Basics of SQL
PL/SQL is a procedural language that Oracle developed as an extension to standard SQL to provide a way to execute procedural logic on the database.

SQL, SQL*Plus, PL/SQL: What's the Difference?
This question has bedeviled many people new to Oracle. There are several products with the letters "SQL" in the title, and these three, SQL*Plus, SQL, and PL/SQL, are often used together. Because of this, it's easy to become confused as to which product is doing the work and where the work is being done. This section briefly describes each of these three products.

SQL
SQL stands for Structured Query Language. This has become the lingua franca of database access languages. It has been adopted by the International Standards Organization (ISO) and has also been adopted by the American National Standards Institute (ANSI). When you code statements such as SELECT, INSERT, UPDATE, and DELETE, SQL is the language you are using. It is a declarative language and is always executed on the database server. Often you will find yourself coding SQL statements in a development tool, such as PowerBuilder or Visual Basic, but at runtime those statements are sent to the server for execution.

PL/SQL
PL/SQL is Oracle's Procedural Language extension to SQL. It, too, usually runs on the database server, but some Oracle products such as Developer/2000 also contain a PL/SQL engine that resides on the client. Thus, you can run your PL/SQL code on either the client or the server depending on which is more appropriate for the task at hand. Unlike SQL, PL/SQL is procedural, not declarative. This means that your code specifies exactly how things get done. As in SQL, however, you need some way to send your PL/SQL code up to the server for execution. PL/SQL also enables you to embed SQL statements within its procedural code. This tight-knit relationship between PL/SQL, SQL, and SQL*Plus is the cause for some of the confusion between the products.

SQL*Plus
SQL*Plus is an interactive program that allows you to type in and execute SQL statements. It also enables you to type in PL/SQL code and send it to the server to be executed. SQL*Plus is one of the most common front ends used to develop and create stored PL/SQL procedures and functions.
What happens when you run SQL*Plus and type in a SQL statement? Where does the processing take place? What exactly does SQL*Plus do, and what does the database do? If you are in a Windows environment and you have a database server somewhere on the network, the following things happen:

1. SQL*Plus transmits your SQL query over the network to the database server.
2. SQL*Plus waits for a reply from the database server.
3. The database server executes the query and transmits the results back to SQL*Plus.
4. SQL*Plus displays the query results on your computer screen.

Even if you're not running in a networked Windows environment, the same things happen. The only difference might be that the database server and SQL*Plus are running on the same physical machine. This would be true, for example, if you were running Personal Oracle on a single PC.
PL/SQL is executed in much the same manner. Type a PL/SQL block into SQL*Plus, and it is transmitted to the database server for execution. If there are any SQL statements in the PL/SQL code, they are sent to the server's SQL engine for execution, and the results are returned back to the PL/SQL program.

DDL statements
Data definition language (DDL) refers to the subgroup of SQL statements that create, alter, or drop database objects.
This sub-category of SQL statements is of particular interest to database architects or database administrators who must define an original database design and who must respond to requirements and extend a database design. It is also of interest to database application developers, because there are times when the easiest way to meet an application requirement is to extend an existing database object definition.

In general DDL statements begin with one of the following keywords: CREATE, ALTER, or DROP. Examples of DDL statements for creating database objects include: CREATE TABLE, CREATE TRIGGER, CREATE PROCEDURE, and CREATE SEQUENCE. These statements generally contain multiple clauses used to define the characteristics and behavior of the database object. Examples of DDL statements for altering database objects include: ALTER TABLE, and ALTER PROCEDURE. These statements generally are used to alter a characteristic of a database object.

DDL statements can be executed from a variety of interactive and application interfaces although they are most commonly executed in scripts or from integrated development environments that support database and database object design.

Describing Tables
The best way to think of a table for most Oracle beginners is to envision a spreadsheet containing several records of data. Across the top, try to see a horizontal list of column names that label the values in these columns. Each record listed across the table is called a row. In SQL*Plus, the command describe enables you to obtain a basic listing of characteristics about the table.

```
SQL> DESCRIBE po_headers_all
Name                  Type         Nullable Default        Comments
--------------------- ------------ -------- -------------------------
PO_HEADER_ID
NUMBER
AGENT_ID
NUMBER(9)
```

TYPE_LOOKUP_CODE
VARCHAR2(25)
LAST_UPDATE_DATE
DATE
LAST_UPDATED_BY
NUMBER
SEGMENT1 VARCHAR2(20) R2(1)

Commenting Objects

You can also add comments to a table or column using the comment command. This is useful especially for large databases where you want others to understand some specific bits of information about a table, such as the type of information stored in the table. An example of using this command to add comments to a table appears in the following block:

```
SQL> comment on table employee is
  2  'This is a table containing employees';
Comment created.
```

You can see how to use the comment command for adding comments on table columns in the following code block:

```
SQL> comment on column employee.empid is
  2  'unique text identifier for employees';
Comment created.
```

Tip

Comment information on tables is stored in an object called USER_TAB_COMMENTS, whereas comment information for columns is stored in a different database object, called USER_COL_COMMENTS. These objects are part of the Oracle data dictionary. You'll find out more about the Oracle data dictionary later in the book.

Create TABLE

The CREATE TABLE statement allows you to create and define a table.
The basic syntax for a CREATE TABLE statement is:

```
CREATE TABLE table_name
( column1 datatype null/not null,
  column2 datatype PRIMARY KEY,
  column3 datatype PRIMARY KEY,
  ...
);
```

Each column must have a datatype. The column should either be defined as "null" or "not null" and if this value is left blank, the database assumes "null" as the default.

Example 1:

```
CREATE TABLE XX_PO_HEADERS_ALL
(
```

```
PO_ID       NUMBER(12) PRIMARY KEY,
PO_NUMBER    NUMBER(12),
SUPPLIER_NAME VARCHAR2(12) NOT NULL,
SUPPLIER_SITE VARCHAR2(12),
SHIP_TO      VARCHAR2(12),
BILL_TO      VARCHAR2(12)
)
```

Example 2:

```
CREATE TABLE XX_PO_LINES_ALL
(
PO_ID       NUMBER(12),
LINE_NUMBER   NUMBER(12),
ITEM_NAME    VARCHAR2(12),
QUANTITY     NUMBER(12),
ITEM_PRICE    NUMBER(12),
ITEM_TAX     NUMBER(12),
LINE_PRICE    NUMBER(12),
ORDER_DATE    DATE,
NEED_DATE    VARCHAR2(12),
BILL_TO      VARCHAR2(12)
)
```

Column Default Values

You can define tables to populate columns with default values as well using the default clause in a create table command. This clause is included as part of the column definition to tell Oracle what the default value for that column should be. When a row is added to the table and no value is defined for the column in the row being added, Oracle populates the column value for that row using the default value for the column. The following code block illustrates this point:

```
SQL> create table display
  2  (col1 varchar2(10),
  3   col2 number default 0);
Table created.
```

CREATE a table from another table

You can also create a table from an existing table by copying the existing table's columns.
The basic syntax is:

```
CREATE TABLE table_name
  AS (SELECT * FROM old_table);
```

Example1:

```
CREATE TABLE XX_PO_HEADERS_ALL_COPY
AS (Select * From XX_PO_HEADERS_ALL)
```

The above statement 'll create a new table that is just an exact copy of XX_PO_HEADERS_ALL.

Example 2: Copying selected columns from another table
The basic syntax is:

```
CREATE TABLE new_table
  AS (SELECT column_1, column2, ... column_n FROM old_table);
```

Example 3: Copying selected columns from multiple tables
The basic syntax is:

```
CREATE TABLE new_table
  AS (SELECT column_1, column2, ... column_n
      FROM old_table_1, old_table_2, ... old_table_n);
```

It is important to note that when creating a table in this way, the new table will be populated with the records from the existing table (based on the SELECT Statement). If you want to create a blank table then use a condition which is always false in the where clause of the select statement.

Creating Temporary Tables

Most of the time, when you create a table in Oracle, the records that eventually populate that table will live inside your database forever (or at least until someone removes them). However, there might be situations where you want records in a table to live inside the database only for a short while. In this case, you can create temporary tables in Oracle, where the data placed into the tables persists for only the duration of the user session, or for the length of your current transaction.

A temporary table is created using the create global temporary table command. Why does a temporary table have to be global? So that the temporary table's definition can be made available to every user on the system. However, the contents of a temporary table are visible only to the user session that added information to the temporary table, even though everyone can see the definition. Temporary tables are a relatively new feature in Oracle, and Oracle hasn't had enough time yet to implement "local" temporary tables (that is, temporary tables that are only available to the user who owns them). Look for this functionality in later database releases. The appropriate create global temporary table command is shown in the following code block:

```
Create global temporary table XXX_PO_HEADERS_ALL as
Select *
From PO_HEADERS_ALL
Where 10=11
```

The purpose of writing the where clause is to make the temporary table blank. If we dont put the where clause the temporary table would contain all the rows of XXX_PO_HEADERS_ALL

ALTER TABLE

The ALTER TABLE statement allows you to rename an existing table. It can also be used to add, modify, or drop a column from an existing table.

Renaming a table

The basic syntax for renaming a table is:

```
ALTER TABLE table_name
 RENAME TO new_table_name;
```

For example:

```
ALTER TABLE suppliers
 RENAME TO vendors;
```

This will rename the suppliers table to vendors.

Adding column(s) to a table

Syntax #1

To add a column to an existing table, the ALTER TABLE syntax is:

```
ALTER TABLE table_name
 ADD column_name column-definition;
```

For example:

```
ALTER TABLE supplier
 ADD supplier_name  varchar2(50);
```

This will add a column called supplier_name to the supplier table.

Syntax #2

To add multiple columns to an existing table, the ALTER TABLE syntax is:

```
ALTER TABLE table_name
ADD (column_1    column-definition,
     column_2    column-definition,
     ...
     column_n    column_definition );
```

Drop column(s) in a table

Syntax #1

To drop a column in an existing table, the ALTER TABLE syntax is:

```
ALTER TABLE table_name
 DROP COLUMN column_name;
```

For example:

```
ALTER TABLE supplier
 DROP COLUMN supplier_name;
```

Modifying column(s)(datatypes) in a table

Syntax #1

To modify a column in an existing table, the ALTER TABLE syntax is:

```
ALTER TABLE table_name
 MODIFY column_name column_type;
```

For example:

```
ALTER TABLE supplier
 MODIFY supplier_name   varchar2(100)    not null;
```

This will modify the column called supplier_name to be a data type of varchar2(100) and force the column to not allow null values.

Syntax #2

To modify multiple columns in an existing table, the ALTER TABLE syntax is:

```
ALTER TABLE table_name
MODIFY (    column_1    column_type,
    column_2    column_type,
    ...
    column_n    column_type );
```

Rename column(s) in a table

(NEW in Oracle 9i Release 2)

Syntax #1

Starting in Oracle 9i Release 2, you can now rename a column.

To rename a column in an existing table, the ALTER TABLE syntax is:

```
ALTER TABLE table_name
 RENAME COLUMN old_name to new_name;
```

DROP TABLE

Submitted by Anonymous on Tue, 03/03/2009 - 15:55

The DROP TABLE statement allows you to remove a table from the database.

The basic syntax for the DROP TABLE statement is:

```
DROP TABLE table_name;
```

For example: DROP TABLE XX_supplier;

This would drop table called XX_supplier.

Sometimes objects are associated with a table that exists in a database along with the table. These objects may include indexes, constraints, and triggers. If the table is dropped, Oracle automatically drops any index, trigger, or constraint associated with the table as well. Here are two other factors to be aware of with respect to dropping tables:

You cannot roll back a drop table command.

To drop a table, the table must be part of your own schema, or you must have the drop any table privilege granted to you.

Truncating Tables

Let's move on to discuss how you can remove all data from a table quickly using a special option available in Oracle. In this situation, the DBA or developer may use the truncate table statement. This statement is a part of the data definition language (DDL) of Oracle, much like

the create table statement and completely unlike the delete statement. Truncating a table removes all row data from a table quickly, while leaving the definition of the table intact, including the definition of constraints and any associated database objects such as indexes, constraints, and triggers on the table. The truncate statement is a high-speed data-deletion statement that bypasses the transaction controls available in Oracle for recoverability in data changes. Truncating a table is almost always faster than executing the delete statement without a where clause, but once this operation has been completed, the data cannot be recovered unless you have a backed-up copy of the data. Here's an example:

```
SQL> truncate table tester;
Table truncated.
```

CREATE OR REPLACE VIEW

Views are queries stored in Oracle that dynamically assemble data into a virtual table. To the person using the view, manipulating the data from the view is just like manipulating the data from a table. In some cases, it is even possible for the user to change data in a view as though the view *were* a table. Let's now explore the topic of creating, using, and managing views in more detail.

Creating a VIEW

The syntax for creating a VIEW is:

```
CREATE VIEW view_name AS
SELECT columns
FROM table
WHERE predicates;
```

A view will not be created if the base table you specify does not exist. However, you can overcome this restriction by using the force keyword in the create view command. This keyword forces Oracle to create the view anyway. However, the view will be invalid because no underlying table data is available to draw from.

For example:

```
Create VIEW XX_PO_DETAILS_v AS
Select a.PO_ID, a.PO_NUMBER, b.ITEM_NAME, b.NEED_DATE
From XX_PO_HEADERS_ALL a, XX_PO_LINES_ALL b
Where a.PO_ID=b.PO_ID
```

This would create a virtual table based on the result set of the select statement. You can now query the view as follows:

```
SELECT *
FROM  XX_PO_DETAILS_vs;
```

Creating Views That Enforce Constraints

Tables that underlie views often have constraints that limit the data that can be added to those tables. As I said earlier, views cannot add data to the underlying table that would violate the table's constraints. However, you can also define a view to restrict the user's ability to change underlying table data even further, effectively placing a special constraint for data

manipulation through the view. This additional constraint says that insert or update statements issued against the view are cannot create rows that the view cannot subsequently select. In other words, if after the change is made, the view will not be able to select the row you changed, the view will not let you make the change. This viewability constraint is configured when the view is defined by adding the with check option to the create view statement. Let's look at an example to clarify my point:

```
create or replace view emp_view as
(select empno, ename, job, deptno
  from emp
  where deptno = 10)
 with check option constraint emp_view_constraint;
```

Updating a VIEW

You can update a VIEW without dropping it by using the following syntax:

```
CREATE OR REPLACE VIEW view_name AS
SELECT columns
FROM table
WHERE predicates;
```

For example:

```
CREATE or REPLACE VIEW sup_orders AS
SELECT suppliers.supplier_id, orders.quantity, orders.price
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id
and suppliers.supplier_name = 'Microsoft';
```

Dropping a VIEW

The syntax for dropping a VIEW is:

```
DROP VIEW view_name;
```

For example:

```
DROP VIEW sup_orders;
```

Creating Simple Views That Can't Change Underlying Table Data

In some cases, you may find that you want to create views that don't let your users change data in the underlying table. In this case, you can use the with read only clause. This clause will prevent any user of the view from making changes to the base table. Let's say that after reprimanding SCOTT severely for calling him a fool, KING wants to prevent all employees from ever changing data in EMP via the EMP_VIEW again. The following shows how he would do it:

```
create or replace view emp_view
 as (select * from emp)
 with read only;
```

Frequently Asked Questions

Question: Can you update the data in a view?

Answer: A view is created by joining one or more tables. When you update record(s) in a view, it updates the records in the underlying tables that make up the view. So, yes, you can update the data in a view providing you have the proper privileges to the underlying tables.

Question: Does the view exist if the table is dropped from the database?

Answer: Yes, in Oracle, the view continues to exist even after one of the tables (that the view is based on) is dropped from the database. However, if you try to query the view after the table has been dropped, you will receive a message indicating that the view has errors.If you recreate the table (that you had dropped), the view will again be fine.

Temporary tables

Global temporary tables

Global temporary tables are distinct within SQL sessions.

The basic syntax is:

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

For example:

```
CREATE GLOBAL TEMPORARY TABLE supplier
(    supplier_id     numeric(10)     not null,
     supplier_name     varchar2(50)     not null,
     contact_name     varchar2(50)
)
```

This would create a global temporary table called supplier .

Local Temporary tables

Local temporary tables are distinct within modules and embedded SQL programs within SQL sessions.

The basic syntax is:

```
DECLARE LOCAL TEMPORARY TABLE table_name ( ...);
```

Constraints

Constraints are rules you can define in your Oracle tables to restrict the type of data you can place in the tables.

Constraint	Description
Primary Key	Stipulates that values in the constrained column(s) must be unique and not NULL. If the primary key applies to multiple columns, then the combination of values in the columns must be unique and not NULL.
Foreign Key	Enforces that only values in the primary key of a parent table may be included as values in the constrained column of the child table.
Unique	Enforces uniqueness on values in the constrained column.
Check	Enforces that values added to the constrained column must be present in a static list of values permitted for the column.
Not Null	Enforces that a value must be defined for this column such that the column may not be NULL for any row.

Two methods exist for defining constraints: the table constraint method and the column constraint method. The constraint is defined as a table constraint if the constraint clause syntax appears after the column and datatype definitions. The constraint is defined as a column

constraint if the constraint definition syntax appears as part of an individual column's definition. All constraints can be defined either as table constraints or as column constraints, with two exceptions:

Not NULL constraints can only be defined as column constraints.

Primary keys consisting of two or more columns (known also as composite primary keys) can only be defined as table constraints. However, single-column primary keys can be defined either as column or table constraints.

Primary Key

A constraint of this type identifies the column or columns whose singular or combined values identify uniqueness in the rows of your Oracle table. Every row in the table must have a value specified for the primary key column(s).

```
SQL> create table employee
(
 empid varchar2(5)  constraint pk_employee_01 primary key,
 lastname varchar2(25),
 firstname varchar2(25),
 salary number(10,4)
);     //column constraint method
```

Or cab be simplified as

```
create table employee
(
 empid varchar2(5)  primary key,
 lastname varchar2(25),
 firstname varchar2(25),
 salary number(10,4)
);     //column constraint method
```

Each primary key constraint was given a meaningful name when defined. Oracle strongly recommends that you give your constraints meaningful names in this way so that you can easily identify the constraint later

```
 create table employee
 (
 empid varchar2(5),
 lastname varchar2(25),
 firstname varchar2(25),
 salary number(10,4),
 constraint pk_employee_01  primary key (empid)
);     //table constraint method
```

Composite Primary Keys

So that you understand the nature of composite primary keys in Oracle for OCP, the following code block shows how to define a composite primary key:

```
create table names
(
firstname varchar2(10),
lastname varchar2(10),
constraint pk_names_01 primary key (firstname, lastname)
);
```

Defining Foreign Key Constraints

To help you understand how to define foreign key constraints, let's think in terms of an example. Let's say we have our own table called DEPARTMENT, which we just created in the last code block. It lists the department number, name, and location for all departments in our own little company. Let's also say that we want to create our EMPLOYEE table with another column, called DEPARTMENT_NUM. Because there is an implied parent-child relationship between these two tables with the shared column, let's make that relationship official by using a foreign key constraint, shown in bold in the following block:

```
create table employee
(
empid varchar2(5) primary key,
lastname varchar2(25),
firstname varchar2(25),
department_num number(5) references department(department_num) on delete set null,
salary number(10,4)
);  //references table_name(column_name) on delete set null,
```

For a foreign-key constraint to be valid, the same column appearing in both tables must have exactly the same datatype. You needn't give the columns the same names, but it's a good idea to do so. The foreign key constraint prevents the DEPARTMENT_NUM column in the EMP table from ever storing a value that can't also be found in the DEPT table. The final clause, on delete set null, is an option relating to the deletion of data from the parent table. If someone attempts to remove a row from the parent table that contains a referenced value from the child table, Oracle sets all corresponding values in the child to NULL. The other option is on delete cascade, where Oracle allows remove all corresponding records from the child table when a referenced record from the parent table is removed.

Unique Key Constraints

Let's say we also want our employee table to store social security or government ID information. The definition of a UNIQUE constraint on the GOVT_ID column prevents anyone from defining a duplicate government ID for any two employees in the table. Take a look at the following example, where the unique constraint definition is shown in bold:

```
create table employee
(
empid varchar2(5) primary key,
lastname varchar2(25),
firstname varchar2(25),
```

```
govt_id number(10) unique,
salary number(10,4),
department_num number(5) references department (department_num),
);
```

Defining Other Types of Constraints

The last two types of constraints are not NULL and CHECK constraints. By default, Oracle allows columns to contain NULL values. The not NULL constraint prevents the data value defined by any row for the column from being NULL. By default, primary keys are defined to be not NULL. All other columns can contain NULL data, unless you explicitly define the column to be not NULL. CHECK constraints allow Oracle to verify the validity of data being entered on a table against static criteria. For example, you could specify that the SALARY column cannot contain values over $250,000. If someone tries to create an employee row with a salary of $1,000,000 per year, Oracle would return an error message saying that the record data defined for the SALARY column has violated the CHECK constraint for that column. Let's look at a code example where both not NULL and check constraints are defined in bold:

```
create table employee
(
empid varchar2(5) primary key,
department_num number(5) references department (department_num),
lastname varchar2(25) not null,
firstname varchar2(25) unique,
salary number(10,4) check (salary <=250000),
govt_id number(10) unique
);
```

Adding Integrity Constraints to Existing Tables

Another constraint-related activity that you may need to do involves adding new constraints to an existing table. This can be easy if there is no data in the table already, but it can be a nightmare if data already exists in the table that doesn't conform to the constraint criteria. The simplest scenario for adding the constraint is to add it to the database before data is inserted. Take a look at the following code block:

```
SQL> create table employee
  2  (empid varchar2(5),
  3   lastname varchar2(25),
  4   firstname varchar2(25),
  5   salary number(10,4),
  6   department_num number(5),
  7   govt_id number(10));
Table created.

alter table employee
add constraint pk_employee_01 primary key (empid);
```

```
alter table employee
add constraint fk_employee_01 foreign key (department_num) references department
(department_num);
alter table employee
add constraint ck_employee_01 check (salary <=250000);
alter table employee
add constraint uk_employee_01 unique (govt_id);
alter table employee modify
(lastname not null);
```

Disabling Constraints

A constraint can be turned on and off. When the constraint is disabled, it will no longer do its job of enforcing rules on the data entered into the table. The following code block demonstrates some sample statements for disabling constraints:

```
alter table employee
disable primary key;
alter table employee
disable constraint uk_employee_01;
```

You may experience a problem if you attempt to disable a primary key when existing foreign keys depend on that primary key. This problem is shown in the following situation:

Enabling a Disabled Constraint

When the constraint is later enabled, the rules defined for the constraint are once again enforced, rendering the constraint as effective as it was when it was first added to the table. You can enable a disabled constraint as follows:

```
alter table department
enable primary key;
alter table employee
enable uk_employee_01;
```

Removing Constraints

Usually, there is little about a constraint that will interfere with your ability to remove it, so long as you either own the table or have been granted appropriate privileges to do so. When a constraint is dropped, any index associated with that constraint (if there is one) is also dropped. Here is an example:

```
alter table employee
drop unique (govt_id);
alter table employee
drop primary key cascade;
alter table employee
drop constraint ck_employee_01;
```

An anomaly can be found when disabling or dropping not NULL constraints. You cannot

disable a not NULL constraint, per se—a column either accepts NULL values or it doesn't. Therefore, you must use the alter table modify clause in all situations where the not NULL constraints on a table must be added or removed. Here's an example:

```
alter table employee
modify (lastname null);
alter table employee
modify (lastname not null);
```

SELECT Statement

The SELECT statement is used to query the database and retrieve selected data that match the criteria that you specify.

The SELECT statement has five main clauses to choose from, although, FROM is the only required clause. Each of the clauses have a vast selection of options, parameters, etc. The clauses will be listed below, but each of them will be covered in more detail later in the tutorial.

Here is the format of the SELECT statement:

```
SELECT [ALL | DISTINCT] column1[,column2]
FROM table1[,table2]
[WHERE "conditions"]
[GROUP BY "column-list"]
[HAVING "conditions]
[ORDER BY "column-list" [ASC | DESC] ]
```

```
SELECT column_name(s)
FROM table_name
and
SELECT * FROM table_name
```

DISTINCT Clause

The DISTINCT clause allows you to remove duplicates from the result set. The DISTINCT clause can only be used with select statements.

The syntax for the DISTINCT clause is:

```
SELECT DISTINCT columns     FROM tables     WHERE predicates;
```

Example #1

Let's take a look at a very simple example.

```
  SELECT DISTINCT city
  FROM suppliers;
```

This SQL statement would return all unique cities from the suppliers table.

Example #2

The DISTINCT clause can be used with more than one field.

For example:

```
  SELECT DISTINCT city, state
```

FROM suppliers;

This select statement would return each unique city and state combination. In this case, the distinct applies to each field listed after the DISTINCT keyword.

SQL WHERE

The SQL WHERE clause is used to select data conditionally, by adding it to already existing SQL SELECT query. We are going to use the Customers table from the previous chapter, to illustrate the use of the SQL WHERE command.

Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator value
```

With the WHERE clause, the following operators can be used:

Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns
BETWEEN	Between an inclusive range

"AND" Condition

The AND condition allows you to create an SQL statement based on 2 or more conditions being met. It can be used in any valid SQL statement - select, insert, update, or delete.

The syntax for the AND condition is:

```
SELECT columns
FROM tables
WHERE column1 = 'value1'
and column2 = 'value2';
```

The AND condition requires that each condition be must be met for the record to be included in the result set. In this case, column1 has to equal 'value1' and column2 has to equal 'value2'.

"OR" Condition

The OR condition allows you to create an SQL statement where records are returned when any one of the conditions are met. It can be used in any valid SQL statement - select, insert, update, or delete.

The syntax for the OR condition is:

```
SELECT columns
FROM tables
```

```
WHERE column1 = 'value1'
or column2 = 'value2';
```

The OR condition requires that any of the conditions be must be met for the record to be included in the result set. In this case, column1 has to equal 'value1' OR column2 has to equal 'value2'.

Combining the "AND" and "OR" Conditions

The AND and OR conditions can be combined in a single SQL statement. It can be used in any valid SQL statement - select, insert, update, or delete. When combining these conditions, it is important to use brackets so that the database knows what order to evaluate each condition.

Example

The first example that we'll take a look at an example that combines the AND and OR conditions.

```
SELECT *
FROM suppliers
WHERE (city = 'New York' and name = 'IBM')
or (city = 'Newark');
```

LIKE Operator

The LIKE operator is used to search for a specified pattern in a column.

SQL LIKE Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern
```

The LIKE condition can be used in any valid SQL statement - select, insert, update, or delete.

The patterns that you can choose from are:

% allows you to match any string of any length (including zero length)

_ allows you to match on a single character

Examples using % wildcard

The first example that we'll take a look at involves using % in the where clause of a select statement. We are going to try to find all of the suppliers whose name begins with 'Hew'.

```
SELECT *
FROM suppliers
WHERE supplier_name like 'Hew%';
```

You can also using the wildcard multiple times within the same string. For example,

```
SELECT *
FROM suppliers
WHERE supplier_name like '%bob%';
```

Examples using _ wildcard
Next, let's explain how the _ wildcard works. Remember that the _ is looking for only one character.
For example,

```
SELECT *
FROM suppliers
WHERE supplier_name like 'Sm_th';
```

This SQL statement would return all suppliers whose name is 5 characters long, where the first two characters is 'Sm' and the last two characters is 'th'. For example, it could return suppliers whose name is 'Smith', 'Smyth', 'Smath', 'Smeth', etc.

IN Function

The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...)
```

The following is an SQL statement that uses the IN function:

```
SELECT *
FROM suppliers
WHERE supplier_name in ( 'IBM', 'Hewlett Packard', 'Microsoft');
```

This would return all rows where the supplier_name is either IBM, Hewlett Packard, or Microsoft. Because the * is used in the select, all fields from the suppliers table would appear in the result set.
It is equivalent to the following statement:

```
SELECT *
FROM suppliers
WHERE supplier_name = 'IBM'
OR supplier_name = 'Hewlett Packard'
OR supplier_name = 'Microsoft';
```

As you can see, using the IN function makes the statement easier to read and more efficient.

BETWEEN

The BETWEEN condition allows you to retrieve values within a range.
The syntax for the BETWEEN condition is:

```
SELECT columns
FROM tables
WHERE column1 between value1 and value2;
```

This SQL statement will return the records where column1 is within the range of value1 and value2 (inclusive). The BETWEEN function can be used in any valid SQL statement - select, insert, update, or delete.

The following is an SQL statement that uses the BETWEEN function:
SELECT *
FROM suppliers
WHERE supplier_id between 5000 AND 5010;

This would return all rows where the supplier_id is between 5000 and 5010, inclusive. It is equivalent to the following SQL statement:
SELECT *
FROM suppliers
WHERE supplier_id >= 5000
AND supplier_id <= 5010;

EXISTS Condition
The EXISTS condition is considered "to be met" if the subquery returns at least one row.
The syntax for the EXISTS condition is:
SELECT columns
FROM tables
WHERE EXISTS (subquery);

The EXISTS condition can be used in any valid SQL statement - select, insert, update, or delete.
Example1:
Let's take a look at a simple example. The following is an SQL statement that uses the EXISTS condition:
SELECT *
FROM suppliers
WHERE EXISTS (select * from orders where suppliers.supplier_id = orders.supplier_id);
This select statement will return all records from the suppliers table where there is at least one record in the orders table with the same supplier_id.

Example 2: NOT EXISTS
The EXISTS condition can also be combined with the NOT operator.
For example,
SELECT * FROM suppliers WHERE not exists (select * from orders Where suppliers.supplier_id = orders.supplier_id);
This will return all records from the suppliers table where there are no records in the orders table for the given supplier_id.

Example 3: DELETE Statement
The following is an example of a delete statement that utilizes the EXISTS condition:
DELETE FROM suppliers WHERE EXISTS (select * from orders where suppliers.supplier_id = orders.supplier_id);

GROUP BY

Group functions allow you to perform data operations on several values in a column of data as though the column were one collective group of data. These functions are also called *group-by functions* because they are often used in a special clause of select statements, called the group by clause.

The syntax for the GROUP BY clause is:

```
SELECT column1, column2, ... column_n, aggregate_function (expression)
FROM tables
WHERE predicates
GROUP BY column1, column2, ... column_n;
```

aggregate_function can be a function such as SUM, COUNT, MIN, or MAX.

Here's a list of the available group functions:

avg(x) Averages all x column values returned by the select statement

count(x) Counts the number of non-NULL values returned by the select statement for column x

max(x) Determines the maximum value in column x for all rows returned by the select statement

min(x) Determines the minimum value in column x for all rows returned by the select statement

stddev(x) Calculates the standard deviation for all values in column x in all rows returned by the select statement

sum(x) Calculates the sum of all values in column x in all rows returned by the select statement

Variance(x) Calculates the variance for all values in column x in all rows returned by the select statement

Example using the SUM function

For example, you could also use the SUM function to return the name of the department and the total sales (in the associated department).

```
SELECT department, SUM(sales) as "Total sales"
FROM order_details
GROUP BY department;
```

Because you have listed one column in your SELECT statement that is not encapsulated in the SUM function, you must use a GROUP BY clause. The department field must, therefore, be listed in the GROUP BY section.

Example using the COUNT function

For example, you could use the COUNT function to return the name of the department and the number of employees (in the associated department) that make over $25,000 / year.

```
SELECT department, COUNT(*) as "Number of employees"
FROM employees
WHERE salary > 25000
GROUP BY department;
```

ROLLUP

This group by operation is used to produce subtotals at any level of aggregation needed. These subtotals then "roll up" into a grand total, according to items listed in the group by expression.

The totaling is based on a one-dimensional data hierarchy of grouped information. For example, let's say we wanted to get a payroll breakdown for our company by department and job position. The following code block would give us that information:

```
SQL> select deptno, job, sum(sal) as salary
  2  from emp
  3  group by rollup(deptno, job);
   DEPTNO JOB         SALARY
--------- --------- ---------
       10 CLERK         1300
       10 MANAGER       2450
       10 PRESIDENT     5000
       10               8750
       20 ANALYST       6000
       20 CLERK         1900
       20 MANAGER       2975
       20              10875
       30 CLERK          950
       30 MANAGER       2850
       30 SALESMAN      5600
       30               9400
                       29025
```

Notice that NULL values in the output of rollup operations typically mean that the row contains subtotal or grand total information. If you want, you can use the nvl( ) function to substitute a more meaningful value.

cube

cube This is an extension, similar to rollup. The difference is that cube allows you to take a specified set of grouping columns and create subtotals for all possible combinations of them. The cube operation calculates all levels of subtotals on horizontal lines across spreadsheets of output and creates cross-tab summaries on multiple vertical columns in those spreadsheets. The result is a summary that shows subtotals for every combination of columns or expressions in the group by clause, which is also known as n-dimensional cross-tabulation. In the following example, notice how cube not only gives us the payroll breakdown of our company by DEPTNO and JOB, but it also gives us the breakdown of payroll by JOB across all departments:

```
SQL>  select deptno, job, sum(sal) as salary
  2  from emp
  3  group by cube(deptno, job);
DEPTNO JOB         SALARY
--------- --------- ---------
       10 CLERK         1300
       10 MANAGER       2450
       10 PRESIDENT     5000
       10               8750
       20 ANALYST       6000
```

```
20 CLERK         1900
20 MANAGER       2975
20               10875
30 CLERK         950
30 MANAGER       2850
30 SALESMAN      5600
30               9400
   ANALYST       6000
   CLERK         4150
   MANAGER       8275
   PRESIDENT     5000
   SALESMAN      5600
                 29025
```

Excluding group Data with having

Once the data is grouped using the group by statement, it is sometimes useful to weed out unwanted data. For example, let's say we want to list the average salary paid to employees in our company, broken down by department and job title. However, for this query, we only care about departments and job titles where the average salary is over $2000. In effect, we want to put a where clause on the group by clause to limit the results we see to departments and job titles where the average salary equals $2001 or higher. This effect can be achieved with the use of a special clause called the having clause, which is associated with group by statements. Take a look at an example of this clause:

```
SQL> select deptno, job, avg(sal)
  2  from emp
  3  group by deptno, job
  4  having avg(sal) > 2000;
   DEPTNO JOB       AVG(SAL)
--------- --------- ---------
       10 MANAGER        2450
       10 PRESIDENT      5000
       20 ANALYST        3000
       20 MANAGER        2975
       30 MANAGER        2850
```

Consider the output of this query for a moment. First, Oracle computes the average for every department and job title in the entire company. Then, the having clause eliminates departments and titles whose constituent employees' average salary is $2000 or less. This selectivity cannot easily be accomplished with an ordinary where clause, because the where clause selects individual rows, whereas this example requires that groups of rows be selected. In this query, you successfully limit output on the group by rows by using the having clause.

HAVING clause

The SQL HAVING clause is used to restrict conditionally the output of a SQL statement, by a SQL aggregate function used in your SELECT list of columns.

```
SELECT column1, column2, ... column_n, aggregate_function (expression)
FROM tables
WHERE predicates
GROUP BY column1, column2, ... column_n
HAVING condition1 ... condition_n;
```

You can't specify criteria in a SQL WHERE clause against a column in the SELECT list for which SQL aggregate function is used. For example the following SQL statement will generate an error:

```
SELECT Employee, SUM (Hours)
FROM EmployeeHours
WHERE SUM (Hours) > 24
GROUP BY Employee
```

The SQL HAVING clause is used to do exactly this, to specify a condition for an aggregate function which is used in your query:

```
SELECT Employee, SUM (Hours)
FROM EmployeeHours
GROUP BY Employee
HAVING SUM (Hours) > 24
```

ORDER BY

So far, we have seen how to get data out of a table using SELECT and WHERE commands. Often, however, we need to list the output in a particular order. This could be in ascending order, in descending order, or could be based on either numerical value or text value. In such cases, we can use the ORDER BY keyword to achieve our goal.

The syntax for an ORDER BY statement is as follows:

```
SELECT "column_name"
FROM "table_name"
[WHERE "condition"]
ORDER BY "column_name" [ASC, DESC]
```

The [] means that the WHERE statement is optional. However, if a WHERE clause exists, it comes before the ORDER BY clause. ASC means that the results will be shown in ascending order, and DESC means that the results will be shown in descending order. If neither is specified, the default is ASC.

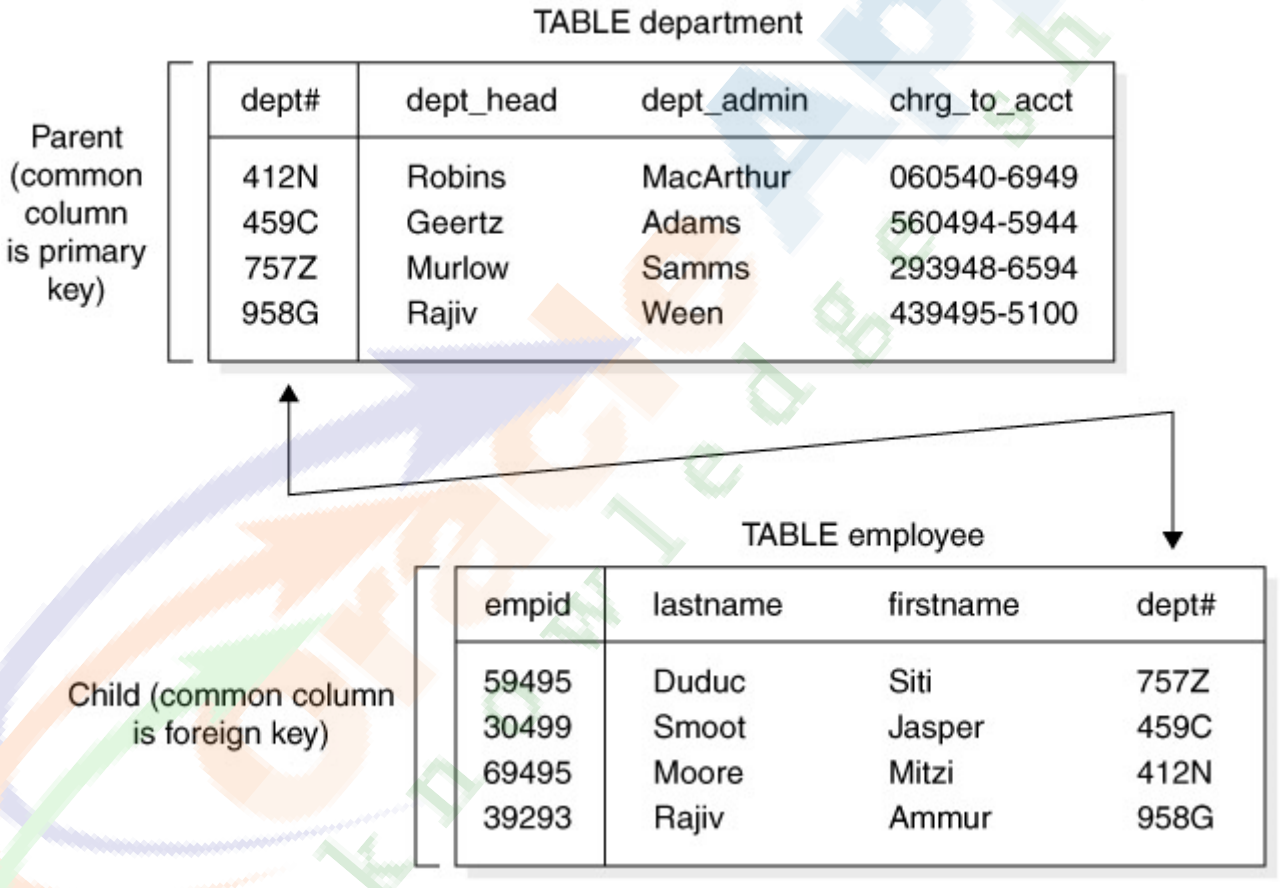
It is possible to order by more than one column. In this case, the ORDER BY clause above becomes ORDER BY "column_name1" [ASC, DESC], "column_name2" [ASC, DESC]

Table Joins

The typical database contains many tables. Some smaller databases may have only a dozen or so tables, whereas other databases may have hundreds or even thousands. The common factor,

however, is that few databases have just one table containing everything you need. Therefore, you usually have to draw data from multiple tables together in a meaningful way. To show data from multiple tables in one query, Oracle allows you to perform table joins.

Here are the two rules you need to remember for table joins. Data from two (or more) tables can be joined, if the same column (under the same or a different name) appears in both tables, and the column is the primary key (or part of that key) in one of the tables. Having a common column in two tables implies a relationship between the two tables. The nature of that relationship is determined by which table uses the column as a primary key. This begs the question, what is a primary key? A primary key is a column in a table used for identifying the uniqueness of each row in a table. The table in which the column appears as a primary key is referred to as the parent table in this relationship (sometimes also called the master table), whereas the column that references the other table in the relationship is often called the child table (sometimes also called the detail table). The common column appearing in the child table is referred to as a foreign key.



Join Syntax

Let's look at an example of a join statement using the Oracle traditional syntax, where we join the contents of the EMP and DEPT tables together to obtain a listing of all employees, along with the names of the departments they work for:

```
SQL> select e.ename, e.deptno, d.dname
  2  from emp e, dept d
  3  where e.deptno = d.deptno;
ENAME      DEPTNO     DNAME
----------  ---------  --------------
SMITH        20        RESEARCH
ALLEN         30        SALES
WARD         30        SALES
JONES        20        RESEARCH
```

Note the many important components in this table join. Listing two tables in the from clause clearly indicates that a table join is taking place. Note also that each table name is followed by a letter: E for EMP or D for DEPT. This demonstrates an interesting concept—just as columns can have aliases, so too can tables. The aliases serve an important purpose—they prevent Oracle from getting confused about which table to use when listing the data in the DEPTNO column. Remember, EMP and DEPT both have a column named DEPTNO
You can also avoid ambiguity in table joins by prefixing references to the columns with the table names, but this often requires extra coding. You can also give the column two different names, but then you might forget that the relationship exists between the two tables. It's just better to use aliases! Notice something else, though. Neither the alias nor the full table name needs to be specified for columns appearing in only one table.

Take a look at another example:

```
SQL> select ename, emp.deptno, dname
  2  from emp, dept
  3  where emp.deptno = dept.deptno;
```

How Many Comparisons Do You Need?

When using Oracle syntax for table joins, a query on data from more than two tables must contain the right number of equality operations to avoid a Cartesian product. To avoid confusion, use this simple rule: If the number of tables to be joined equals N, include at least N-1 equality conditions in the select statement so that each common column is referenced at least once. Similarly, if you are using the ANSI/ISO syntax for table joins, you need to use N-1 join tablename on join_condition clauses for every N tables being joined.
For N joined tables using Oracle or ANSI/ISO syntax for table joins, you need at least N-1 equijoin conditions in the where clause of your select statement or N-1 join tablename on join_condition clauses in order to avoid a Cartesian product, respectively.

Cartesian Products

Notice also that our where clause includes a comparison on DEPTNO linking data in EMP to that of DEPT. Without this link, the output would have included all data from EMP and DEPT, jumbled together in a mess called a Cartesian product. Cartesian products are big, meaningless listings of output that are nearly never what you want. They are formed when you omit a join condition in your SQL statement, which causes Oracle to join all rows in the first table to all

rows in the second table. Let's look at a simple example in which we attempt to join two tables, each with three rows, using a select statement with no where clause, resulting in output with nine rows:

```
SQL> select a.col1, b.col_2
  2  from example_1 a, example_2 b;
    COL1 COL_2
-------- ------------------------------
       1 one
       2 one
       3 one
       1 two
       2 two
       3 two
```

You must always remember to include join conditions in join queries to avoid Cartesian products. But take note of another important fact. Although we know that where clauses can contain comparison operations other than equality, to avoid Cartesian products, you must always use equality operations in a comparison joining data from two tables. If you want to use another comparison operation, you must first join the information using an equality comparison and then perform the other comparison somewhere else in the where clause. This is why table join operations are also sometimes referred to as equijoins. Take a look at the following example that shows proper construction of a table join, where the information being joined is compared further using a nonequality operation to eliminate the employees from accounting:

```
SQL> select ename, emp.deptno, dname
  2  from emp, dept
  3  where emp.deptno = dept.deptno
  4  and dept.deptno > 10;
```

ANSI/ISO Join Syntax (Oracle9i and higher)

In Oracle9i, Oracle introduces strengthened support for ANSI/ISO join syntax. To join the contents of two tables together in a single result according to that syntax, we must include a join tablename on join_condition in our SQL statement. If we wanted to perform the same table join as before using this new syntax, our SQL statement would look like the following:

```
Select ename, emp.deptno, dname
from emp join dept
on emp.deptno = dept.deptno;
ENAME      DEPTNO    DNAME
---------- --------- --------------
SMITH         20       RESEARCH
ALLEN         30       SALES
```

```
WARD        30      SALES
JONES       20      RESEARCH
```

Note how different this is from Oracle syntax. First, ANSI/ISO syntax separates join comparisons from all other comparisons by using a special keyword, on, to indicate what the join comparison is. You can still include a where clause in your ANSI/ISO-compliant join query, the only difference is that the where clause will contain only those additional conditions you want to use for filtering your data. You also do not list all your tables being queried in one from clause. Instead, you use the join clause directly after the from clause to identify the table being joined.
Never combine Oracle's join syntax with ANSI/ISO's join syntax! Also, there are no performance differences between Oracle join syntax and ANSI/ISO join syntax.

Cartesian Products: An ANSI/ISO Perspective
In some cases, you might actually want to retrieve a Cartesian product, particularly in financial applications where you have a table of numbers that needs to be cross-multiplied with another table of numbers for statistical analysis purposes. ANSI/ISO makes a provision in its syntax for producing Cartesian products through the use of a cross-join. A cross-join is produced when you use the cross keyword in your ANSI/ISO-compliant join query. Recall from a previous example that we produced a Cartesian product by omitting the where clause when joining two sample tables, each containing three rows, to produce nine rows of output. We can produce this same result in ANSI/ISO SQL by using the cross keyword, as shown here in bold:

```
Select col1, col_2
from example_1 cross join example_2;
COL1    COL_2
--------- ------------
1       one
2       one
3       one
1       two
2       two
3       two
1       three
```

Natural Joins
One additional type of join you need to know about for OCP is the natural join. A natural join is a join between two tables where Oracle joins the tables according to the column(s) in the two tables sharing the same name (naturally!). Natural joins are executed whenever the natural keyword is present. Let's look at an example. Recall our use of the EMP and DEPT tables from our discussion above. Let's take a quick look at the column listings for both tables:

```
SQL> describe emp
Name                    Null        Type
------------------------ ---------   ------------
```

```
EMPNO             NOT NULL    NUMBER(4)
ENAME                         VARCHAR2(10)
JOB                          VARCHAR2(9)
MGR                           NUMBER(4)

SQL> describe dept
Name                  Null         Type
----------------------- --------     -----------
DEPTNO             NOT NULL  NUMBER(2)
DNAME                         VARCHAR2(14)
LOC                          VARCHAR2(13)
```

As you can see, DEPTNO is the only column in common between these two tables, and appropriately enough, it has the same name in both tables. This combination of facts makes our join query of EMP and DEPT tables a perfect candidate for a natural join. Take a look and see:

```
Select ename, deptno, dname
from emp natural join dept;

ENAME       DEPTNO    DNAME
----------      ---------    --------------
SMITH          20       RESEARCH
ALLEN          30       SALES
WARD          30       SALES
```

Outer Joins

Outer joins extend the capacity of Oracle queries to include handling of situations where you want to see information from tables even when no corresponding records exist in the common column. The purpose of an outer join is to include non-matching rows, and the outer join returns these missing columns as NULL values.

Left Outer Join

A left outer join will return all the rows that an inner join returns plus one row for each of the other rows in the first table that did not have a match in the second table.

Suppose you want to find all employees and the projects they are currently responsible for. You want to see those employees that are not currently in charge of a project as well. The following query will return a list of all employees whose names are greater than 'S', along with their assigned project numbers.

```
 SELECT EMPNO, LASTNAME, PROJNO
   FROM CORPDATA.EMPLOYEE LEFT OUTER JOIN CORPDATA.PROJECT
        ON EMPNO = RESPEMP
   WHERE LASTNAME > 'S'
```

The result of this query contains some employees that do not have a project number. They are listed in the query, but have the null value returned for their project number.

EMPNO	LASTNAME	PROJNO
000020	THOMPSON	PL2100
000100	SPENSER	OP2010
000170	YOSHIMURA	-
000250	SMITH	AD3112

In oracle we can specify(in Oracle 8i or prior vesrion this was the only option as they were not supporting the ANSI syntex) left outer join by putting a (+) sign on the right of the column which can have NULL data corresponding to non-NULL values in the column values from the other table.

```
example: select   last_name,   department_name
from   employees e,   departments d
where   e.department_id(+) = d.department_id;
```

Right Outer Join

A right outer join will return all the rows that an inner join returns plus one row for each of the other rows in the second table that did not have a match in the first table. It is the same as a left outer join with the tables specified in the opposite order.

The query that was used as the left outer join example could be rewritten as a right outer join as follows:

```
SQL> -- Earlier version of outer join
SQL> -- select e.ename, e.deptno, d.dname
SQL> -- from dept d, emp e
SQL> -- where d.deptno = e.deptno (+);
SQL>
SQL> -- ANSI/ISO version
SQL> select e.ename, e.deptno, d.dname
SQL> from emp e right outer join dept d
SQL> on d.deptno = e.deptno;
```

Full Outer Joins

Oracle9i and higher

Oracle9i also makes it possible for you to easily execute a full outer join, including all records from the tables that would have been displayed if you had used both the left outer join or right outer join clauses. Let's take a look at an example:

```
SQL> select e.ename, e.deptno, d.dname
2 from emp e full outer join dept d
3 on d.deptno = e.deptno;
```

Oracle Outer Join Syntax	ANSI/ISO Equivalent
from tab_a a, tab_b b where a.col_1 (+) = b.col_1	from tab_a a left outer join tab_b b on a.col_1 = b.col_1
from tab_a a, tab_b b where a.col_1 = b.col_1 (+)	from tab_a a right outer join tab_b b on a.col_1 = b.col_1
from tab_a a, tab_b b where a.col_1 (+) = b.col_1 and b.col_2 = 'VALUE'	from tab_a a left outer join tab_b b on a.col_1 = b.col_1 and b.col_2 = 'VALUE'

UNION & INTERSECT

UNION Query

The UNION query allows you to combine the result sets of 2 or more "select" queries.
It removes duplicate rows between the various "select" statements.
Each SQL statement within the UNION query must have the same number of fields in the result sets with similar data types.

The syntax for a UNION query is:

Select field1, field2, . field_n from tables
UNION
Select field1, field2, . field_n from tables;

Example #1

The following is an example of a UNION query:

```
Select supplier_id from suppliers
UNION
Select supplier_id from orders;
```

In this example, if a supplier_id appeared in both the suppliers and orders table, it would appear once in your result set. The UNION removes duplicates.

Example #2 - With ORDER BY Clause

The following is a UNION query that uses an ORDER BY clause:

```
Select supplier_id, supplier_name from suppliers where supplier_id > 2000
UNION
Select company_id, company_name from companies where company_id > 1000
ORDER BY 2;
```

Since the column names are different between the two "select" statements, it is more advantageous to reference the columns in the ORDER BY clause by their position in the result set. In this example, we've sorted the results by supplier_name / company_name in ascending order, as denoted by the "ORDER BY 2".

UNION ALL Query

The UNION ALL query allows you to combine the result sets of 2 or more "select" queries. It returns all rows (even if the row exists in more than one of the "select" statements).
Each SQL statement within the UNION ALL query must have the same number of fields in the result sets with similar data types.
The syntax for a UNION ALL query is:

```
Select field1, field2, . field_n from tables
UNION ALL
Select field1, field2, . field_n from tables;
```

Example #1
The following is an example of a UNION ALL query:

```
Select supplier_id from suppliers
UNION ALL
Select supplier_id from orders;
```

If a supplier_id appeared in both the suppliers and orders table, it would appear multiple times in your result set. The UNION ALL does not remove duplicates.

Example #2 - With ORDER BY Clause
The following is a UNION query that uses an ORDER BY clause:

```
Select supplier_id, supplier_name from suppliers where supplier_id > 2000
UNION ALL
Select company_id, company_name from companies where company_id > 1000
ORDER BY 2;
```

Since the column names are different between the two "select" statements, it is more advantageous to reference the columns in the ORDER BY clause by their position in the result set. In this example, we've sorted the results by supplier_name / company_name in ascending order, as denoted by the "ORDER BY 2".

INTERSECT Query

The INTERSECT query allows you to return the results of 2 or more "select" queries. However, it only returns the rows selected by all queries. If a record exists in one query and not in the other, it will be omitted from the INTERSECT results.
Each SQL statement within the INTERSECT query must have the same number of fields in the result sets with similar data types.
The syntax for an INTERSECT query is:

```
Select field1, field2, . field_n from tables
INTERSECT
Select field1, field2, . field_n from tables;
```

Example #1
The following is an example of an INTERSECT query:

```
Select supplier_id from suppliers
INTERSECT
```

Select supplier_id from orders;
In this example, if a supplier_id appeared in both the suppliers and orders table, it would appear in your result set.

Example #2 - With ORDER BY Clause
The following is an INTERSECT query that uses an ORDER BY clause:
Select supplier_id, supplier_name from suppliers where supplier_id > 2000

INTERSECT
Select company_id, company_name from companies where company_id > 1000
ORDER BY 2;
Since the column names are different between the two "select" statements, it is more advantageous to reference the columns in the ORDER BY clause by their position in the result set. In this example, we've sorted the results by supplier_name / company_name in ascending order, as denoted by the "ORDER BY 2".

MINUS Query
The MINUS query returns all rows in the first query that are not returned in the second query.
Each SQL statement within the MINUS query must have the same number of fields in the result sets with similar data types.
The syntax for an MINUS query is:
Select field1, field2, . field_n from tables
MINUS
Select field1, field2, . field_n from tables;

Example #1
The following is an example of an MINUS query:
Select supplier_id from suppliers
MINUS
Select supplier_id from orders;
In this example, the SQL would return all supplier_id values that are in the suppliers table and not in the orders table. What this means is that if a supplier_id value existed in the suppliers table and also existed in the orders table, the supplier_id value would not appear in this result set.

Example #2 - With ORDER BY Clause
The following is an MINUS query that uses an ORDER BY clause:
Select supplier_id, supplier_name from suppliers where supplier_id > 2000
MINUS
Select company_id, company_name from companies where company_id > 1000
ORDER BY 2;
Since the column names are different between the two "select" statements, it is more advantageous to reference the columns in the ORDER BY clause by their position in the result

set. In this example, we've sorted the results by supplier_name / company_name in ascending order, as denoted by the "ORDER BY 2".

Subqueries

Subquery or Inner query or Nested query is a query in a query. A subquery is usually added in the WHERE Clause of the sql statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value.

```
select ename, deptno, sal
 2  from emp
 3  where deptno =
 4   ( select deptno
 5     from dept
 6     where loc = 'NEW YORK' );

ENAME      DEPTNO      SAL
---------- ---------- ---------
CLARK          10       2450
KING           10       5000
MILLER         10       1300
```

Subqueries can be used to obtain values for parent select statements when specific search criteria isn't known. To do so, the where clause in the parent select statement must have a comparison operation where the unknown value being compared is determined by the result of the subquery. The inner subquery executes once, right before the main outer query executes. The subquery returns its results to the main outer query as shown in above example

Notes:

1. Subqueries must appear inside parentheses, or else Oracle will have trouble distinguishing the subquery from the parent query. You should also make sure to place subqueries on the right side of the comparison operator.
2. Subqueries are an alternate way of returning data from multiple tables.
3. Subqueries can be used with the following sql statements along with the comparision operators like =, <, >, >=, <= etc.
   - SELECT
   - INSERT
   - UPDATE
   - DELETE

Differnt Usage

IN

You can also use the in comparison, which is similar to the case statement offered in many programming languages, because resolution can be established based on the parent column's equality with any element in the group. Let's take a look at an example:

```
SQL> select ename, job, sal
  2  from emp
  3  where deptno in
  4   ( select deptno
  5     from dept
  6     where dname in
  7     ('ACCOUNTING', 'SALES'));
```

EXISTS/NOT EXISTS

Another way of including a subquery in the where clause of a select statement is to use the exists clause. This clause enables you to test for the existence of rows in the results of a subquery, and its logical opposite is not exists. When you specify the exists operation in a where clause, you must include a subquery that satisfies the exists operation. If the subquery returns data, the exists operation returns TRUE, and a record from the parent query will be returned. If not, the exists operation returns FALSE, and no record for the parent query will be returned. Let's look at an example in which we obtain the same listing of employees working in the New York office, only this time, we use the exists operation:

```
SQL> select e.ename, e.job, e.sal
  2  from emp e
  3  where exists
  4    ( select d.deptno
  5      from dept d
  6      where d.loc = 'NEW YORK'
  7      and d.deptno = e.deptno);
ENAME      JOB           SAL
----------  ---------  ---------
CLARK      MANAGER       2450
KING       PRESIDENT     5000
MILLER     CLERK         1300
```

Correlated Subquery

A query is called correlated subquery when both the inner query and the outer query are interdependent. For every row processed by the inner query, the outer query is processed as well. The inner query depends on the outer query before it can be processed.

```
SELECT p.product_name FROM product p
WHERE p.product_id = (SELECT o.product_id FROM order_items o
WHERE o.product_id = p.product_id);
```

Listing and Writing Different Types of Subqueries

The following list identifies several different types of subqueries you may need to understand and use on the OCP exam:

Single-row subqueries The main query expects the subquery to return only one value.

Multirow subqueries The main query can handle situations where the subquery returns more than one value.

Multiple-column subqueries A subquery that contains more than one column of return data in addition to however many rows are given in the output. These types of subqueries will be discussed later in the chapter.

Inline views A subquery in a from clause used for defining an intermediate result set to query from. These types of subqueries will be discussed later in the chapter.

Single-Row Subqueries
The main query expects the sub query to return only one value.
Check out the following example, which should look familiar:

```
SQL>  select ename, deptno, sal
  2   from emp
  3   where deptno =
  4    ( select deptno
  5      from dept
  6      where loc = 'NEW YORK' );
ENAME       DEPTNO      SAL
----------  ---------  ---------
CLARK          10         2450
KING           10         5000
MILLER         10         1300
```

Though the above query results have 3 rows it is a single-row subquery Because, the subquery on the DEPT table to derive the output from EMP returns only one row of data.
Multi row subquery

A multi row subquery returns one or more rows. Since it returns multiple values, the query must use the set comparison operators (IN,ALL,ANY). If you use a multi row subquery with the equals comparison operators, the database will return an error if more than one row is returned.

Exampe:

```
select last_name from employees where manager_id in
(select employee_id from employees where department_id in
(select department_id from departments where location_id in
(select location_id from locations where country_id='UK')));
```

with
You can improve the performance of this query by having Oracle9i execute the subquery only once, then simply letting Oracle9i reference it at the appropriate points in the main query. The following code block gives a better logical idea of the work Oracle must perform to give you the result. In it, the bold text represents the common parts of the subquery that are performed only once, and the places where the subquery is referenced:

```
SQL> with summary as
```

```
 2  (select dname, sum(sal) as dept_total
 3   from emp, dept
 4   where emp.deptno = dept.deptno
 5   group by dname)
 6  select dname, dept_total
 7  from summary
 8  where dept_total >
 9  (select sum(dept_total) * 1/3
10   from summary)
11  order by dept_total desc;

DNAME            DEPT_TOTAL
-------------------- ----------
RESEARCH              10875
```

Multiple-Column Subqueries

Notice that in all the prior examples, regardless of whether one row or multiple rows were returned from the sub query, each of those rows contained only one column's worth of data to compare at the main query level. The main query can be set up to handle multiple columns in each row returned, too. To evaluate how to use multiple-column sub queries, let's consider an example

```
Select *
From PO_LINES_ALL
Where (PO_HEADER_ID, PO_LINE_ID) IN
(
Select PO_HEADER_ID, PO_LINE_ID
From PO_LINE_LOCATIONS_ALL
WHERE QUANTITY_RECEIVED < QUANTITY/2
AND CLOSED_CODE <> 'CLOSED FOR RECEIVING'
)
```

The benefit of writing query in above format is that separating the requirements in tables. From PO_LINE_LOCATIONS_ALL we are only taking those data which are relevant for our purpose and our end aim is to view the PO_LINEA_ALL entries corresponding to some required conditions satisfied by entries in PO_LINE_LOCATIONS_AL

Inline view : Subqueries in a from Clause

You can also write subqueries that appear in your from clause. Writing subqueries in the from clause of the main query can be a handy way to collect an intermediate set of data that the main query treats as a table for its own query-access purposes. This subquery in the from clause of your main query is called an inline view. You must enclose the query text for the inline view in parentheses and also give a label for the inline view so that columns in it can be referenced later. The subquery can be a select statement that utilizes joins, the group by clause, or the order by clause

```
Select a.PO_HEADER_ID, a.Segment1, b.unit_price, b.Quantity
From PO_HEADERS_ALL a,
(
Select unit_price, Quantity, po_header_id
From PO_LINES_ALL
) b
Where a.PO_HEADER_ID=b.PO_HEADER_ID
```

Inline Views and Top-N Queries

Top-N queries use inline views and are handy for displaying a short list of table data, based on "greatest" or "least" criteria. For example, let's say that profits for our company were exceptionally strong this year, and we want a list of the three lowest-paid employees in our company so that we could give them a raise. A top-N query would be useful for this purpose. Take a look at a top-N query that satisfies this business scenario:

```
SQL> select ename, job, sal, rownum
  2  from (select ename, job, sal from emp
  3        order by sal)
  4  where rownum <=3;
ENAME      JOB           SAL     ROWNUM
----------  ---------     ---------  ---------
SMITH      CLERK          800        1
JAMES      CLERK          950        2
ADAMS      CLERK          1100       3
```

You need to know two important things about top-N queries for OCP. The first is their use of the inline view to list all data in the table in sorted order. The second is their use of ROWNUM—a virtual column identifying the row number in the table—to determine the top number of rows to return as output. Conversely, if we have to cut salaries based on poor company performance and want to obtain a listing of the highest-paid employees, whose salaries will be cut, we would reverse the sort order inside the inline view, as shown here:

```
SQL> select ename, job, sal, rownum
  2  from (select ename, job, sal from emp
  3        order by sal desc)
  4  where rownum <=3;
ENAME      JOB           SAL    ROWNUM
---------- --------- --------- ---------
KING       PRESIDENT     5000       1
SCOTT      ANALYST       3000       2
FORD       ANALYST       3000       3
```

DML Statements

Data Manipulation Language (DML) is a family of computer languages used by computer programs database users to retrieve, insert, delete and update data in a database.

Currently the most popular data manipulation language is that of SQL, which is used to retrieve and manipulate data in a Relational database. Other forms of DML are those used by IMS/DLI, CODASYL databases (such as IDMS), and others.
Data manipulation languages were initially only used by computer programs, but (with the advent of SQL) have come to be used by people, as well.

Data Manipulation Language (DML) is used to retrieve, insert and modify database information. These commands will be used by all database users during the routine operation of the database. Let's take a brief look at the basic DML commands:
Data Manipulation Languages have their functional capability organized by the initial word in a statement, which is almost always a verb. In the case of SQL, these verbs are:

* Select
* Insert
* Update
* Delete

INSERT Statement
The INSERT statement allows you to insert a single record or multiple records into a table.
The general syntax for an insert statement is insert into tablename (column_list) values (valuesl_list), where tablename is the
name of the table you want to insert data into, column_list is the list of columns for which you will define values on the record being added, and values_list is the list of those values you will define. The datatype of the data you add as values in the values list must correspond to the datatype for the column identified in that same position in the column list.
The syntax for the INSERT statement is:

```
INSERT INTO table_name
(column-1, column-2, ... column-n) VALUES (value-1, value-2, ... value-n);
```

Example 1:

```
INSERT INTO  XX_PO_HEADERS_ALL
(PO_ID, PO_NUMBER, SUPPLIER_NAME) VALUES(6, 10, 'ARCODA')
```

Example 2:
you may not necessarily need to define explicit columns of the table. You only need to do that when you don't plan to populate every column in the record you are inserting with a value.

```
insert into employee
values ('02039','WALLA','RAJENDRA',60000,'01-JAN-96','604B');
```

Example 3:

```
INSERT INTO suppliers
(supplier_id, supplier_name) SELECT account_no, name FROM customers WHERE city =
'Newark';
```

Example 4:
The following is an example of how you might insert 3 rows into the *suppliers* table in Oracle.

```
INSERT ALL
INTO  XX_PO_HEADERS_ALL(PO_ID, PO_NUMBER, SUPPLIER_NAME) VALUES(4, 10, 'ARCODA')
INTO  XX_PO_HEADERS_ALL(PO_ID, PO_NUMBER, SUPPLIER_NAME) VALUES(5, 10, 'ARCODA')
Select * from dual
```

UPDATE Statement
Data manipulation on Oracle tables does not end after you add new records to your tables. Often, the rows in a table will need to be changed. In order to make those changes, the update statement can be used.
The UPDATE statement allows you to update a single record or multiple records in a table.
The syntax for the UPDATE statement is:

```
UPDATE table
SET column = expression
WHERE predicates;
```

Example #1 - Simple example
Let's take a look at a very simple example.

```
UPDATE suppliers
SET name = 'HP'
WHERE name = 'IBM';
```

This statement would update all supplier names in the suppliers table from IBM to HP.

Example #2 - More complex example
You can also perform more complicated updates.
You may wish to update records in one table based on values in another table. Since you can't list more than one table in the UPDATE statement, you can use the EXISTS clause.
For example:

```
UPDATE suppliers
SET supplier_name =     ( SELECT customers.name
FROM customers
WHERE customers.customer_id = suppliers.supplier_id)
WHERE EXISTS
 ( SELECT customers.name
   FROM customers
   WHERE customers.customer_id = suppliers.supplier_id);
```

Whenever a supplier_id matched a customer_id value, the supplier_name would be overwritten to the customer name from the customers table.

DELETE Statement
The DELETE statement allows you to delete a single record or multiple records from a table.

The syntax for the DELETE statement is:

```
DELETE FROM table
WHERE predicates;
```

Example #1 : Simple example
Let's take a look at a simple example:

```
DELETE FROM suppliers
WHERE supplier_name = 'IBM';
```

This would delete all records from the suppliers table where the supplier_name is IBM. You may wish to check for the number of rows that will be deleted. You can determine the number of rows that will be deleted by running the following SQL statement before performing the delete.

Example #2 : More complex example
You can also perform more complicated deletes.
You may wish to delete records in one table based on values in another table. Since you can't list more than one table in the FROM clause when you are performing a delete, you can use the EXISTS clause.
For example:

```
DELETE FROM suppliers
WHERE EXISTS
 ( select customers.name
   from customers
   where customers.customer_id = suppliers.supplier_id
   and customers.customer_name = 'IBM' );
```

This would delete all records in the suppliers table where there is a record in the customers table whose name is IBM, and the customer_id is the same as the supplier_id.

Merge into
The merge command syntax is

```
merge into table1
using table2 on (join_condition)
when matched update set col1 = value
when not matched insert (column_list) values (column_values).
```

The statement components work in the following way:
1. In the merge into table1 clause, you identify a table into which you would like to update data in an existing row or add new data if the row doesn't already exist as table1.

2. In the using table2 clause, you identify a second table from which rows will be drawn in order to determine if the data already exists as table2. This can be a different table or the same table as table1. However, if table2 is the same table as table1, or if the two tables have similar columns, then you must use table aliases to preface all column references with the correct copy of the table. Otherwise, Oracle will return an error stating that your column references are

ambiguously defined.
In the on (join_condition) clause, you define the join condition to link the two tables together. If table2 in the using clause is the same table as table1 in the merge into clause, or if the two tables have similar columns, then you must use table aliases or the table.column syntax when referencing columns in the join or filter conditions. Otherwise, Oracle will return an error stating that your column references are ambiguously defined.

3. In the when matched then update set col1 = value clause, you define the column(s) Oracle should update in the first table if a match in the second table is found. If table2 in the using clause is the same table as table1 in the merge into clause, or if the two tables have similar columns, then you must use table aliases or the table.column syntax when referencing columns in the update operation. Otherwise, Oracle will return an error stating that your column references are ambiguously defined.

4. In the when not matched then insert (column_list) values (value_list) clause, you define what Oracle should insert into the first table if a match in the second table is not found. If table2 in the using clause is the same table as table1 in the merge into clause, or if the two tables have similar columns, then you must use table aliases or the table.column syntax to preface all column references in column_list. Otherwise, Oracle will return an error stating that your column references are ambiguously defined.

Example
Consider the following scenario. Say you manage a movie theater that is part of a national chain. Everyday, the corporate headquarters sends out a data feed that you put into your digital billboard over the ticket sales office, listing out all the movies being played at that theater, along with showtimes. The showtime information changes daily for existing movies in the feed.

```
merge into movies M1
using movies M2 on (M2.movie_name = M1.movie_name and M1.movie_name = 'GONE
WITH THE WIND')
when matched then update set M1.showtime = '7:30 PM'
when not matched then insert (M1.movie_name, M1.showtime) values ('GONE WITH THE
WIND','7:30 PM');
```

Transaction Control
One of the great benefits Oracle provides you is the ability to make changes in database using SQL statements and then decide later whether we want to save or discard them. Oracle enables you to execute a series of data-change statements together as one logical unit of work, called a transaction, that's terminated when you decide to save or discard the work. A transaction begins with your first executable SQL statement. Some advantages for offering transaction processing in Oracle include the following:
Transactions enable you to ensure read-consistency to the point in time a transaction began for all users in the Oracle database.

Transactions enable you to preview changes before making them permanent in Oracle. Transactions enable you to group logically related SQL statements into one logical unit of work. Transaction processing consists of a set of controls that enable a user issuing an insert, update, or delete statement to declare a beginning to the series of data-change statements he or she will issue. When the user has finished making the changes to the database, the user can save the data to the database by explicitly ending the transaction. Alternatively, if a mistake is made at any point during the transaction, the user can have the database discard the changes made to the database in favor of the way the data existed before the transaction.

The commands that define transactions are as follows:
Set transaction Initiates the beginning of a transaction and sets key features. This command is optional. A transaction will be started automatically when you start SQL*Plus, commit the previous transaction, or roll back the previous transaction.

Commit Ends the current transaction by saving database changes and starts a new transaction.

Rollback Ends the current transaction by discarding database changes and starts a new transaction.

Savepoint Defines breakpoints for the transaction to enable partial rollbacks.

Locks
Set transaction
This command can be used to define the beginning of a transaction. If any change is made to the database after the set transaction command is issued but before the transaction is ended, all changes made will be considered part of that transaction. The set transaction statement is not required, because a transaction begins under the following circumstances:
As soon as you log onto Oracle via SQL*Plus and execute the first command
Immediately after issuing a rollback or commit statement to end a transaction
When the user exits SQL*Plus
When the system crashes
When a data control language command such as alter database is issued

By default, a transaction will provide both read and write access unless you override this default by issuing set transaction read only. You can set the transaction isolation level with set transaction as well. The set transaction isolation level serializable command specifies serializable transaction isolation mode as defined in SQL92. If a serializable transaction contains data manipulation language (DML) that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, the DML statement fails. The set transaction isolation level read committed command is the default Oracle transaction behavior. If the transaction contains DML that requires row locks held by another transaction, the DML statement waits until the row locks are released

Commit

The commit statement in transaction processing represents the point in time where the user has made all the changes he or she wants to have logically grouped together, and because no mistakes have been made, the user is ready to save the work. The work keyword is an extraneous word in the commit syntax that is designed for readability.
Issuing a commit statement also implicitly begins a new transaction on the database because it closes the current transaction and starts a new one. By issuing a commit, data changes are made permanent in the database. The previous state of the data is lost. All users can view the data, and all savepoints are erased. It is important also to understand that an implicit commit occurs on the database when a user exits SQL*Plus or issues a data-definition language (DDL) command such as a create table statement, used to create a database object, or an alter table statement, used to alter a database object.

The following is an example:

```
SQL> COMMIT;
Commit complete.
SQL> COMMIT WORK;
Commit complete.
```

Rollback

If you have at any point issued a data-change statement you don't want, you can discard the changes made to the database with the use of the rollback statement. The previous state of the data is restored. Locks on the affected rows are released. After the rollback command is issued, a new transaction is started implicitly by the database session. In addition to rollbacks executed when the rollback statement is issued, implicit rollback statements are conducted when a statement fails for any reason or if the user cancels a statement with the CTRL-C cancel command. The following is an example:

```
SQL> ROLLBACK;
Rollback complete
```

Savepoint

In some cases involving long transactions or transactions that involve many data changes, you may not want to scrap all your changes simply because the last statement issued contains unwanted changes. Savepoints are special operations that enable you to divide the work of a transaction into different segments. You can execute rollbacks to the savepoint only, leaving prior changes intact. Savepoints are great for situations where part of the transaction needs to be recovered in an uncommitted transaction. At the point the rollback to savepoint so_far_so_good statement completes in the following code block, only changes made before the savepoint was defined are kept when the commit statement is issued:

```
UPDATE products
SET quantity = 55
WHERE product# = 59495;
SAVEPOINT so_far_so_good;
//Savepoint created.
```

```
UPDATE spanky.products
SET quantity = 504;
ROLLBACK TO SAVEPOINT so_far_so_good;
COMMIT;
```

Locks

The final aspect of the Oracle database that enables the user to employ transaction processing is the lock, the mechanism by which Oracle prevents data from being changed by more than one user at a time. Several different types of locks are available, each with its own level of scope. Locks available on a database are categorized into table-level locks and row-level locks.
A table-level lock enables only the user holding the lock to change any piece of row data in the table, during which time no other users can make changes anywhere on the table. A table lock can be held in any of several modes: row share (RS), row exclusive (RX), share (S), share row exclusive (SRX), and exclusive (X). The restrictiveness of a table lock's mode determines the modes in which other table locks on the same table can be obtained and held.

A row-level lock gives the user the exclusive ability to change data in one or more rows of the table. However, any row in the table that is not held by the row-level lock can be changed by another user

Tip

An update statement acquires a special row-level lock called a row-exclusive lock, which means that for the period of time the update statement is executing, no other user in the database can view or change the data in the row. The same goes for delete or insert operations. Another update statement—the select for update statement—acquires a more lenient lock called the share row lock. This lock means that for the period of time the update statement is changing the data in the rows of the table, no other user may change that row, but users may look at the data in the row as it changes.

Other Database Objects

Some of the objects that are part of the relational database produced by Oracle and that are used in the functions just mentioned are as follows:
Tables, views, and synonyms Used to store and access data. Tables are the basic unit of storage in Oracle. Views logically represent subsets of data from one or more tables. Synonyms provide alternate names for database objects.

Indexes and the Oracle RDBMS Used to speed access to data.

Sequences Used for generating numbers for various purposes.

Triggers and integrity constraints Used to maintain the validity of data entered.
Privileges, roles, and profiles Used to manage database access and usage.
Packages, procedures, and functions Application PL/SQL code used in the database.

Using Public and Private Synonyms

The objects in Oracle you create are available only in your schema unless you grant access to the objects explicitly to other users. We'll discuss privileges and user access in the next section. However, even when you grant permission to other users for using an object, the boundary created by schema ownership will force other users to prefix the object name with your schema name in order to access your object. For example, SCOTT owns the EMP table. If TURNER wants to access SCOTT's EMP table, he must refer to EMP as SCOTT.EMP. If TURNER doesn't, the following happens:

*SELECT * FROM emp WHERE empno = 7844;*

So, TURNER can't even see his own employee data—in fact, Oracle tells him that the EMP table doesn't even exist (pretty sneaky, eh?). Yet, as soon as TURNER prefixes the EMP table with its schema owner, SCOTT, a whole world of data opens up for TURNER, as you can see in the following code block:

SELECT empno, ename, sal FROM SCOTT.emp 2 WHERE empno = 7844;

If remembering which user owns which table seems unnecessarily complicated, synonyms can be used on the database for schema transparency. Synonyms are alternative names that can be created as database objects in Oracle to refer to a table or view. You can refer to a table owned by another user using synonyms. Creating a synonym eliminates the need to qualify the object name with the schema and provides you with an alternative name for a table, view, sequence, procedure, or other objects. Synonyms are also used to shorten lengthy object names.

Two types of synonyms exist in Oracle: private synonyms and public synonyms. You can use a private synonym within your own schema to refer to a table or view by an alternative name. Private synonyms are exactly that—they are private to your schema and therefore usable only by you. A private synonym name must be distinct from all other objects owned by the same user.

Think of private synonyms as giving you the ability to develop "pet names" for database objects in Oracle. You can use public synonyms to enable all users in Oracle to access database objects you own without having to prefix the object names with your schema name. This concept of referencing database objects without worrying about the schema the objects are part of is known as schema transparency. Public synonyms are publicly available to all users of Oracle; however, you need special privileges to create public synonyms. We'll talk more about the privilege required for creating public synonyms in the next section. For now, the following code block demonstrates how to create private and public synonyms, respectively:

```
create synonym all_my_emps for emp;
```

Tip

Synonyms do not give you access to data in a table that you do not already have access to. Only privileges can do that. Synonyms simply enable you to refer to a table without prefixing the schema name to the table reference. When resolving a database table name, Oracle looks first to see whether the table exists in your schema. If Oracle doesn't find the table, Oracle searches for a private synonym. If none is found, Oracle looks for a public synonym.

Drop Synonyms

Synonyms are dropped using the drop synonym command, as shown in the following code block:

```
Drop synonym emp;
```

Sequences

A sequence is a database object that generates integers according to rules specified at the time the sequence is created. A sequence automatically generates unique numbers and is sharable between different users in Oracle. Sequences have many purposes in database systems—the most common of which is to generate primary keys automatically. However, nothing binds a sequence to a table's primary key, so in a sense it's also a sharable object

Sequences are created with the create sequence statement

```
CREATE SEQUENCE
START WITH
INCREMENT BY
MINVALUE
MAXVALUE
CYCLE
ORDER
CACHE
```

1. Start with n Enables the creator of the sequence to specify the first value generated by the sequence. Once created, the sequence will generate the value specified by start with the first time the sequence's NEXTVAL virtual column is referenced. If no start with value is specified, Oracle defaults to a start value of 1.

2. Increment by n Defines the number by which to increment the sequence every time the NEXTVAL virtual column is referenced. The default for this clause is 1 if it is not explicitly specified. You can set n to be positive for incrementing sequences or negative for decrementing or countdown sequences.

3. Minvalue n Defines the minimum value that can be produced by the sequence. If no minimum value is specified, Oracle will assume the default, nominvalue.

4. Maxvalue n Defines the maximum value that can be produced by the sequence. If no maximum value is desired or specified, Oracle will assume the default, nomaxvalue.

5. Cycle Enables the sequence to recycle values produced when maxvalue or minvalue is reached. If cycling is not desired or not explicitly specified, Oracle will assume the default, nocycle. You cannot specify cycle in conjunction with nomaxvalue or nominvalue. If you want your sequence to cycle, you must specify maxvalue for incrementing sequences or minvalue for decrementing or countdown sequences.

6. Cache n Enables the sequence to cache a specified number of values to improve performance. If caching is not desired or not explicitly specified, Oracle will assume the default, which is to cache 20 values.

7. Order Enables the sequence to assign values in the order in which requests are received by the sequence. If order is not desired or not explicitly specified, Oracle will assume the default, noorder.

Example 1:

```
CREATE SEQUENCE supplier_seq
   MINVALUE 1
   MAXVALUE 9999999999999999999999999999
   START WITH 1
   INCREMENT BY 1
   CACHE 20;
```

This would create a sequence object called *supplier_seq*. The first sequence number that it would use is 1 and each subsequent number would increment by 1 (ie: 2,3,4,...}. It will cache up to 20 values for performance.

Example 2:

The below sequence is a dercrment one. It starts with 100 and decreases by 1.

```
CREATE SEQUENCE XX_Notification_number
START WITH 100
INCREMENT BY -1
MAXVALUE 100
MINVALUE 1
CYCLE
CACHE 20
```

Referencing Sequences in Data Changes

Sequence-value generation can be incorporated directly into data changes made by insert and update statements. This direct use of sequences in insert and update statements is the most common use for sequences in a database. In the situation where the sequence generates a primary key for all new rows entering the database table, the sequence would likely be referenced directly from the insert statement. Note, however, that this approach sometimes fails when the sequence is referenced by triggers. Therefore, it is best to reference sequences within the user interface or within stored procedures. The following statements illustrate the use of sequences directly in changes made to tables:

```
INSERT INTO expense(expense_no, empid, amt, submit_date)
VALUES(countdown_20.nextval, 59495, 456.34, '21-NOV-99');
```

SEQUENCE_NAME.NEXTVAL & SEQUENCE_NAME.CURRVAL

Once the sequence is created, it is referenced using the CURRVAL and NEXTVAL pseudocolumns. The users of the database can view the current value of the sequence by using a select statement. Similarly, the next value in the sequence can be generated with a select statement. Because sequences are not tables—they are only objects that generate integers via the

use of virtual columns—the DUAL table acts as the "virtual" table from which the virtual column data is pulled. As stated earlier, values cannot be placed into the sequence; instead, they can only be selected from the sequen

Example 3:

```
Select XX_Notification_number.NEXTVAL from dual
Select XX_Notification_number.CURRVAL from dual
```

Alter sequence

The time may come when the sequence of a database will need its rules altered in some way. For example, you may want sequence XX_Notification_number to decrement by a different number. Any parameter of a sequence can be modified by issuing the alter sequence statement. The following is an example:

```
Alter Sequence sequence_name
//Write new values of the sequence parameters
START WITH 100
INCREMENT BY -1
MAXVALUE 100
MINVALUE 1
CYCLE
CACHE 20
```

Example 4:

```
alter sequence XX_Notification_number
increment by -2;
```

Index

An index can be created in a table to find data more quickly and efficiently. The users cannot see the indexes, they are just used to speed up searches/queries.

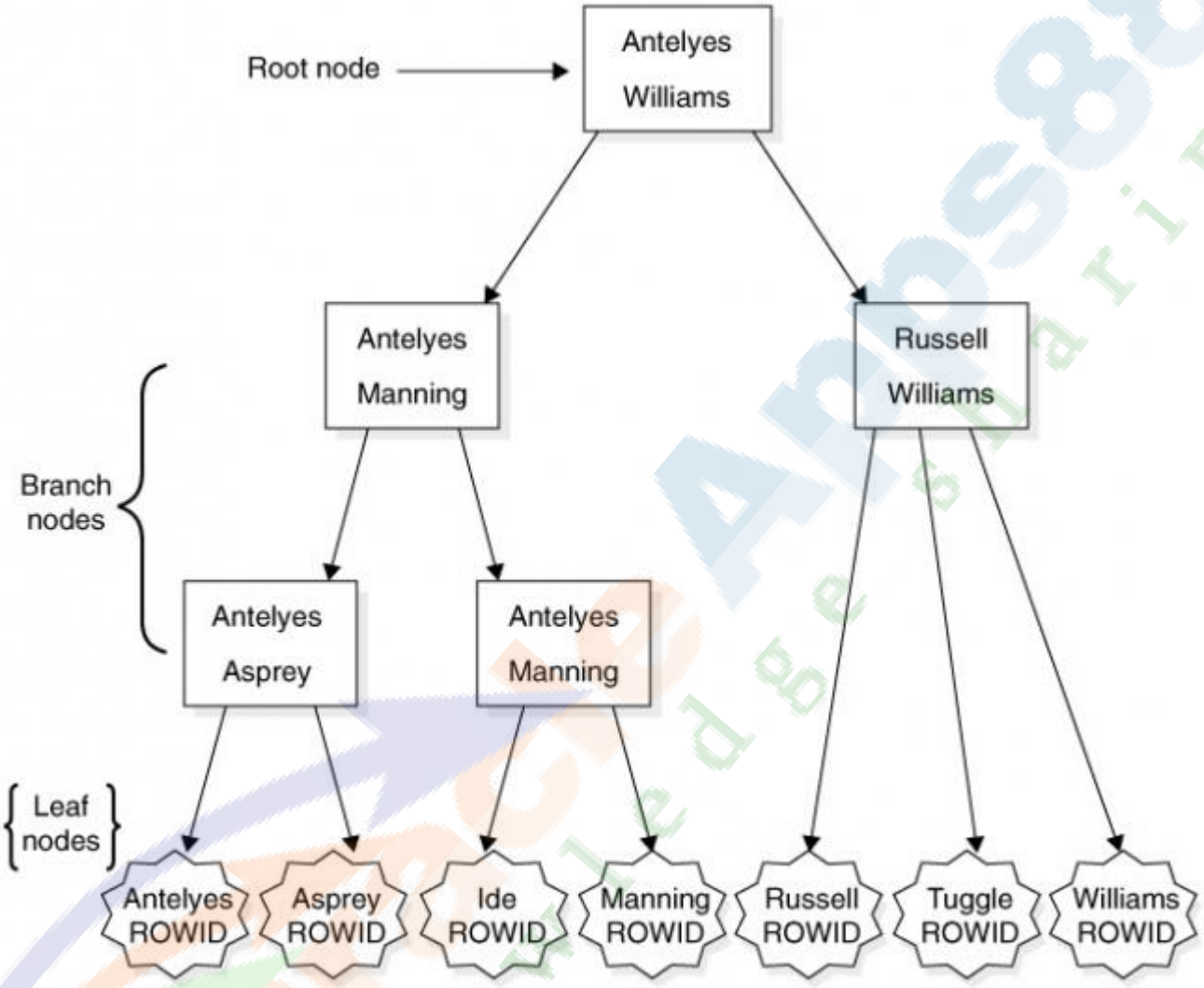
Indexes are objects in the database that provide a mapping of all the values in a table column, along with the ROWID(s) for all rows in the table that contain that value for the column. A ROWID is a unique identifier for a row in an Oracle database table. Indexes have multiple uses on the Oracle database. Indexes can be used to ensure uniqueness on a database, and they can also boost performance when you're searching for records in a table. Indexes are used by the Oracle Server to speed up the retrieval of rows by using a pointer. The improvement in performance is gained when the search criteria for data in a table include a reference to the indexed column or columns.

In Oracle, indexes can be created on any column in a table except for columns of the LONG datatype. Especially on large tables, indexes make the difference between an application that drags its heels and an application that runs with efficiency. However, many performance considerations must be weighed before you make the decision to create an index.

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So you should only create indexes on columns (and tables) that will be frequently searched against.

B-tree Index Structure

The traditional index in the Oracle database is based on a highly advanced algorithm for sorting data called a B-tree. A B-tree contains data placed in layered, branching order, from top to bottom, resembling an upside-down tree. The midpoint of the entire list is placed at the top of the "tree" and is called the root node. The midpoints of each half of the remaining two lists are placed at the next level, and so on



By using a divide-and-conquer method for structuring and searching for data, the values of a column are only a few hops away on the tree, rather than several thousand sequential reads through the list away. However, traditional indexes work best when many distinct values are in the column or when the column is unique.

The algorithm works as follows:

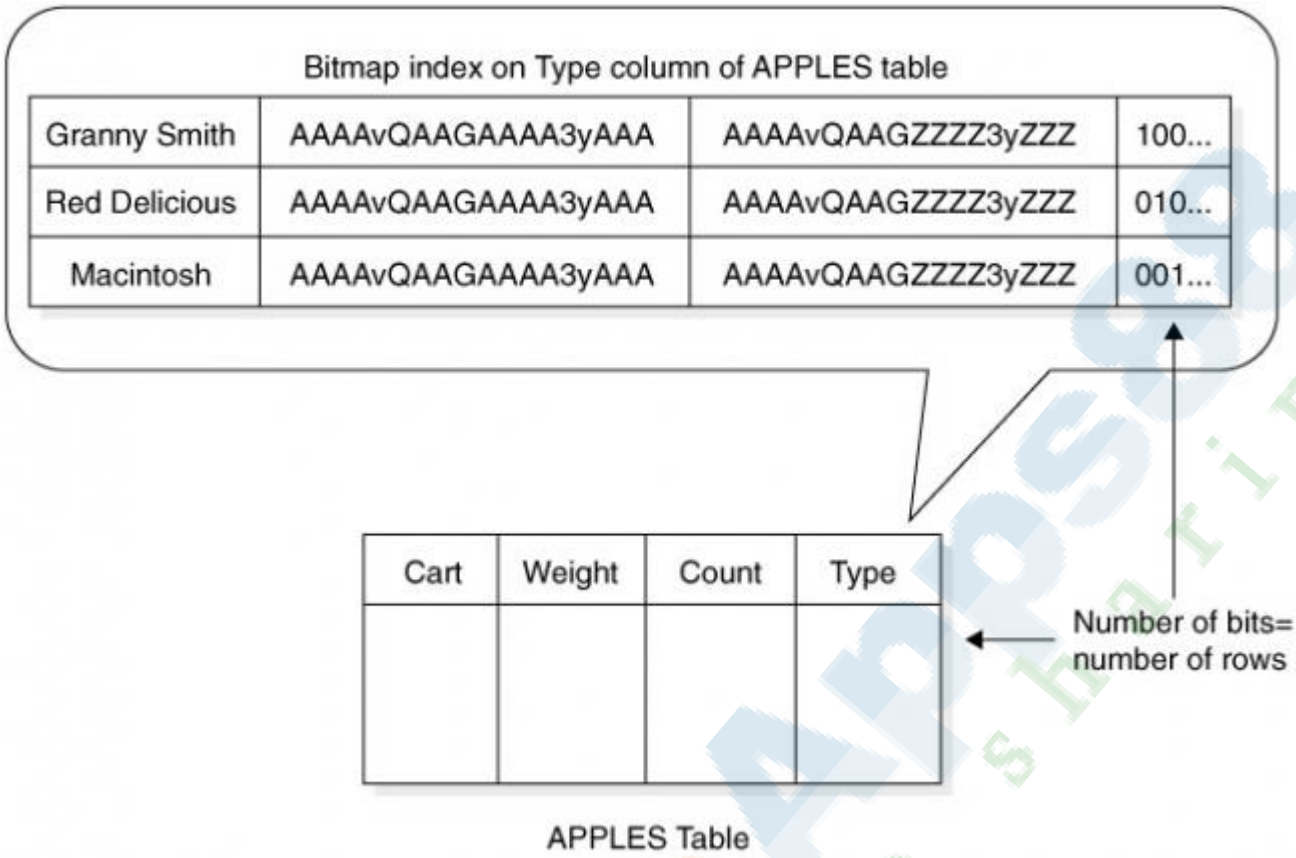
1. Compare the given value to the value in the halfway point of the list. If the value at hand is greater, discard the lower half of the list. If the value at hand is less, discard the upper half of the list.
2. Repeat step 1 for the remaining part of the list until a value is found or the list exhausted.

Along with the data values of a column, each individual node of an index also stores a piece of information about the column value's row location on disk. This crucial piece of lookup data is called a ROWID. The ROWID for the column value points Oracle directly to the disk location of the table row corresponding to the column value. A ROWID identifies the location of a row in a data block in the datafile on disk. With this information, Oracle can then find all the data associated with the row in the table.

Tip: The ROWID for a table is an address for the row on disk. With the ROWID, Oracle can find the data on disk rapidly.

Bitmap Index Structure

This topic is pretty advanced, so consider yourself forewarned. The other type of index available in Oracle is the bitmap index. Try to conceptualize a bitmap index as being a sophisticated lookup table, having rows that correspond to all unique data values in the column being indexed. Therefore, if the indexed column contains only three distinct values, the bitmap index can be visualized as containing three rows. Each row in a bitmap index contains four columns. The first column contains the unique value for the column being indexed. The next column contains the start ROWID for all rows in the table. The third column in the bitmap index contains the end ROWID for all rows in the table. The last column contains a bitmap pattern, in which every row in the table will have one bit. Therefore, if the table being indexed contains 1,000 rows, this last column of the bitmap index will have 1,000 corresponding bits in this last column of the bitmap index. Each bit in the bitmap index will be set to 0 (off) or 1 (on), depending on whether the corresponding row in the table has that distinct value for the column. In other words, if the value in the indexed column for that row matches this unique value, the bit is set to 1; otherwise, the bit is set to 0. Figure 7-2 displays a pictorial representation of a bitmap index containing three distinct values.



Each row in the table being indexed adds only a bit to the size of the bitmap pattern column for the bitmap index, so growth of the table won't affect the size of the bitmap index too much. However, each distinct value adds another row to the bitmap index, which adds another entire bitmap pattern with one bit for each row in the table. Be careful about adding distinct values to a column with a bitmap index, because these indexes work better when few distinct values are allowed for a column. The classic example of using a bitmap index is where you want to query a table containing employees based on a GENDER column, indicating whether the employee is male or female. This information rarely changes about a person, and only two distinct possibilities, so a traditional B-tree index is not useful in this case. However, this is exactly the condition where a bitmap index would aid performance. Therefore, the bitmap index improves performance in situations where traditional indexes are not useful, and vice versa.
Tip : Up to 32 columns from one table can be included in a single B-tree index on that table, whereas a bitmap index can include a maximum of 30 columns from the table.

CREATE INDEX

You can create a unique B-tree index on a column manually by using the create index name on table (column) statement containing the unique keyword. This process is the manual equivalent of creating a unique or primary key constraint on a table. (Remember, unique indexes are created automatically in support of those constraints.)
Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column_name)
```

Creates a unique index on a table. Duplicate values are not allowed:
CREATE UNIQUE INDEX index_name
ON table_name (column_name)

Creating Function-Based Indexes

To create a function-based index in your own schema on your own table, you must have the CREATE INDEX and QUERY REWRITE system privileges. To create the index in another schema or on another schema's table, you must have the CREATE ANY INDEX and GLOBAL QUERY REWRITE privileges. The table owner must also have the EXECUTE object privilege on the functions used in the function-based index.
The function-based index is a new type of index in Oracle that is designed to improve query performance by making it possible to define an index that works when your where clause contains operations on columns. Traditional B-tree indexes won't be used when your where clause contains columns that participate in functions or operations. For example, suppose you have table EMP with four columns: EMPID, LASTNAME, FIRSTNAME, and SALARY. The SALARY column has a B-tree index on it. However, if you issue the select * from EMP where (SALARY*1.08) > 63000 statement, the RDBMS will ignore the index, performing a full table scan instead. Function-based indexes are designed to be used in situations like this one, where your SQL statements contain such operations in their where clauses. The following code block shows a function-based index defined:

```
SQL> CREATE INDEX ixd_emp_01
  2  ON emp(SAL*1.08);
```

By using function-based indexes like this one, you can optimize the performance of queries containing function operations on columns in the where clause, like the query shown previously. As long as the function you specify is repeatable, you can create a function-based index around i

DROP Indexes

When an index is no longer needed in the database, the developer can remove it with the drop index command. Once an index is dropped, it will no longer improve performance on searches using the column or columns contained in the index. No mention of that index will appear in the data dictionary any more either. You cannot drop the index that is used for a primary key. The syntax for the drop index statement is the same, regardless of the type of index being dropped (unique, bitmap, or B-tree). If you want to rework the index in any way, you must first drop the old index and then create the new one. The following is an example:
DROP INDEX employee_last_first_indx_01;

Create DATABASE LINK

A database link is a schema object in one database that enables you to access objects on another database. The other database need not be an Oracle Database system. However, to access non-Oracle systems you must use Oracle Heterogeneous Services.

You might want to create a database link, for instance, if you want the data in a remote database updated when the local database is updated. Here's how to accomplish this:
The first thing you need to do is to create a database link pointing to the other location. The database link can be created with a command similar to the following:

```
CREATE [SHARED | PUBLIC] DATABASE LINK <link-name>
[CONNECT TO <username> IDENTIFIED BY <password>]
[USING <connect-string>];
```

To create a fixed user database link, a username and password to connect with must be specified. For example:

```
CREATE PUBLIC DATABASE LINK BASING
CONNECT TO USRCRY IDENTIFIED BY CRYPWD
USING BASING_ORA;
Select count(*) from table@basing;
```

You'll want to give the database link a better name, use the appropriate userid/password to connect to the remote database, and configure your TNSNAMES.ORA file with a TNS alias to point to that database.
User Access Control
In this chapter, you will learn about and demonstrate knowledge in the following areas of user access and privileges in the Oracle database:
Creating users
Granting and revoking object privileges
Using roles to manage database access
The basic Oracle database security model consists of two parts. The first part consists of password authentication for all users of the Oracle database. Password authentication is available either directly from the Oracle server or from the operating system supporting the Oracle database. When Oracle's own authentication system is used, password information is stored in Oracle in an encrypted format. The second part of the Oracle security model consists of controlling which database objects a user may access, the level of access a user may have to these objects, and whether a user has the authority to place new objects into the Oracle database. At a high level, these controls are referred to as privileges. We'll talk about privileges and database access later in this section.

Create Users

The most basic version of the command for creating users defines only the user we want to create, along with a password, as seen in the following example:

```
Create User USERNAME identified by PASSWORD;
create user turner identified by ike;
```

Tip :
The user does not have privileges at this point. The DBA can then grant privileges to the user. The privileges determine the actions that the user can do with the objects in the database. Also, usernames can be up to 30 characters in length and can contain alphanumeric characters as well as the $, #, and _ characters.
User Access Control

In this chapter, you will learn about and demonstrate knowledge in the following areas of user access and privileges in the Oracle database:
Creating users
Granting and revoking object privileges
Using roles to manage database access
The basic Oracle database security model consists of two parts. The first part consists of password authentication for all users of the Oracle database. Password authentication is available either directly from the Oracle server or from the operating system supporting the Oracle database. When Oracle's own authentication system is used, password information is stored in Oracle in an encrypted format. The second part of the Oracle security model consists of controlling which database objects a user may access, the level of access a user may have to these objects, and whether a user has the authority to place new objects into the Oracle database. At a high level, these controls are referred to as privileges. We'll talk about privileges and database access later in this section.

Create Users
The most basic version of the command for creating users defines only the user we want to create, along with a password, as seen in the following example:
Create User USERNAME identified by PASSWORD;
create user turner identified by ike;
Tip :
The user does not have privileges at this point. The DBA can then grant privileges to the user. The privileges determine the actions that the user can do with the objects in the database. Also, usernames can be up to 30 characters in length and can contain alphanumeric characters as well as the $, #, and _ characters.

System/Object Privileges
Privileges are the right to execute particular SQL statements. Two types of privileges exist in Oracle: object privileges and system privileges. *Object privileges* regulate access to database objects in Oracle, such as querying or changing data in tables and views, creating foreign key constraints and indexes on tables, executing PL/SQL programs, and a handful of other activities. *System privileges* govern every other type of activity in Oracle, such as connecting to the database, creating tables, creating sequences, creating views, and much, much more. Privileges are given to users with the **grant** command, and they are taken away with the **revoke** command. The ability to grant privileges to other users in the database rests on users who can administer the privileges. The owner of a database object can administer object privileges related to that object, whereas the DBA administers system privileges

Object privileges (Grant Privileges on Tables)
You can grant users various privileges to tables. These privileges can be any combination of select, insert, update, delete, references, alter, and index. Below is an explanation of what each privilege means.

Privilege	Description
Select	Ability to query the table with a select statement.
Insert	Ability to add new rows to the table with the insert statement.
Update	Ability to update rows in the table with the update statement.
Delete	Ability to delete rows from the table with the delete statement.
References	Ability to create a constraint that refers to the table.
Alter	Ability to change the table definition with the alter table statement.
Index	Ability to create an index on the table with the create index statement.

The syntax for granting privileges on a table is:

```
grant privileges on object to user;
```

For example, if you wanted to grant select, insert, update, and delete privileges on a table called suppliers to a user name smithj, you would execute the following statement:

```
grant select, insert, update, delete on suppliers to smithj;
```

You can also use the all keyword to indicate that you wish all permissions to be granted. For example:

```
grant all on suppliers to smithj;
```

The keyword all can be use as a consolidated method for granting object privileges related to a table. Note that all in this context is not a privilege; it is merely a specification for all object privileges for a database object

If you wanted to grant select access on your table to all users, you could grant the privileges to the public keyword. For example:

```
grant select on suppliers to public;
```

Revoke Privileges on Tables

Once you have granted privileges, you may need to revoke some or all of these privileges. To do this, you can execute a revoke command. You can revoke any combination of select, insert, update, delete, references, alter, and index.

The syntax for revoking privileges on a table is:

```
revoke privileges on object from user;
// Grant this on this to this
```

For example, if you wanted to revoke delete privileges on a table called suppliers from a user named anderson, you would execute the following statement:

```
revoke delete on suppliers from anderson;
```

If you wanted to revoke all privileges on a table, you could use the all keyword. For example:

```
revoke all on suppliers from anderson;
```

If you had granted privileges to public (all users) and you wanted to revoke these privileges, you could execute the following statement:

```
revoke all on suppliers from public;
```

System Privileges

Several categories of system privileges relate to each object. Those categories determine the scope of ability that the privilege grantee will have. The classes or categories of system privileges are listed in this section. Note that in the following subtopics, the privilege itself gives the ability to perform the action against your own database objects, and the any keyword refers to the ability to perform the action against any database object of that type in Oracle.

Database Access These privileges control who accesses the database, when he or she can access it, and what he or she can do regarding management of his or her own session. Privileges include create session, alter session, and restricted session. Users These privileges are used to manage users on the Oracle database. Typically, these privileges are reserved for DBAs or security administrators. Privileges include create user, become user, alter user, and drop user.

Tables You already know that tables store data in the Oracle database. These privileges govern which users can create and maintain tables. The privileges include create table, create any table, alter any table, backup any table, drop any table, lock any table, comment any table, select any table, insert any table, update any table, and delete any table. The create table or create any table privilege also enables you to drop the table. The create table privilege also bestows the ability to create indexes on the table and to run the analyze command on the table. To be able to truncate a table, you must have the drop any table privilege granted to you.

Indexes You already know that indexes are used to improve SQL statement performance on tables containing lots of row data. The privileges include create any index, alter any index, and drop any index. You should note that no create index system privilege exists. The create table privilege also enables you to alter and drop indexes that you own and that are associated with the table.

Synonyms A synonym is a database object that enables you to reference another object by a different name. A public synonym means that the synonym is available to every user in the database for the same purpose. The privileges include create synonym, create any synonym, drop any synonym, create public synonym, and drop public synonym. The create synonym privilege also enables you to alter and drop synonyms that you own.

Views You already know that a view is an object containing a SQL statement that behaves like a table in Oracle, except that it stores no data. The privileges include create view, create any view, and drop any view. The create view privilege also enables you to alter and drop views that you own.

Sequences You already know that a sequence is an object in Oracle that generates numbers according to rules you can define. Privileges include create sequence, create any sequence, alter any sequence, drop any sequence, and select any sequence. The create sequence privilege also enables you to drop sequences that you own.

Roles Roles are objects that can be used for simplified privilege management. You create a role, grant privileges to it, and then grant the role to users. Privileges include create role, drop any

role, grant any role, and alter any role.

Transactions These privileges are for resolving in-doubt distributed transactions being processed on the Oracle database. Privileges include force transaction and force any transaction.

PL/SQL There are many different types of PL/SQL blocks in Oracle. These privileges enable you to create, run, and manage those different types of blocks. Privileges include create procedure, create any procedure, alter any procedure, drop any procedure, and execute any procedure. The create procedure privilege also enables you to alter and drop PL/SQL blocks that you own.

Triggers A trigger is a PL/SQL block in Oracle that executes when a specified DML activity occurs on the table to which the trigger is associated. Privileges include create trigger, create any trigger, alter any trigger, and drop any trigger. The create trigger privilege also enables you to alter and drop triggers that you own.
Examples

```
grant create session to turner;
// Grant this to this
```

```
revoke create session from turner;
// Revoke this from this
```

Open to the Public
Another aspect of privileges and access to the database involves a special user on the database. This user is called PUBLIC. If a system privilege or object privilege is granted to the PUBLIC user, then every user in the database has that privilege. Typically, it is not advised that the DBA should grant many privileges or roles to PUBLIC, because if a privilege or role ever needs to be revoked, then every stored package, procedure, or function will need to be recompiled. Let's take a look:

```
GRANT select, update, insert ON emp TO public;
```

Tip
Roles can be granted to the PUBLIC user as well. We'll talk more about roles in the next discussion.

Granting Object Privileges All at Once
The keyword all can be use as a consolidated method for granting object privileges related to a table. Note that all in this context is not a privilege; it is merely a specification for all object privileges for a database object. The following code block shows how all is used:
GRANT ALL ON emp TO turner;

Giving Administrative Ability along with Privileges

When another user grants you a privilege, you then have the ability to perform whatever task the privilege enables you to do. However, you usually can't grant the privilege to others, nor can you relinquish the privilege without help of the user who granted the privilege to you. If you want some additional power to administer the privilege granted to you, the user who gave you the privilege must also give you administrative control over that privilege. For example, let's say KING now completely trusts TURNER to manage the creation of tables (a system privilege) and wants to give him access to the EMP table (an object privilege). Therefore, KING tells SCOTT to give TURNER administrative control over both these privileges, as follows:

```
GRANT CREATE TABLE TO turner WITH ADMIN OPTION;
GRANT SELECT, UPDATE ON turner TO SPANKY WITH GRANT OPTION;
```

Tip

The GRANT OPTION is not valid when granting an object privilege to a role.

A system privilege or role can be granted with the ADMIN OPTION. A grantee with this option has several expanded capabilities. The grantee can grant or revoke the system privilege or role to or from any user or other role in the database. However, a user cannot revoke a role from himself. The grantee can further grant the system privilege or role with the ADMIN OPTION. The grantee of a role can alter or drop the role.

Finally, if a role is granted using the with admin option clause, the grantee can alter the role or even remove it. You'll learn more about roles in the next discussion.

You can grant INSERT, UPDATE, or REFERENCES privileges on individual columns in a table.

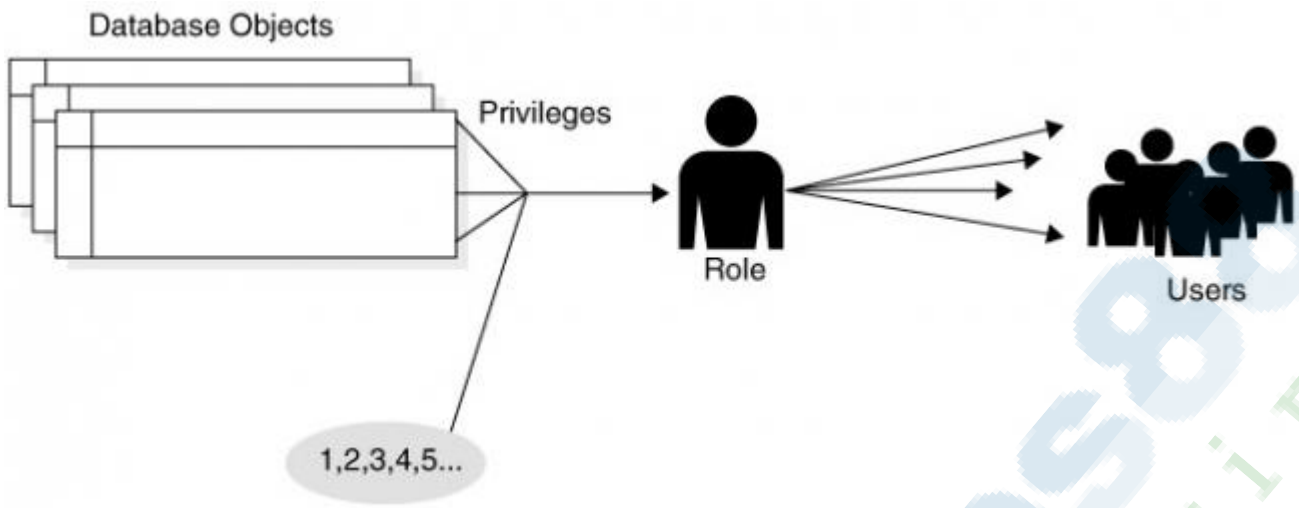
Cascading Effects

No cascading effects of revoking system privileges from users occur.

when an object privilege is revoked from a grantor of that privilege, all grantees receiving the privilege from the grantor also lose the privilege. However, in cases where the object privilege involves update, insert, or delete, if subsequent grantees have made changes to data using the privilege, the rows already changed don't get magically transformed back the way they were before

Using Roles to Manage Database Access

When your databases has lots of tables, object privileges can become unwieldy and hard to manage. You can simplify the management of privileges with the use of a database object called a role. A role acts in two capacities in the database. First, the role can act as a focal point for grouping the privileges to execute certain tasks. Second, the role can act as a "virtual user" of a database, to which all the object privileges required to execute a certain job function can be granted, such as data entry, manager review, batch processing, and so on



Step 1: Design and Crate roles is a process that can happen outside the database. A role mechanism can be used to provide authorization. A single person or a group of people can be granted a role or a group of roles. One role can be granted in turn to other roles. By defining different types of roles, administrators can manage access privileges much more easily. You simply sit down and think to yourself, how many different purposes will users be accessing this application for? Once this step is complete, you can create the different roles required to manage the privileges needed for each job function. Let's say people using the EMP table have two different job roles: those who can query the table and those who can add records to the table. The next step is to create roles that correspond to each activity. This architecture of using roles as a middle layer for granting privileges greatly simplifies administration of user privileges. Let's take a look at how to create the appropriate roles:

```
Create role role_name;
```

Step 2 : Granting Privileges to Roles

The next step is to grant object privileges to roles. You can accomplish this task using the same command as you would use to grant privileges directly to users—the grant command. The following code block demonstrates how to grant privileges to both our roles:

```
Grant select on emp to rpt_writer;
```

Revoking privileges from roles works the same way as it does for users. System privileges are granted and revoked with roles in the exact same way they are with users as well.

Step 3: Granting Roles to Users

Once a role is created and privileges are granted to it, the role can then be granted to users. This step is accomplished with the grant command. Let's see an example of how this is done:

```
Grant rpt_writer to turner;
```

If a role already granted to a user is later granted another privilege, that additional privilege is available to the user immediately. The same statement can be made for privileges revoked from roles already granted to users, too.

Step 4: Revoking and Dropping Roles
Finally, a role can be revoked by the role's owner or by a privileged administrative user using the revoke statement, much like revoking privileges:

```
revoke rpt_writer from turner;
```

Roles can be deleted from the database using the drop role statement. When a role is dropped, the associated privileges are revoked from the users granted the role. The following code block shows how to eliminate roles from Oracle:

```
drop role rpt_writer;
```

Step 5: Modifying Roles if necessary
Let's say that after we create our roles, we realize that changing records in the EMP table is serious business. To create an added safeguard against someone making a change to the EMP table in error, the DATA_CHANGER role can be altered to require a password by using the alter role identified by statement. Anyone wanting to modify data in EMP with the privileges given via the DATA_CHANGER role must first supply a password. Code for altering the role is shown in the following example:

```
Alter role data_changer
Identified by highly#secure;
```

Step 6: Defining User Default Roles
Now user TURNER can execute all the privileges given to him via the RPT_WRITER role, and user FORD can do the same with the privileges given from DATA_CHANGER. Or can he? Recall that we specified that DATA_CHANGER requires a password in order for the grantee to utilize its privileges. Let's make a little change to FORD's user ID so that this status will take effect:

```
alter user ford default role none;
```

You can use the following keywords in the alter user default role command to define default roles for users: all, all except rolename, and none. Note that users usually cannot issue alter user default role themselves to change their default roles—only a privileged user such as the DBA can do it for them.

Step 7: Enabling the Current Role
FORD knows he is supposed to be able to accomplish this task because he has the DATA_CHANGER role. Then he remembers that this role has a password on it. FORD can use the set role command to enable the DATA_CHANGER role in the following way:

```
set role data_changer identified by highly#secure;
```

Now FORD can make the change he needs to make:

```
SQL> insert into scott.emp (empno, ename, job)
  2  values (1234, 'SMITHERS','MANAGER');
```

You must already have been granted the roles that you name in the SET ROLE statement. Also, you can disable all roles with the SET ROLE NONE statement.