the data of the view based on the view's defining query. For example, when an application queries the CUST_CA view described in the previous list, Oracle processes the query against the data described by the view's defining query.

7.5.1. Creating Views

To create a view, you use the SQL command CREATE VIEW. The following is an abbreviated syntax listing of the CREATE VIEW command:

```
CREATE [OR REPLACE] VIEW [schema.]view
 AS subquery
 [WITH READ ONLY]
```

The next few sections and practice exercises explain more about the specific types of views that Oracle supports, and provide you with examples of using the various clauses, parameters, and options of the CREATE VIEW command.

7.5.2. Read Only Views

One type of view that Oracle supports is a read only view. As you might expect, database applications can use a read only view to retrieve corresponding table data, but cannot insert, update, or delete table data through a read only view.

EXERCISE 7.11: Creating a Read Only View

Enter the following statement to create a read only view of the ORDERS table that corresponds to the orders that are currently on backlog.

```
CREATE VIEW backlogged_Orders
AS SELECT * FROM orders
WHERE status = 'B'
WITH READ ONLY;
```

Notice the following points about this first example of the CREATE VIEW command.

- The AS clause of the CREATE VIEW command specifies the view's defining query. The result set of a view's defining query determines the view's structure (columns and rows).
- To create a read only view, you must specify the WITH READ ONLY option of the CREATE VIEW command to explicitly declare that the view is read only; otherwise, Oracle creates the view as an updateable view.

7.5.3. Updateable Views

Oracle also allows you to define updateable views that an application can use to insert, update, and delete table data or query data.

EXERCISE 7.12: Creating an Updateable View

To create a view as an updateable view, simply omit the WITH READ ONLY option of the CREATE VIEW command when you create the view. For example, enter the following CREATE VIEW statement, which creates an updateable join view of the ORDERS and PARTS tables.

```
 CREATE VIEW orders_items
AS
 SELECT o.id AS orderid,
  o.orderdate AS orderdate,
  o.c_id AS customerid,
  i.id AS itemid,
  i.quantity AS quantity,
  i.p_id AS partid
 FROM orders o, items i
WHERE o.id = i.o_id;
```

Even though you declare a view as updateable, Oracle doesn't automatically support INSERT, UPDATE, and DELETE statements for the view unless the view's definition complies with the materialized view principle. Briefly stated, the materialized view principle ensures that the server can correctly map an insert, update, or delete operation through a view to the underlying table data of the view.

The ORDERS ITEMS view is an example of a view that does not comply with the materialized view principle because the view joins data from two tables. Therefore, even though you created the ORDERS ITEMS view as updateable, Oracle does not support INSERT, UPDATE, and DELETE statements with the view until you create one or more INSTEAD OF triggers for the updateable view.

7.5.4. INSTEAD OF Triggers and Updateable Views

Even when a view's attributes violate the materialized view principle, you can make the view updateable if you define INSTEAD OF triggers for the view. An INSTEAD OF trigger is a special type of row trigger that you define for a view. An

INSTEAD OF trigger explains what should happen when INSERT, UPDATE, or DELETE statements target the view that would otherwise not be updateable.

EXERCISE 7.13: Creating an INSTEAD OF Trigger for an Updateable View

To create an INSTEAD OF trigger, use the following syntax of the CREATE TRIGGER command:

```
CREATE [OR REPLACE] TRIGGER trigger
 INSTEAD OF
 {DELETE|INSERT|UPDATE [OF column [,column] ... ]}
 [OR {DELETE|INSERT|UPDATE [OF column [,column] ... ]} ] ...
 ON table/view }
 ... PL/SQL block ...
END [trigger]
```

For example, enter the following statement, which creates an INSTEAD OF trigger that defines the logic for handling an INSERT statement that targets the ORDERS_ITEMS view.

```
CREATE OR REPLACE TRIGGER orders_items_insert
  INSTEAD OF INSERT ON orders_items
   DECLARE
   currentOrderId INTEGER;
   currentOrderDate DATE;
 BEGIN
 -- Determine if the order already exists.
 SELECT id, orderdate
 INTO currentOrderId, currentOrderDate
 FROM orders
 WHERE id = :new.orderid;
  -- If the NO DATA FOUND exception is not raised,
  -- insert a new item into the ITEMS table.
 INSERT INTO items
 (o_id, id, quantity, p_id)
 VALUES (:new.orderid, :new.itemid,
 :new.quantity, :new.partid);
EXCEPTION
WHEN no_data_found THEN
 INSERT INTO orders
  (id, orderdate, c_id)
 VALUES (:new.orderid, :new.orderdate,
 :new.customerid);
 INSERT INTO items
   (o_id, id, quantity, p_id)
  VALUES (:new.orderid, :new.itemid,
   :new.quantity, :new.partid);
END orders_items_insert;
/
```

Now, when an INSERT statement targets the ORDER_ITEMS view, Oracle will translate the statement using the logic of the ORDERS_ITEMS_INSERT trigger to insert

rows into the underlying ORDERS and ITEMS tables. For example, enter the following INSERT statement:

```
INSERT INTO orders_items
( orderid, orderdate, customerid, itemid, quantity)
VALUES (1, '18-JUN-99', 1, 1, 1);
```

Now, query the ORDERS and ITEMS tables to see that the trigger worked as planned.

```
SELECT * FROM orders;

ID     ORDERDATE     SHIPDATE    PAIDDATE   S     C_ID
--     ---------     --------    --------   --    ----
1      18 JUN 99                            F       1

SELECT * FROM items;

O_ID   ID P_ID QUANTITY
------  ----- ------ -------------
1      1            1
```

7.6. Sequences

An OLTP (On-Line Transaction Processing) application, such as an airline reservation system, typically supports a large number of concurrent users. As each user's transaction inserts one or more new rows into various database tables, coordinating the generation of unique primary keys among multiple, concurrent transactions can be a significant challenge for the application.

Fortunately, Oracle has a feature that makes the generation of unique values a trivial matter. A sequence is a schema object that generates a series of unique integers, and is appropriate only for tables that use simple, numerical columns as keys, such as the ID columns used in all tables of our practice schema. When an application inserts a new row into a table, the application simply requests a database sequence to provide the next available value in the sequence for the new row's primary key value. What's more, the application can subsequently reuse a generated sequence number to coordinate the foreign key values in related child rows. Oracle manages sequence generation with an insignificant amount of overhead, allowing even the most demanding of online transaction processing (OLTP) applications to perform well.

7.6.1. Creating and Managing Sequences

To create a sequence, you use the SQL command CREATE SEQUENCE.

```
CREATE SEQUENCE [schema.]sequence
 [START WITH integer]
 [INCREMENT BY integer]
```

```
[MAXVALUE integer|NOMAXVALUE]
[MINVALUE integer|NOMINVALUE]
[CYCLE|NOCYCLE]
[CACHE integer|NOCACHE]
[ORDER|NOORDER]
```

Notice that when you create a sequence, you can customize it to suit an application's particular needs; for example, an Oracle sequence can ascend or descend by one or more integers, have a maximum or minimum value, and more.

If need be, you can subsequently alter the properties of a sequence using the SQL command ALTER SEQUENCE. The ALTER SEQUENCE command supports the same options and parameters as the CREATE SEQUENCE command, with the exception of the START WITH parameter.

EXERCISE 7.14: Creating a Sequence

Enter the following CREATE SEQUENCE statement to create a sequence for sales order IDs.

```
CREATE SEQUENCE order_ids
START WITH 2
INCREMENT BY 1
NOMAXVALUE;
```

The ORDER_IDS sequence starts with the integer 2 (remember, we already have a record in the ORDERS table with an ID set to 1), increments every sequence generation by 1, and has no maximum value.

EXERCISE 7.15: Using and Reusing a Sequence Number

To generate a new sequence number for your user session, a SQL statement must reference the sequence and its NEXTVAL pseudocolumn. Enter the following INSERT statement to insert a new sales order and use the ORDER IDS sequence to generate a unique order ID.

NOTE
A pseudocolumn is similar to a column in a table. SQL statements can reference pseudocolumns to retrieve data, but cannot insert, update, or delete data by referencing a pseudocolumn.

```
INSERT INTO orders
(id, c_id, orderdate, status)
VALUES (order_ids.NEXTVAL,2,'18-JUN-99','B');
```

NOTE
Once your session generates a new sequence number, only your session can reuse the sequence number--other sessions generating sequence

numbers with the same sequence receive subsequent sequence numbers of their own.

To reuse the current sequence number assigned to your session, a SQL statement must reference the sequence and its CURRVAL pseudocolumn. Using the CURRVAL pseudocolumn, your session can reuse the current sequence number any number of times, even after a transaction commits or rolls back. For example, enter the following INSERT statements to insert several new line items into the ITEMS table for the current order, and then commit the transaction.

```
 INSERT INTO items
(o_id, id, quantity)
VALUES (order_ids.CURRVAL,1,1);

INSERT INTO items
(o_id, id, quantity)
VALUES (order_ids.CURRVAL,2,4);

INSERT INTO items
(o_id, id, quantity)
VALUES (order_ids.CURRVAL,3,5);

COMMIT;
```

7.7. Synonyms

When developers build a database application, it's prudent to avoid having application logic directly reference tables, views, and other database objects. Otherwise, applications must be updated and recompiled after an administrator makes a simple modification to an object, such as a name change or structural change.

To help make applications less dependent on database objects, you can create synonyms for database objects. A *synonym* is an alias for a table, view, sequence, or other schema object that you store in the database. Because a synonym is just an alternate name for an object, it requires no storage other than its definition. When an application uses a synonym, Oracle forwards the request to the synonym's underlying base object.

7.7.1. Private and Public Synonyms

Oracle allows you to create both public and private synonyms. A public synonym is an object alias (another name) that is available to every user in a database. A private synonym is a synonym within the schema of a specific user who has control over its use by others.

7.7.2. Creating Synonyms

To create a synonym, use the SQL command *CREATE SYNONYM*.

```
CREATE [PUBLIC] SYNONYM [schema.]synonym
 FOR [schema.]object
```

If you include the optional PUBLIC keyword, Oracle creates a synonym as a public synonym; otherwise, Oracle creates a synonym as a private synonym.

EXERCISE 7.16: Creating a Synonym

Enter the following statement to create the private synonym CUST in the current schema. The private synonym is an alias for the CUSTOMERS table in the same schema.

```
CREATE SYNONYM cust
FOR customers;
```

Next, enter the following statement to create a public synonym for the SALESREPS table of the current schema.

```
CREATE PUBLIC SYNONYM salespeople
FOR salesreps;
```

EXERCISE 7.17: Using a Synonym

The use of a synonym is transparent just reference the synonym anywhere you would its underlying object. For example, enter the following query that uses the new CUST synonym.

```
SELECT id, lastname FROM cust;
```

The result set is as follows:

```
 ID     LASTNAME
--     ------------
1         Joy

2         Musial
```

7.8. Indexes

The performance of an application is always critical. That's because the productivity of an application user directly relates to the amount of time that the user must sit idle while the application tries to complete work. With database applications, performance depends greatly on how quickly an application can access table data. Typically, disk I/O is the primary performance determining factor for table access the less disk I/O that's necessary to access table data, the better the dependent applications will perform. In general, it's best to try to minimize the amount of disk access that applications must perform when working with database tables.

NOTE
This section introduces indexes to support subsequent sections of this book. For complete information about the various types of indexes that Oracle supports and other performance related topics, see Chapter 12.

The judicious use of table indexes is the principal method of reducing disk I/O and improving the performance of table access. Just like an index in a book, an index of a table column (or set of columns) allows Oracle to quickly find specific table records. When an application queries a table and uses an indexed column in its selection criteria, Oracle automatically uses the index to quickly find the target rows with minimal disk I/O. Without an index, Oracle has to read the entire table from disk to locate rows that match a selection criteria. The presence of an index for a table is entirely optional and transparent to users and developers of database applications. For example:

- Applications can access table data with or without associated indexes.
- When an index is present and it will help the performance of an application request, Oracle automatically uses the index; otherwise, Oracle ignores the index.
- Oracle automatically updates an index to keep it in synch with its table.

Although indexes can dramatically improve the performance of application requests, it's unwise to index every column in a table. Indexes are meaningful only for the key columns that application requests specifically use to find rows of interest. Furthermore, index maintenance generates overhead--unnecessary indexes can actually slow down your system rather than improve its performance.

Oracle supports several different types of indexes to satisfy many types of application requirements. The most frequently used type of index in an Oracle database is a B tree index, sometimes referred to a normal index in the Oracle documentation set. The following sections explain more about B tree indexes, which you can create for a table's columns.

7.8.1. B Tree Indexes

The default and most common type of index for a table column is a B tree index. A B tree index, or normal index, is an ordered tree of index nodes, each of which contains one or more index entries. Each index entry corresponds to a row in the table, and contains two elements:

- The indexed column value (or set of values) for the row
- The ROWID (or physical disk location) of the row

A B-tree index contains an entry for every row in the table, unless the index entry for a row is null. Figure 7-4 illustrates a typical B-tree index.

When using a B tree index, Oracle descends the tree of index nodes looking for index values that match the selection criteria of the query. When it finds a match, Oracle

uses the corresponding ROWID to locate and read the associated table row data from disk.

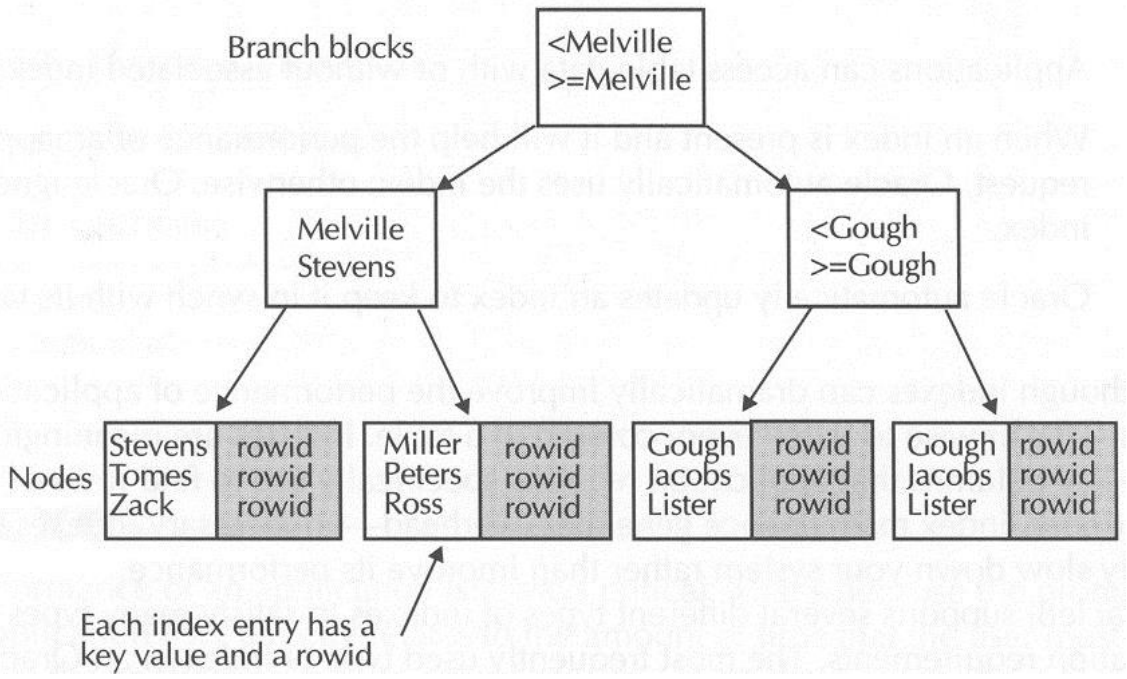

FIGURE 7-4. *A B-tree index*

7.8.2. Using B- Tree Indexes Appropriately

B tree indexes are not appropriate for all types of applications and all types of columns in a table. In general, B tree indexes are the best choice for OLTP applications where data is constantly being inserted, updated, and deleted. In such environments, B tree indexes work best for key columns that contain many distinct values relative to the total number of key values in the column. The primary and alternate keys in a table are perfect examples of columns that should have B tree indexes. Conveniently, Oracle automatically creates B tree indexes for all primary key and unique integrity constraints of a table.

7.8.3. Creating B Tree Indexes

To create an index, you use the SQL command CREATE INDEX. The following is an abbreviated version of the syntax listing for the CREATE INDEX command that focuses solely on the parts of that command that pertain to B tree (normal) indexes.

```
CREATE [UNIQUE] INDEX [schema.]index
 ON { [schema.] table
 ( column [ASC|DESC] [, column [ASCIDESC]] ... )
```

Notice the following points about the CREATE INDEX command:

- By including the optional keyword UNIQUE, you can prevent duplicate values in the index. However, rather than creating a unique index, Oracle Corp. recommends that you declare a unique constraint for a table so that the integrity constraint is visible along with other integrity constraints in the database.
- You must specify one or more columns to be indexed. For each index, you can specify that you want the index to store values in ascending or descending order.

  NOTE
  Versions of Oracle previous to Oracle8i supported the DESC keyword when creating a B tree index, but always created ascending indexes. Oracle now supports descending indexes.

EXERCISE 7.18: Creating a B-Tree Index

To facilitate faster joins between the ITEMS and PARTS tables, enter the following command, which creates a B tree index for the columns in the PARTS_FK foreign key column of the ITEMS table.

```
CREATE INDEX items_p_id
ON items (p_id ASC);
```

7.9. The Data Dictionary: A Unique Schema

Every Oracle database uses a number of system tables and views to keep track of *metadata*--data about the data in a database. This collection of system objects is called the Oracle database's *data dictionary* or *system catalog*. Oracle organizes a database's data dictionary within the SYS schema.

As you create and manage schemas in an Oracle database, you can reveal information about associated schema objects by querying the tables and views of the data dictionary. For example, Table 7-3 provides a list of the several data dictionary views that correspond to the schema objects introduced in this chapter.

Type of Schema Object	**Data Dictionary Views of Interest**
Tables and Columns	DBA_TABLES, ALL_TABLES, and USER_TABLES display general information about database tables. DBA_TAB_COLUMNS, ALL _TAB_ COLUMNS, and USER _TAB _COLUMNS display information about the columns in each database table. ***NOTE:*** *DBA_OBJECTS, ALL_OBJECTS, and USER_ OBJECTS display information about schema objects, including tables.*
Integrity Constraints	DBA_CONSTRAINTS, ALL _CONSTRAINTS, and USER _CONSTRAINTS display general information

	about constraints. DBA _CONS _COLUMNS, ALL _CONS _COLUMNS, and USER _CONS _COLUMNS display information about columns and associated constraints. Views, DBA _VIEWS, ALL_ VIEWS, and USER _VIEWS. **NOTE:** *DBA_OBJECTS, ALL_OBJECTS, and USER_OBJECTS also display information about schema objects, including views.*
Sequences	DBA_SEQUENCES, ALL_SEQUENCES, and USER_SEQUENCES. ***NOTE:*** *DBA_OBJECTS, ALL _OBJECTS, and USER _OBJECTS display information about schema objects, including sequences.*
Synonyms	DBA _SYNONYMS, ALL _SYNONYMS, and USER _SYNONYMS. ***NOTE:*** *DBA _OBJECTS, ALL _OBJECTS, and USER _OBJECTS display information about schema objects, including synonyms.*
Indexes	DBA_INDEXES, ALL _INDEXES, USER _INDEXES, DBA_IND COLUMNS, ALL_IND_COLUMNS, and USER _IND _COLUMNS.

TABLE 7-3. *The Data Dictionary Views that Correspond to Tables, Columns, Constraints, Views, Sequences, Synonyms, and Indexes*

Categories of Data Dictionary Views

Oracle's data dictionary contains several different categories of data dictionary views:

- Views that begin with the prefix "DBA " present all information in the corresponding data dictionary base tables. Because the DBA views are comprehensive, they are accessible only to users that have the SELECT ANY TABLE system privilege. (See Chapter 9 for more information about privileges and database security.)
- Views that begin with the prefix "ALL " are available to all users and show things specific to the privilege domain of the current user.
- Views that begin with the prefix 'USER " are available to all users and show things specific to the current user.

EXERCISE 7.19: Querying the Data Dictionary

In this final practice exercise of this chapter, let's query the data dictionary to reveal information about the integrity constraints created in the previous exercises in this chapter. For this query, we need to target the USER_CONSTRAINTS data dictionary

view. First, enter the following DESCRIBE command to reveal the columns available in the USER_CONSTRAINTS view.

```
DESCRIBE user_constraints;

Name                 Null?                Type
-------------------- -------------        --------------------
OWNER                NOT NULL             VARCHAR2(30)
CONSTRAINT_NAME      NOT NULL             VARCHAR2(30)
CONSTRAINT_TYPE                           VARCHAR2(1)
TABLE_NAME           NOT NULL             VARCHAR2(30)
SEARCH_CONDITION                          LONG
R_OWNER                                   VARCHAR2(30)
R_CONSTRAINT_NAME                         VARCHAR2(30)
DELETE_RULE                               VARCHAR2(9)
STATUS                                    VARCHAR2(8)
DEFERRABLE                                VARCHAR2(14)
DEFERRED                                  VARCHAR2(9)
VALIDATED                                 VARCHAR2(13)
GENERATED                                 VARCHAR2(14)
BAD                                       VARCHAR2(3)
RELY                                      VARCHAR2(4)
LAST_CHANGE                               DATE
```

As you can see, the USER_CONSTRAINTS view contains many columns for recording the properties of integrity constraints. For this exercise, enter the following query to display the name of each constraint, the table that it is associated with, the type of constraint, and whether the constraint is deferrable.

```
SELECT constraint_name, table_name,
 DECODE(constraint_type,
   'C', 'CHECK',
   'P', 'PRIMARY KEY',
   'U','UNIQUE',
   'R','REFERENTIAL',
   '0','VIEW WITH READ ONLY',
   'OTHER') constraint_type,
   deferrable
 FROM user_constraints;
```

If you completed the previous exercises in this chapter, the result set of the query should be as follows:

```
CONSTRAINT NAME    TABLE NAME        CONSTRAINT TYPE       DEFERRABLE
---------------    -------------     -----------------     -------------
SYS_C002892        BACKLOGGED_ORDERS VIEW WITH READ ONLY   NOT DEFERRABLE
CUSTOMERS_PK       CUSTOMERS         PRIMARY KEY           NOT DEFERRABLE
SYS_CO02882        CUSTOMERS         UNIQUE                NOT DEFERRABLE
SALESREPS_FK       CUSTOMERS         REFERENTIAL           NOT DEFERRABLE
QUANTITY_NN        ITEMS             CHECK                 NOT DEFERRABLE
ITEMS_PK           ITEMS             PRIMARY KEY           NOT DEFERRABLE
ORDERS_FK          ITEMS             REFERENTIAL           NOT DEFERRABLE
PARTS_FK           ITEMS             REFERENTIAL           NOT DEFERRABLE
SYS_C002876        ORDERS            CHECK                 NOT DEFERRABLE
```

```
STATUS_CK          ORDERS           CHECK                DEFERRABLE
ORDERS_PK          ORDERS           PRIMARY KEY          NOT DEFERRABLE
C_ID_NN            ORDERS           CHECK                NOT DEFERRABLE
CUSTOMERS_FK       ORDERS           REFERENTIAL          NOT DEFERRABLE
SYS_C002879        PARTS            CHECK                NOT DEFERRABLE
PARTS_PK           PARTS            PRIMARY KEY          NOT DEFERRABLE
SALESREPS_PK       SALESREPS        PRIMARY KEY          NOT DEFERRABLE


16 rows selected.
```

Notice that Oracle generated unique names starting with the prefix "SYS_ " for all of the constraints that you did not explicitly name. The DECODE expression in the query's SELECT clause translates codes in the CONSTRAINT_TYPE column to readable information.

Chapter Summary

This chapter has introduced many different types of objects that you can create in a basic relational database schema.

- Tables are the basic data structure in any relational database. A table is nothing more than an organized collection of rows that all have the same columns. A column's datatype describes the basic type of data that is acceptable in the column. To create and alter a table's structure, you use the SQL commands CREATE TABLE and ALTER TABLE.
- To enforce business rules that describe the acceptable data for columns in a table, you can declare integrity constraints along with a table. You can use domain integrity constraints, such as not null constraints and check constraints, to explicitly define the domain of acceptable values for a column. You can use entity integrity constraints, such as primary key and unique constraints, to prevent duplicate rows in a table. And finally, you can use referential integrity constraints to establish and enforce the relationships among different columns and tables in a database. You can declare all types of integrity constraints when you create a table with the CREATE TABLE command, or after table creation with the ALTER TABLE command.
- A view is a schema object that presents data from one or more tables. A view is nothing more than a query that Oracle stores in a database's data dictionary as a schema object. When you use a view to do something, Oracle derives the data of the view from the view's defining query. To create a view, you use the SQL command CREATE VIEW.
- A sequence is a schema object that generates a series of unique integers. Sequences are most often used to generate unique primary keys for ID type columns. When an application inserts a new row into a table, the application can request a database sequence to generate the next available value in the sequence for the new row's primary key value. The application can subsequently reuse a generated sequence number to coordinate the foreign key values in related child rows. To create a sequence, use the SQL command CREATE SEQUENCE. To generate and then reuse a sequence number, reference the sequence's NEXTVAL and CURRVAL pseudocolumns, respectively.
- To help make applications less dependent on tables and other schema objects, you can create synonyms for schema objects. A synonym is an alias for a table, view, sequence, or other schema object that you store in the database. You create synonyms with the SQL command CREATE SYNONYM.
- To improve the performance of table access, you can create an index for one or more columns in the table. Use the SQL command CREATE INDEX to create an index.