IS 2510

Fundamentals of Database Systems

Lab Manual – Oracle SQL

By: Lecturer, Mohamed El Desouki

Contents

SQL: CREATE TABLE Statement

The **SQL CREATE TABLE statement** allows you to create and define a table.

SQL CREATE TABLE Syntax

The syntax for the **SQL CREATE TABLE statement** is:

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

table_name is the name of the table that you wish to create.

column1, *column2* are the columns that you wish to create in the table. 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.

SQL CREATE TABLE Example

Let's look at a SQL CREATE TABLE example.

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50)
);
```

This SQL CREATE TABLE example creates a table called *suppliers* which has 3 columns.

- The first column is called *supplier_id* which is created as a number datatype (maximum 10 digits in length) and can not contain null values.
- The second column is called *supplier_name* which is a varchar2 datatype (50 maximum characters in length) and also can no contain null values.
- The third column is called *contact_name* which is a varchar2 datatype but can contain null values.

Now the only problem with this SQL CREATE TABLE statement is that you have no defined a primary key for the table. We could modify this SQL CREATE TABLE statement and define the supplier_id as the primary key as follows:

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);
```

Practice Exercise #1 - SQL CREATE TABLE Statement:

Create a SQL table called *customers* that stores customer ID, name, and address information.

Solution for Practice Exercise #1 - SQL CREATE TABLE Statement:

The SQL CREATE TABLE statement for the *customers* table is:

```
CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50) not null,
  address varchar2(50),
  city varchar2(50),
  state varchar2(25),
  zip_code varchar2(10)
);
```

Practice Exercise #2 - SQL CREATE TABLE Statement:

Create a SQL table called *customers* that stores customer ID, name, and address information.

But this time, the customer ID should be the primary key for the table.

Solution for Practice Exercise #2 - SQL CREATE TABLE Statement:

The SQL CREATE TABLE statement for the *customers* table is:

```
CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50) not null,
  address varchar2(50),
  city varchar2(50),
  state varchar2(25),
  zip_code varchar2(10),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Practice Exercise #3 - SQL CREATE TABLE Statement:

Based on the *departments* table below, create a SQL table called *employees* that stores employee number, employee name, department, and salary information. The primary key for the *employees* table should be the employee number. Create a foreign key on the *employees* table that references the *departments* table based on the department_id field.

```
CREATE TABLE departments
( department_id number(10) not null,
  department_name varchar2(50) not null,
  CONSTRAINT departments_pk PRIMARY KEY (department_id)
);
```

Solution for Practice Exercise #3 - SQL CREATE TABLE Statement:

The SQL CREATE TABLE statement for the *employees* table is:

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  department_id number(10),
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number),
  CONSTRAINT fk_departments
    FOREIGN KEY (department_id)
    REFERENCES departments(department_id)
);
```

SQL: Data Types

The following is a list of general SQL datatypes that may not be supported by all relational databases.

Data Type	**Syntax**	**Explanation (if applicable)**
number	numeric(p,s)	Where ***p*** is a precision value; ***s*** is a scale value. For example, numeric(6,2) is a number that has 4 digits before the decimal and 2 digits after the decimal.
character	char(x)	Where ***x*** is the number of characters to store. This data type is space padded to fill the number of characters specified.
character varying	varchar2(x)	Where ***x*** is the number of characters to store. This data type does NOT space pad.
bit	bit(x)	Where ***x*** is the number of bits to store.
date	date	Stores year, month, and day values.
time	time	Stores the hour, minute, and second values.

SQL: DROP TABLE Statement

The **SQL DROP TABLE statement** allows you to remove or delete a table from the SQL database.

SQL DROP TABLE Syntax

The syntax for the **SQL DROP TABLE statement** is:

```
DROP TABLE table_name;
```

table_name is the name of the table to remove from the database.

SQL DROP TABLE Statement Examples

Let's look at a simple SQL DROP TABLE statement example.

For example:

```
DROP TABLE supplier;
```

This would drop table called *supplier*.

SQL: ALTER TABLE Statement

The **SQL ALTER TABLE statement** is used to add, modify, or drop/delete columns in a table. The **SQL ALTER TABLE statement** is also used to rename a table.

SQL ALTER TABLE - Add column in table

Add Column Syntax

To add a column in a table, the SQL ALTER TABLE syntax is:

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

Let's look at SQL ALTER TABLE example that adds a column.

For example:

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

This SQL ALTER TABLE example will add a column called *supplier_name* to the *supplier* table.

Add Multiple Columns Syntax

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

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

Let's look at SQL ALTER TABLE example that adds more than one column.

For example:

```
ALTER TABLE supplier
  ADD (supplier_name varchar2(50),
       city varchar2(45));
```

This SQL ALTER TABLE example will add two columns, *supplier_name* as a varchar2(50) field and *city* as a varchar2(45) field to the *supplier* table.

SQL ALTER TABLE - Modify column in table

Modify Column Syntax

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

```
ALTER TABLE table_name
```

```
  MODIFY column_name column_type;
```

Let's look at SQL ALTER TABLE example that modifies a column.

For example:

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

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

Modify Multiple Columns Syntax

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

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

Let's look at SQL ALTER TABLE example that modifies more than one column.

For example:

```
ALTER TABLE supplier
  MODIFY (supplier_name varchar2(100) not null,
          city varchar2(75));
```

This SQL ALTER TABLE example will modify both the *supplier_name* and *city* columns.

SQL ALTER TABLE - Drop column in table

Drop Column Syntax

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

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

Let's look at SQL ALTER TABLE example that drops (ie: deletes) a column from a table.

For example:

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

This SQL ALTER TABLE example will drop the column called *supplier_name* from the table called *supplier*.

SQL ALTER TABLE - Rename column in table (NEW in Oracle 9i Release 2)

Rename Column Syntax

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

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

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

Let's look at SQL ALTER TABLE example that renames a column in a table.

For example:

```
ALTER TABLE supplier
  RENAME COLUMN supplier_name to sname;
```

This SQL ALTER TABLE example will rename the column called *supplier_name* to *sname*.

SQL ALTER TABLE - Rename table

Rename Table Syntax

To rename a table, the **SQL ALTER TABLE syntax** is:

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

Let's look at SQL ALTER TABLE example that renames a table.

For example:

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

This SQL ALTER TABLE example will rename the *suppliers* table to *vendors*.

Practice Exercise #1 - SQL ALTER TABLE Statement:

Based on the *departments* table below, rename the *departments* table to *depts*.

```
CREATE TABLE departments
( department_id number(10) not null,
  department_name varchar2(50) not null,
  CONSTRAINT departments_pk PRIMARY KEY (department_id)
);
```

Solution for Practice Exercise #1 - SQL ALTER TABLE Statement:

The following SQL ALTER TABLE statement would rename the *departments* table to *depts*:

```
ALTER TABLE departments
  RENAME TO depts;
```

Practice Exercise #2 - SQL ALTER TABLE Statement:

Based on the *employees* table below, add a column called *salary* that is a number(6) datatype.

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  department_id number(10),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Solution for Practice Exercise #2 - SQL ALTER TABLE Statement:

The following SQL ALTER TABLE statement would add a *salary* column to the *employees* table:

```
ALTER TABLE employees
  ADD salary number(6);
```

Practice Exercise #3 - SQL ALTER TABLE Statement:

Based on the *customers* table below, add two columns - one column called *contact_name* that is a varchar2(50) datatype and one column called *last_contacted* that is a date datatype.

```
CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50) not null,
  address varchar2(50),
  city varchar2(50),
  state varchar2(25),
  zip_code varchar2(10),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Solution for Practice Exercise #3 - SQL ALTER TABLE Statement:

The following SQL ALTER TABLE statement would add the *contact_name* and *last_contacted* columns to the *customers* table:

```
ALTER TABLE customers
  ADD (contact_name varchar2(50),
       last_contacted date);
```

Practice Exercise #4 - SQL ALTER TABLE Statement:

Based on the *employees* table below, change the *employee_name* column to a varchar2(75) datatype.

```
CREATE TABLE employees
```

```
( employee_number number(10) not null,
  employee_name >varchar2(50) not null,
  department_id number(10),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Solution for Practice Exercise #4 - SQL ALTER TABLE Statement:

The following SQL ALTER TABLE statement would change the datatype for the *employee_name* column to varchar2(75):

```
ALTER TABLE employees
  MODIFY employee_name varchar2(75);
```

Practice Exercise #5 - SQL ALTER TABLE Statement:

Based on the *customers* table below, change the *customer_name* column to NOT allow null values and change the *state* column to a varchar2(2) datatype.

```
CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50),
  address varchar2(50),
  city varchar2(50),
  state varchar2(25),
  zip_code varchar2(10),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Solution for Practice Exercise #5 - SQL ALTER TABLE Statement:

The following SQL ALTER TABLE statement would modify the *customer_name* and *state* columns accordingly in the *customers* table:

```
ALTER TABLE customers
  MODIFY (customer_name varchar2(50) not null,
          state varchar2(2));
```

Practice Exercise #6 - SQL ALTER TABLE Statement:

Based on the *employees* table below, drop the *salary* column.

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  department_id number(10),
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Solution for Practice Exercise #6 - SQL ALTER TABLE Statement:

The following SQL ALTER TABLE statement would drop the *salary* column from the *employees* table:

```
ALTER TABLE employees  DROP COLUMN salary;
```

Practice Exercise #7 - SQL ALTER TABLE Statement:

Based on the *departments* table below, rename the *department_name* column to *dept_name*.

```
CREATE TABLE departments
( department_id number(10) not null,
  department_name varchar2(50) not null,
  CONSTRAINT departments_pk PRIMARY KEY (department_id)
);
```

Solution for Practice Exercise #7 - SQL ALTER TABLE Statement:

The following SQL ALTER TABLE statement would rename the *department_name* column to *dept_name* in the *departments* table:

```
ALTER TABLE departments
  RENAME COLUMN department_name to dept_name;
```

SQL: INSERT Statement

The **SQL INSERT statement** is a SQL query to insert a single record or multiple records into a table.

SQL INSERT Syntax

The syntax for the **SQL INSERT statement** when inserting a single record using the **VALUES keyword** is:

```
INSERT INTO table
(column1, column2, ... )
VALUES
(expression1, expression2, ... );
```

Or the syntax for the **SQL INSERT statement** when inserting multiple records **using a sub-select** is:

```
INSERT INTO table
(column1, column2, ... )
SELECT expression1, expression2, ...
FROM source_table
WHERE conditions;
```

table is the table to insert the records into.

column1, *column2* are the columns in the *table* to insert values.

expression1, *expression2* are the values to assign to the columns in the table. So *column1* would be assigned the value of *expression1*, *column2* would be assigned the value of *expression2*, and so on.

source_table is the source table when inserting data from another table.

conditions are conditions that must be met for the records to be inserted.

Note

- When inserting records into a table using the SQL INSERT statement, you must provide a value for every NOT NULL column.
- You can omit a column from the SQL INSERT statement if the column allows NULL values.

SQL INSERT Example - Using VALUES keyword

The simplest way to create a SQL INSERT query to list the values using the VALUES keyword.

For example:

```
INSERT INTO suppliers
(supplier_id, supplier_name)
VALUES
(24553, 'IBM');
```

This SQL INSERT statement would result in one record being inserted into the suppliers table. This new record would have a supplier_id of 24553 and a supplier_name of IBM.

SQL INSERT Example - Using sub-select

You can also create more complicated SQL INSERT statements using sub-selects.

For example:

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

By placing a "select" in the SQL INSERT statement, you can perform multiples inserts quickly.

With this type of insert, you may wish to check for the number of rows being inserted. You can determine the number of rows that will be inserted by running the following SQL SELECT statement **before** performing the insert.

```
SELECT count(*)
FROM customers
WHERE city = 'Newark';
```

Frequently Asked Questions (SQL INSERT)

Question: I am setting up a database with clients. I know that you use the SQL INSERT statement to insert information in the database, but how do I make sure that I do not enter the same client information again?

Answer: You can make sure that you do not insert duplicate information by using the SQL EXISTS condition.

For example, if you had a table named *clients* with a primary key of *client_id*, you could use the following SQL INSERT statement:

```
INSERT INTO clients
(client_id, client_name, client_type)
SELECT supplier_id, supplier_name, 'advertising'
FROM suppliers
WHERE NOT EXISTS (SELECT *
                  FROM clients
                  WHERE clients.client_id = suppliers.supplier_id);
```

This SQL INSERT statement inserts multiple records with a subselect.

If you wanted to insert a single record, you could use the following SQL INSERT statement:

```
INSERT INTO clients
(client_id, client_name, client_type)
SELECT 10345, 'IBM', 'advertising'
FROM dual WHERE NOT EXISTS (SELECT * FROM clients
                  WHERE clients.client_id = 10345);
```

The use of the **dual** table allows you to enter your values in a select statement, even though the values are not currently stored in a table.

Question: How can I insert multiple rows of explicit data in one SQL command in Oracle?

Answer: The following is an example of how you might insert 3 rows into the *suppliers* table in Oracle, using a SQL INSERT statement:

```
INSERT ALL
  INTO suppliers (supplier_id, supplier_name) VALUES (1000, 'IBM')
  INTO suppliers (supplier_id, supplier_name) VALUES (2000, 'Microsoft')
  INTO suppliers (supplier_id, supplier_name) VALUES (3000, 'Google')
SELECT * FROM dual;
```

Practice Exercise #1 - SQL INSERT Statement:

Based on the *employees* table, insert an employee record whose *employee_number* is 1001, *employee_name* is Sally Johnson and *salary* is $32,000:

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Solution for Practice Exercise #1 - SQL INSERT Statement:

The following SQL INSERT statement would insert this record into the *employees* table:

```
INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1001, 'Sally Johnson', 32000);
```

Practice Exercise #2 - SQL INSERT Statement:

Based on the *suppliers* table, insert a supplier record whose *supplier_id* is 5001 and *supplier_name* is Apple:

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);
```

Solution for Practice Exercise #2 - SQL INSERT Statement:

The following SQL INSERT statement would insert this record into the *suppliers* table:

```
INSERT INTO suppliers (supplier_id, supplier_name)
VALUES (5001, 'Apple');
```

Practice Exercise #3 - SQL INSERT Statement:

Based on the *customers* and *old_customers* table, insert into the *customers* table all records from the *old_customers* table whose *status* is DELETED.

```
CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);

CREATE TABLE old_customers
( old_customer_id number(10) not null,
  old_customer_name varchar2(50) not null,
  old_city varchar2(50),
  status varchar2(20),
  CONSTRAINT old_customers_pk PRIMARY KEY (old_customer_id)
);
```

Solution for Practice Exercise #3 - SQL INSERT Statement:

The following SQL INSERT statement would be the solution to insert into the customers table using a sub-select:

```
INSERT INTO customers
(customer_id, customer_name, city)
SELECT old_customer_id, old_customer_name, old_city
FROM old_customers
WHERE status = 'DELETED';
```

SQL: DELETE Statement

The **SQL DELETE statement** is a SQL query to delete a single record or multiple records from a table.

SQL DELETE Syntax

The syntax for the **SQL DELETE statement** is:

```
DELETE FROM table
WHERE conditions;
```

table is the table that you wish to delete records from.

conditions are conditions that must be met for the records to be deleted.

Note

- You do not need to list fields in the SQL DELETE statement since you are deleting the entire row from the table.

SQL DELETE Example - One condition

Let's look at a simple SQL DELETE query example, where we just have one condition in our SQL DELETE statement.

For example:

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

This SQL DELETE example 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 SELECT statement **before** performing the delete.

```
SELECT count(*)
FROM suppliers
WHERE supplier_name = 'IBM';
```

SQL DELETE Example - Two conditions

Let's look at a SQL DELETE example, where we just have two conditions in the SQL DELETE statement.

For example:

```
DELETE FROM products
WHERE units >= 12
AND category = 'Clothing';
```

This SQL DELETE example would delete all records from the products table where the *units* is greater than or equal to 12 and the *category* is Clothing.

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 SELECT statement **before** performing the delete.

```
SELECT count(*)
FROM products
WHERE units >= 12
AND category = 'Clothing';
```

SQL DELETE Example - Using SQL EXISTS Clause

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 SQL FROM clause when you are performing a delete, you can use the SQL EXISTS clause.

For example:

```
DELETE FROM suppliers
WHERE EXISTS
  ( SELECT customers.customer_name
    FROM customers
    WHERE customers.customer_id = suppliers.supplier_id
    AND customers.customer_name = 'IBM' );
```

This SQL DELETE example 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.

If you wish to determine the number of rows that will be deleted, you can run the following SQL SELECT statement **before** performing the delete.

```
SELECT COUNT(*) FROM suppliers
WHERE EXISTS
  ( SELECT customers.customer_name
    FROM customers
    WHERE customers.customer_id = suppliers.supplier_id
    AND customers.customer_name = 'IBM' );
```

Frequently Asked Questions (SQL DELETE)

Question: How would I write a SQL DELETE statement to delete all records in TableA whose data in field1 & field2 DO NOT match the data in fieldx & fieldz of TableB?

Answer: You could try something like this for your SQL DELETE statement:

```
DELETE FROM TableA
WHERE NOT EXISTS
  ( SELECT *
    FROM TableB
     WHERE TableA.field1 = TableB.fieldx AND TableA.field2 = TableB.fieldz );
```

Practice Exercise #1 - SQL DELETE Statement:

Based on the *employees* table, delete all employee records whose salary is greater than $40,000:

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Solution for Practice Exercise #1 - SQL DELETE Statement:

The following SQL DELETE statement would delete these records from the *employees* table:

```
DELETE FROM employees
WHERE salary > 40000;
```

Practice Exercise #2 - SQL DELETE Statement:

Based on the *suppliers* table, delete the supplier record whose *supplier_id* is 5001 and *supplier_name* is Apple:

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);
```

Solution for Practice Exercise #2 - SQL DELETE Statement:

The following SQL DELETE statement would delete this record from the *suppliers* table:

```
DELETE FROM suppliers
WHERE supplier_id = 5001
AND supplier_name = 'Apple';
```

Practice Exercise #3 - SQL DELETE Statement:

Based on the *customers* and *old_customers* table, delete from the *customers* table all records that exist in the *old_customers* table (matching the *customer_id* field from the *customers* table to the *old_customer_id* field in the *old_customers* table).

```
CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

```
CREATE TABLE old_customers
( old_customer_id number(10) not null,
  old_customer_name varchar2(50) not null,
  old_city varchar2(50),
  status varchar2(20),
  CONSTRAINT old_customers_pk PRIMARY KEY (old_customer_id)
);
```

Solution for Practice Exercise #3 - SQL DELETE Statement:

The following SQL DELETE statement would be the solution (using the SQL EXISTS clause) that would delete the records from the *customers* table:

```
DELETE FROM customers
WHERE EXISTS
  ( SELECT old_customers.old_customer_id
    FROM old_customers
    WHERE old_customers.old_customer_id = customers.customer_id );
```

SQL: UPDATE Statement

The **SQL UPDATE statement** is a SQL query to update a single record or multiple records in a table.

SQL UPDATE Syntax

The syntax for the **SQL UPDATE statement** is:

```
UPDATE table
SET column1 = expression1,
    column2 = expression2,
    ...
WHERE conditions;
```

column1, *column2* are the columns that you wish to update.

expression1, *expression2* are the new values to assign to the *column1*, *column2*. So *column1* would be assigned the value of *expression1*, *column2* would be assigned the value of *expression2*, and so on.

conditions are the conditions that must be met for the update to execute.

SQL UPDATE Example - Updating single column

Let's look at a very simple SQL UPDATE query example.

```
UPDATE suppliers
SET supplier_id = 50001
WHERE supplier_name = 'Apple';
```

This SQL UPDATE example would update the *supplier_id* to 50001 in the *suppliers* table where the *supplier_name* is 'Apple'.

SQL UPDATE Example - Updating multiple columns

Let's look at a SQL UPDATE example where you might want to update more than one column with a single SQL UPDATE statement.

```
UPDATE suppliers
SET supplier_name = 'Apple',
    product = 'iPhone'
WHERE supplier_name = 'RIM';
```

When you wish to update multiple columns, you can do this by separating the column/value pairs with commas.

This SQL UPDATE statement example would update the *supplier_name* to "Apple" and *product* to "iPhone" where the *supplier_name* is "RIM".

SQL UPDATE Example - Using SQL EXISTS Clause

You can also perform more complicated updates in SQL.

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 SQL UPDATE statement, you can use the SQL EXISTS clause.

For example:

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

In this SQL UPDATE example, whenever a *supplier_id* matched a *customer_id* value, the *supplier_name* would be overwritten to the *customer_name* from the *customers* table.

Practice Exercise #1 - SQL UPDATE Statement:

Based on the *suppliers* table populated with the following data, update the *city* to "Santa Clara" for all records whose *supplier_name* is "NVIDIA".

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5001, 'Microsoft', 'New York');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5002, 'IBM', 'Chicago');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5003, 'Red Hat', 'Detroit');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5004, 'NVIDIA', 'New York');
```

Solution for Practice Exercise #1 - SQL UPDATE Statement:

The following SQL UPDATE statement would perform this update in SQL.

```
UPDATE suppliers
SET city = 'Santa Clara'
```

```
WHERE supplier_name = 'NVIDIA';
```

The *suppliers* table would now look like this:

SUPPLIER_ID	SUPPLIER_NAME	CITY
5001	Microsoft	New York
5002	IBM	Chicago
5003	Red Hat	Detroit
5004	NVIDIA	Santa Clara

Practice Exercise #2 - SQL UPDATE Statement:

Based on the *suppliers* and *customers* table populated with the following data, update the *city* in the *suppliers* table with the *city* in the *customers* table when the *supplier_name* in the *suppliers* table matches the *customer_name* in the *customers* table.

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5001, 'Microsoft', 'New York');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5002, 'IBM', 'Chicago');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5003, 'Red Hat', 'Detroit');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES (5005, 'NVIDIA', 'LA');


CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);

INSERT INTO customers (customer_id, customer_name, city)
VALUES (7001, 'Microsoft', 'San Francisco');

INSERT INTO customers (customer_id, customer_name, city)
VALUES (7002, 'IBM', 'Toronto');
```

```
INSERT INTO customers (customer_id, customer_name, city)
VALUES (7003, 'Red Hat', 'Newark');
```

Solution for Practice Exercise #2 - SQL UPDATE Statement:

The following SQL UPDATE statement would perform this update in SQL.

```
UPDATE suppliers
SET city = (SELECT customers.city
            FROM customers
            WHERE customers.customer_name = suppliers.supplier_name)
WHERE EXISTS (SELECT customers.city
              FROM customers
              WHERE customers.customer_name = suppliers.supplier_name);
```

The *suppliers* table would now look like this:

SUPPLIER_ID	SUPPLIER_NAME	CITY
5001	Microsoft	San Francisco
5002	IBM	Toronto
5003	Red Hat	Newark
5004	NVIDIA	LA

Oracle/PLSQL: Primary Keys

What is a primary key?

A **primary key** is a single field or combination of fields that uniquely defines a record. None of the fields that are part of the primary key can contain a null value. A table can have only one primary key.

Note

In Oracle, a primary key cannot contain more than 32 columns.

A primary key can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

Using a CREATE TABLE statement

The syntax for creating a primary key using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,
  ...

  CONSTRAINT constraint_name PRIMARY KEY (column1, column2, ... column_n)
);
```

For Example

```
CREATE TABLE supplier
(
  supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);
```

In this example, we've created a primary key on the supplier table called supplier_pk. It consists of only one field - the supplier_id field.

We could also create a primary key with more than one field as in the example below:

```
CREATE TABLE supplier
(
  supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name)
);
```

Using an ALTER TABLE statement

The syntax for creating a primary key in an ALTER TABLE statement is:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name PRIMARY KEY (column1, column2, ... column_n);
```

For Example

```
ALTER TABLE supplier
ADD CONSTRAINT supplier_pk PRIMARY KEY (supplier_id);
```

In this example, we've created a primary key on the existing supplier table called supplier_pk. It consists of the field called supplier_id.

We could also create a primary key with more than one field as in the example below:

```
ALTER TABLE supplier
ADD CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name);
```

Drop a Primary Key

The syntax for dropping a primary key is:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE supplier
DROP CONSTRAINT supplier_pk;
```

In this example, we're dropping a primary key on the supplier table called supplier_pk.

Disable a Primary Key

The syntax for disabling a primary key is:

```
ALTER TABLE table_name
DISABLE CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE supplier
DISABLE CONSTRAINT supplier_pk;
```

In this example, we're disabling a primary key on the supplier table called supplier_pk.

Enable a Primary Key

The syntax for enabling a primary key is:

```
ALTER TABLE table_name
ENABLE CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE supplier
ENABLE CONSTRAINT supplier_pk;
```

In this example, we're enabling a primary key on the supplier table called supplier_pk.

Oracle/PLSQL: Foreign Keys

What is a foreign key?

A **foreign key** means that values in one table must also appear in another table.

The referenced table is called the **parent table** while the table with the foreign key is called the **child table**. The foreign key in the child table will generally reference a primary key in the parent table.

A foreign key can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

Using a CREATE TABLE statement

The syntax for creating a foreign key using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,
  ...

  CONSTRAINT fk_column
    FOREIGN KEY (column1, column2, ... column_n)
    REFERENCES parent_table (column1, column2, ... column_n)
);
```

For Example

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  CONSTRAINT fk_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
);
```

In this example, we've created a primary key on the supplier table called *supplier_pk*. It consists of only one field - the supplier_id field. Then we've created a foreign key called *fk_supplier* on the products table that references the supplier table based on the supplier_id field.

We could also create a foreign key with more than one field as in the example below:

```
CREATE TABLE supplier<
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name));
```

```
CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  CONSTRAINT fk_supplier_comp
    FOREIGN KEY (supplier_id, supplier_name)
    REFERENCES supplier(supplier_id, supplier_name)
);
```

In this example, our foreign key called *fk_foreign_comp* references the supplier table based on two fields - the supplier_id and supplier_name fields.

Using an ALTER TABLE statement

The syntax for creating a foreign key in an ALTER TABLE statement is:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
   FOREIGN KEY (column1, column2, ... column_n)
   REFERENCES parent_table (column1, column2, ... column_n);
```

For Example

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id)
  REFERENCES supplier(supplier_id);
```

In this example, we've created a foreign key called *fk_supplier* that references the supplier table based on the supplier_id field.

We could also create a foreign key with more than one field as in the example below:

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id, supplier_name)
  REFERENCES supplier(supplier_id, supplier_name);
```

Oracle/PLSQL: Foreign Keys with cascade delete

What is a foreign key?

A **foreign key** means that values in one table must also appear in another table.

The referenced table is called the **parent table** while the table with the foreign key is called the **child table**. The foreign key in the child table will generally reference a primary key in the parent table.

A foreign key with a cascade delete means that if a record in the parent table is deleted, then the corresponding records in the child table with automatically be deleted. This is called a cascade delete.

A foreign key with a cascade delete can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

Using a CREATE TABLE statement

The syntax for creating a foreign key using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,
  ...

  CONSTRAINT fk_column
     FOREIGN KEY (column1, column2, ... column_n)
     REFERENCES parent_table (column1, column2, ... column_n)
     ON DELETE CASCADE
);
```

For Example

```
CREATE TABLE supplier
( supplier_id numeric(10) >not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  CONSTRAINT fk_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
    ON DELETE CASCADE
);
```

In this example, we've created a primary key on the supplier table called *supplier_pk*. It consists of only one field - the supplier_id field. Then we've created a foreign key called *fk_supplier* on the products table that references the supplier table based on the supplier_id field.

Because of the cascade delete, when a record in the supplier table is deleted, all records in the products table will also be deleted that have the same supplier_id value.

We could also create a foreign key (with a cascade delete) with more than one field as in the example below:

```
CREATE TABLE supplier
( supplier_id numeric(10)< not null,
  supplier_name varchar2(50) >not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  CONSTRAINT fk_supplier_comp
    FOREIGN KEY (supplier_id, supplier_name)
    REFERENCES supplier(supplier_id, supplier_name)
    ON DELETE CASCADE
);
```

In this example, our foreign key called *fk_foreign_comp* references the supplier table based on two fields - the supplier_id and supplier_name fields.

The cascade delete on the foreign key called *fk_foreign_comp* causes all corresponding records in the products table to be cascade deleted when a record in the supplier table is deleted, based on supplier_id and supplier_name.

Using an ALTER TABLE statement

The syntax for creating a foreign key in an ALTER TABLE statement is:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
   FOREIGN KEY (column1, column2, ... column_n)
   REFERENCES parent_table (column1, column2, ... column_n)
   ON DELETE CASCADE;
```

For Example

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id)
  REFERENCES supplier(supplier_id)
  ON DELETE CASCADE;
```

In this example, we've created a foreign key (with a cascade delete) called *fk_supplier* that references the supplier table based on the supplier_id field.

We could also create a foreign key (with a cascade delete) with more than one field as in the example below:

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id, supplier_name)
  REFERENCES supplier(supplier_id, supplier_name) ON DELETE CASCADE;
```

Oracle/PLSQL: Foreign Keys with "set null on delete"

What is a foreign key?

A **foreign key** means that values in one table must also appear in another table.

The referenced table is called the **parent table** while the table with the foreign key is called the **child table**. The foreign key in the child table will generally reference a primary key in the parent table.

A foreign key with a "set null on delete" means that if a record in the parent table is deleted, then the corresponding records in the child table will have the foreign key fields set to null. The records in the child table will **not** be deleted.

A foreign key with a "set null on delete" can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

Using a CREATE TABLE statement

The syntax for creating a foreign key using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,
  ...

  CONSTRAINT fk_column
     FOREIGN KEY (column1, column2, ... column_n)
     REFERENCES parent_table (column1, column2, ... column_n)
     ON DELETE SET NULL
);
```

For Example

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10),
  CONSTRAINT fk_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
    ON DELETE SET NULL
);
```

In this example, we've created a primary key on the supplier table called *supplier_pk*. It consists of only one field - the supplier_id field. Then we've created a foreign key called *fk_supplier* on the products table that references the supplier table based on the supplier_id field.

Because of the set null on delete, when a record in the supplier table is deleted, all corresponding records in the products table will have the supplier_id values set to null.

We could also create a foreign key "set null on delete" with more than one field as in the example below:

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id, supplier_name)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10),
  supplier_name varchar2(50),
  CONSTRAINT fk_supplier_comp
    FOREIGN KEY (supplier_id, supplier_name)
    REFERENCES supplier(supplier_id, supplier_name)
    ON DELETE SET NULL
);
```

In this example, our foreign key called *fk_foreign_comp* references the supplier table based on two fields - the supplier_id and supplier_name fields.

The delete on the foreign key called *fk_foreign_comp* causes all corresponding records in the products table to have the supplier_id and supplier_name fields set to null when a record in the supplier table is deleted, based on supplier_id and supplier_name.

Using an ALTER TABLE statement

The syntax for creating a foreign key in an ALTER TABLE statement is:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name
   FOREIGN KEY (column1, column2, ... column_n)
   REFERENCES parent_table (column1, column2, ... column_n)
   ON DELETE SET NULL;
```

For Example

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id)
  REFERENCES supplier(supplier_id)
  ON DELETE SET NULL;
```

In this example, we've created a foreign key "with a set null on delete" called *fk_supplier* that references the supplier table based on the supplier_id field.

We could also create a foreign key "with a set null on delete" with more than one field as in the example below:

```
ALTER TABLE products
ADD CONSTRAINT fk_supplier
  FOREIGN KEY (supplier_id, supplier_name)
  REFERENCES supplier(supplier_id, supplier_name)
  ON DELETE SET NULL;
```

Oracle/PLSQL: Drop a foreign key

The syntax for dropping a foreign key is:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

For Example

If you had created a foreign key as follows:

```
CREATE TABLE supplier
( supplier_id< numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  CONSTRAINT fk_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
);
```

In this example, we've created a primary key on the supplier table called *supplier_pk*. It consists of only one field - the supplier_id field. Then we've created a foreign key called *fk_supplier* on the products table that references the supplier table based on the supplier_id field.

If we then wanted to drop the foreign key called fk_supplier, we could execute the following command:

```
ALTER TABLE products
DROP CONSTRAINT fk_supplier;
```

Oracle/PLSQL: Disable a foreign key

The syntax for disabling a foreign key is:

```
ALTER TABLE table_name
DISABLE CONSTRAINT constraint_name;
```

For Example

If you had created a foreign key as follows:

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  CONSTRAINT fk_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
);
```

In this example, we've created a primary key on the supplier table called *supplier_pk*. It consists of only one field - the supplier_id field. Then we've created a foreign key called *fk_supplier* on the products table that references the supplier table based on the supplier_id field.

If we then wanted to disable the foreign key called fk_supplier, we could execute the following command:

```
ALTER TABLE products
DISABLE CONSTRAINT fk_supplier;
```

Oracle/PLSQL: Enable a foreign key

The syntax for enabling a foreign key is:

```
ALTER TABLE table_name
ENABLE CONSTRAINT constraint_name;
```

For Example

If you had created a foreign key as follows:

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_pk PRIMARY KEY (supplier_id)
);

CREATE TABLE products
( product_id numeric(10) not null,
  supplier_id numeric(10) not null,
  CONSTRAINT fk_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
);
```

In this example, we've created a primary key on the supplier table called *supplier_pk*. It consists of only one field - the supplier_id field. Then we've created a foreign key called *fk_supplier* on the products table that references the supplier table based on the supplier_id field.

If the foreign key had been disabled and we wanted to enable it, we could execute the following command:

```
ALTER TABLE products
ENABLE CONSTRAINT fk_supplier;
```

Oracle/PLSQL: Unique Constraints

What is a unique constraint?

A **unique constraint** is a single field or combination of fields that uniquely defines a record. Some of the fields can contain null values as long as the combination of values is unique.

Note

In Oracle, a unique constraint can not contain more than 32 columns.

A unique constraint can be defined in either a CREATE TABLE statement or an ALTER TABLE statement.

What is the difference between a unique constraint and a primary key?

Primary Key	Unique Constraint
None of the fields that are part of the primary key can contain a null value.	Some of the fields that are part of the unique constraint can contain null values as long as the combination of values is unique.

Oracle does not permit you to create both a primary key and unique constraint with the same columns.

Using a CREATE TABLE statement

The syntax for creating a unique constraint using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,
  ...

  CONSTRAINT constraint_name UNIQUE (column1, column2, . column_n)
);
```

For Example

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_unique UNIQUE (supplier_id)
);
```

In this example, we've created a unique constraint on the supplier table called supplier_unique. It consists of only one field - the supplier_id field.

We could also create a unique constraint with more than one field as in the example below:

```
CREATE TABLE supplier
( supplier_id numeric(10) not null,
  supplier_name varchar2(50) not null,
  contact_name varchar2(50),
  CONSTRAINT supplier_unique UNIQUE (supplier_id, supplier_name)
);
```

Using an ALTER TABLE statement

The syntax for creating a unique constraint in an ALTER TABLE statement is:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE (column1, column2, ... column_n);
```

For Example

```
ALTER TABLE supplier
ADD CONSTRAINT supplier_unique UNIQUE (supplier_id);
```

In this example, we've created a unique constraint on the existing supplier table called supplier_unique. It consists of the field called supplier_id.

We could also create a unique constraint with more than one field as in the example below:

```
ALTER TABLE supplier
ADD CONSTRAINT supplier_unique UNIQUE (supplier_id, supplier_name);
```

Drop a Unique Constraint

The syntax for dropping a unique constraint is:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE supplier
DROP CONSTRAINT supplier_unique;
```

In this example, we're dropping a unique constraint on the supplier table called supplier_unique.

Disable a Unique Constraint

The syntax for disabling a unique constraint is:

```
ALTER TABLE table_name
DISABLE CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE supplier
DISABLE CONSTRAINT supplier_unique;
```

In this example, we're disabling a unique constraint on the supplier table called supplier_unique.

Enable a Unique Constraint

The syntax for enabling a unique constraint is:

```
ALTER TABLE table_name
ENABLE CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE supplier
ENABLE CONSTRAINT supplier_unique;
```

In this example, we're enabling a unique constraint on the supplier table called supplier_unique.

Oracle/PLSQL: Check Constraints

What is a check constraint?

A **check constraint** allows you to specify a condition on each row in a table.

Note

- A check constraint can NOT be defined on a SQL View.
- The check constraint defined on a table must refer to only columns in that table. It can not refer to columns in other tables.
- A check constraint can NOT include a SQL Subquery.

A check constraint can be defined in either a SQL CREATE TABLE statement or a SQL ALTER TABLE statement.

Using a CREATE TABLE statement

The syntax for creating a check constraint using a CREATE TABLE statement is:

```
CREATE TABLE table_name
(
  column1 datatype null/not null,
  column2 datatype null/not null,

  ...

  CONSTRAINT constraint_name CHECK (column_name condition) [DISABLE]

);
```

The DISABLE keyword is optional. If you create a check constraint using the DISABLE keyword, the constraint will be created, but the condition will not be enforced.

For Example

```
CREATE TABLE suppliers
(
  supplier_id numeric(4),
  supplier_name varchar2(50),
  CONSTRAINT check_supplier_id
  CHECK (supplier_id BETWEEN 100 and 9999)
);
```

In this first example, we've created a check constraint on the suppliers table called check_supplier_id. This constraint ensures that the supplier_id field contains values between 100 and 9999.

```
CREATE TABLE suppliers
(
  supplier_id numeric(4),
  supplier_name varchar2(50),
  CONSTRAINT check_supplier_name
  CHECK (supplier_name = upper(supplier_name)));
```

In this second example, we've created a check constraint called check_supplier_name. This constraint ensures that the supplier_name column always contains uppercase characters.

Using an ALTER TABLE statement

The syntax for creating a check constraint in an ALTER TABLE statement is:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name CHECK (column_name condition) [DISABLE];
```

The DISABLE keyword is optional. If you create a check constraint using the DISABLE keyword, the constraint will be created, but the condition will not be enforced.

For Example

```
ALTER TABLE suppliers
ADD CONSTRAINT check_supplier_name
  CHECK (supplier_name IN ('IBM', 'Microsoft', 'NVIDIA'));
```

In this example, we've created a check constraint on the existing suppliers table called check_supplier_name. It ensures that the supplier_name field only contains the following values: IBM, Microsoft, or NVIDIA.

Drop a Check Constraint

The syntax for dropping a check constraint is:

```
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE suppliers
DROP CONSTRAINT check_supplier_id;
```

In this example, we're dropping a check constraint on the suppliers table called check_supplier_id.

Enable a Check Constraint

The syntax for enabling a check constraint is:

```
ALTER TABLE table_name
ENABLE CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE suppliers
ENABLE CONSTRAINT check_supplier_id;
```

In this example, we're enabling a check constraint on the suppliers table called check_supplier_id.

Disable a Check Constraint

The syntax for disabling a check constraint is:

```
ALTER TABLE table_name
DISABLE CONSTRAINT constraint_name;
```

For Example

```
ALTER TABLE suppliers
DISABLE CONSTRAINT check_supplier_id;
```

In this example, we're disabling a check constraint on the suppliers table called check_supplier_id.

Oracle/PLSQL: SELECT Statement

The **Oracle SELECT statement** is used to retrieve records from one or more tables in Oracle.

Oracle SELECT Syntax

The syntax for the **Oracle SELECT statement** is:

```
SELECT expressions
FROM tables
WHERE conditions;
```

expressions are the columns or calculations that you wish to retrieve.

tables are the tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

conditions are conditions that must be met for the records to be selected.

Oracle SELECT Example - Select all fields from one table

Let's look at how to use an Oracle SELECT query to select all fields from a table.

```
SELECT *
FROM order_details
WHERE quantity >= 10
ORDER BY quantity DESC;
```

In this Oracle SELECT statement example, we've used * to signify that we wish to select all fields from the *order_details* table where the quantity is greater than or equal to 10. The result set is sorted by *quantity* in descending order.

Oracle SELECT Example - Selecting individual fields from one table

You can also use the Oracle SELECT statement to select individual fields from the table, as opposed to all fields from the table.

For example:

```
SELECT order_id, quantity, unit_price
FROM order_details
WHERE quantity < 500
ORDER BY quantity ASC, unit_price DESC;
```

This Oracle SELECT example would return only the *order_id*, *quantity*, and *unit_price* fields from the *order_details* table where the *quantity* is less than 500. The results are sorted by *quantity* in ascending order and then *unit_price* in descending order.

Oracle SELECT Example - Select fields from multiple tables

You can also use the Oracle SELECT statement to retrieve fields from multiple tables.

```
SELECT order_details.order_id, customers.customer_name
FROM customers
INNER JOIN order_details
ON customers.customer_id = order_details.customer_id
ORDER BY order_id;
```

This Oracle SELECT example joins two tables together to gives us a result set that displays the *order_id* and *customer_name* fields where the *customer_id* value matches in both the *customers* and *order_details* table. The results are sorted by *order_id* in ascending order.

Learn more about Oracle joins.

Practice Exercise #1 - Oracle SELECT Statement:

Based on the *employees* table below, select all fields from the *employees* table whose salary is greater to $75,000 (no sorting is required):

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Solution for Practice Exercise #1 - Oracle SELECT Statement:

The following Oracle SELECT statement would select these records from the *employees* table:

```
SELECT *
FROM employees
WHERE salary > 75000;
```

Practice Exercise #2 - Oracle SELECT Statement:

Based on the *suppliers* table below, select the unique *city* values that reside in the *state* of California and order the results in **ascending order** by *city*:

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  city varchar2(50),
  state varchar2(25),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);
```

Solution for Practice Exercise #2 - Oracle SELECT Statement:

The following Oracle SELECT statement would select these records from the *suppliers* table:

```
SELECT DISTINCT city
FROM suppliers
WHERE state = 'California'
ORDER BY city ASC;
```

Practice Exercise #3 - Oracle SELECT Statement:

Based on the *suppliers* table and the *orders* table below, select the *supplier_id* and *supplier_name* from the suppliers table and select the *order_date* from the *orders* table where there is a matching *supplier_id* value in both the *suppliers* and *orders* tables. Order the results by *supplier_id* in **descending order**.

```
CREATE TABLE suppliers
( supplier_id number(10) not null,
  supplier_name varchar2(50) not null,
  city varchar2(50),
  state varchar2(25),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);

CREATE TABLE orders
( order_id number(10) not null,
  supplier_id number(10) not null,
  order_date date not null,
  quantity number(5),
  CONSTRAINT orders_pk PRIMARY KEY (order_id)
);
```

Solution for Practice Exercise #3 - Oracle SELECT Statement:

The following Oracle SELECT statement would select these records from the *suppliers* and *orders* table (using an Oracle INNER JOIN):

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id
ORDER BY supplier_id DESC;
```

Practice Exercise #4 - Oracle SELECT Statement:

Based on the *customers* and *old_customers* table, select the *customer_id* and *customer_name* from the *customers* table that exist in the *old_customers* table (matching the *customer_id* field from the *customers* table to the *old_customer_id* field in the *old_customers* table). Order the results in **ascending order** by *customer_name* and then **descending order** by *customer_id*.

```
CREATE TABLE customers
( customer_id number(10) not null,
  customer_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);

CREATE TABLE old_customers
( old_customer_id number(10) not null,
  old_customer_name varchar2(50) not null,
  old_city varchar2(50),
  status varchar2(20),
  CONSTRAINT old_customers_pk PRIMARY KEY (old_customer_id)
);
```

Solution for Practice Exercise #4 - Oracle SELECT Statement:

The following Oracle SELECT statement would select the records from the *customers* table (using the Oracle EXISTS clause):

```
SELECT customer_id, customer_name
FROM customers
WHERE EXISTS
  ( SELECT old_customers.old_customer_id
    FROM old_customers
    WHERE old_customers.old_customer_id = customers.customer_id )
ORDER BY customer_name ASC, customer_id DESC;
```

Or alternatively you could exclude the **ASC keyword** for *customer_name* in the ORDER BY clause. Both of these SELECT statements would generate the same results:

```
SELECT customer_id, customer_name
FROM customers
WHERE EXISTS
  ( SELECT old_customers.old_customer_id
    FROM old_customers
    WHERE old_customers.old_customer_id = customers.customer_id )
ORDER BY customer_name, customer_jd DESC;
```

Oracle/PLSQL: WHERE Clause

The **Oracle WHERE clause** is used to filter the results from a SELECT, INSERT, UPDATE, or DELETE statement.

Oracle WHERE Syntax

The syntax for the **Oracle WHERE Clause** is:

```
WHERE conditions;
```

It is difficult to explain the syntax for the Oracle WHERE clause, so let's look at some examples.

Oracle WHERE Example - Single condition

```
SELECT *
FROM customers
WHERE last_name = 'Anderson';
```

In this Oracle WHERE clause example, we've used the WHERE clause to filter our results from the *customers* table. The SELECT statement above would return all rows from the *customers* table where the *last_name* is Anderson. Because the * is used in the SELECT, all fields from the *customers* table would appear in the result set.

Oracle WHERE Example - Using "AND"

```
SELECT *
FROM suppliers
WHERE state = 'California'
AND supplier_id <= 750;
```

This Oracle WHERE clause example uses the WHERE clause to define multiple conditions. In this case, this SELECT statement uses the "AND" Condition to return all *suppliers* that are located in the *state* of California and whose *supplier_id* is less than or equal to 750.

Oracle WHERE Example - Using "OR"

```
SELECT supplier_id
FROM suppliers
WHERE supplier_name = 'Apple'
OR supplier_name = 'Microsoft';
```

This Oracle WHERE clause example uses the WHERE clause to define multiple conditions, but instead of using the "AND" Condition, it uses the "OR" Condition. In this case, this SELECT statement would return all *supplier_id* values where the *supplier_name* is Apple **or** Microsoft.

Oracle WHERE Example - Combining "AND" & "OR

```
SELECT *
FROM suppliers
WHERE (state = 'Florida' AND supplier_name = 'IBM')
OR (supplier_id > 5000);
```

This Oracle WHERE clause example uses the WHERE clause to define multiple conditions, but it combines the "AND" Condition and the "OR" Condition. This example would return all *suppliers* that reside in the *state* of Florida and whose *supplier_name* is IBM as well as all suppliers whose *supplier_id* is greater than 5000.

The brackets determine the order that the AND and OR conditions are evaluated. Just like you learned in the order of operations in Math class!

Oracle WHERE Example - Joining Tables

```
SELECT suppliers.suppler_name, orders.order_id
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id
WHERE suppliers.state = 'California';
```

This Oracle WHERE clause example uses the WHERE clause to join multiple tables together in a single SELECT statement. This SELECT statement would return all *supplier_name* and *order_id* values where there is a matching record in the *suppliers* and *orders* tables based on *supplier_id*, and where the supplier's *state* is California.

Oracle/PLSQL: "AND" Condition

The **Oracle "AND" Condition** (also known as the Oracle "AND" Operator) is used to create 2 or more conditions to be met. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

Oracle "AND" Syntax

The syntax for the **Oracle "AND" Condition** is:

```
SELECT columns
FROM tables
WHERE condition1
AND condition2;
```

The **Oracle "AND" condition** requires that each condition (ie: condition1 and condition2) be must be met for the record to be included in the result set.

Oracle "AND" Example - SELECT Statement

The first Oracle "AND" Condition query involves a SELECT statement with 2 conditions.

For example:

```
SELECT *
FROM customers
WHERE state = 'Florida'
AND customer_id > 5000;
```

This Oracle "AND" example would return all customers that reside in the *state* of Florida and have a *customer_id* > 5000. Because the * is used in the SELECT statement, all fields from the *customers* table would appear in the result set.

Oracle "AND" Example - JOINING Tables

Our next Oracle "AND" example shows how the "AND" condition can be used to join multiple tables in a SELECT statement.

For example:

```
SELECT orders.order_id, suppliers.supplier_name
FROM suppliers
WHERE suppliers.supplier_id = orders.supplier_id
AND suppliers.supplier_name = 'Microsoft';
```

Though the above SQL works just fine, you would more traditionally write this SQL as follows using a proper INNER JOIN.

For example:

```
SELECT orders.order_id, suppliers.supplier_name
```

```
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id
WHERE suppliers.supplier_name = 'Microsoft';
```

This Oracle "AND" condition example would return all rows where the *supplier_name* is Microsoft. And the *supplier*s and *orders* tables are joined on *supplier_id*. You will notice that all of the fields are prefixed with the table names (ie: orders.order_id). This is required to eliminate any ambiguity as to which field is being referenced; as the same field name can exist in both the *suppliers* and the *orders* tables.

In this case, the result set would only display the *order_id* and *supplier_name* fields (as listed in the first part of the SELECT statement.).

Oracle "AND" Example - INSERT Statement

This next Oracle "AND" example demonstrates how the "AND" Condition can be used in the INSERT statement.

For example:

```
INSERT INTO suppliers
(supplier_id, supplier_name)
SELECT customer_id, customer_name
FROM customers
WHERE customer_name = 'Microsoft'
AND customer_id <= 1000;
```

This Oracle "AND" Condition example would insert into the *suppliers* table, all *customer_id* and *customer_name* records from the *customers* table whose customer_name is Microsoft and have a *customer_id* less than or equal to 1000.

Oracle "AND" Example - UPDATE Statement

This Oracle "AND" condition example shows how the "AND" Condition can be used in the UPDATE statement.

For example:

```
UPDATE suppliers
SET supplier_name = 'Apple'
WHERE supplier_name = 'RIM'
AND offices = 8;
```

This Oracle "AND" Condition example would update all *supplier_name* values in the *suppliers* table to Apple where the *supplier_name* was RIM and had 8 offices.

Oracle "AND" Example - DELETE Statement

Finally, this last Oracle "AND" example demonstrates how the "AND" Condition can be used in the DELETE statement.

For example:

```
DELETE FROM suppliers
WHERE supplier_name = 'Apple'
AND product = 'iPod';
```

This Oracle "AND" Condition example would delete all records from the *suppliers* table whose *supplier_name* was Apple and product was iPod.

Oracle/PLSQL: "OR" Condition

The **Oracle "OR" Condition** is used to create a SQL statement where records are returned when any one of the conditions are met. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

Oracle "OR" Syntax

The syntax for the **Oracle "OR" Condition** is:

```
SELECT columns
FROM tables
WHERE condition1
OR condition2;
```

The **Oracle "OR" Condition** requires that any of the conditions (ie: condition1 or condition2) be must be met for the record to be included in the result set.

Oracle "OR" Example - SELECT

The first Oracle "OR" Condition example that we'll take a look at involves an Oracle SELECT statement with 2 conditions:

```
SELECT *
FROM customers
WHERE state = 'California'
OR available_available > 500;
```

This Oracle "OR" Condition example would return all customers that reside in either the state of California or have available_credit greater than 500. Because the * is used in the SELECT statement, all fields from the *customers* table would appear in the result set.

Oracle "OR" Example - SELECT with 3 conditions

The next Oracle "OR" example looks at an Oracle SELECT statement with 3 conditions. If any of these conditions is met, the record will be included in the result set.

```
SELECT supplier_id
FROM suppliers
WHERE supplier_name = 'IBM'
OR city = 'New York'
OR offices > 5;
```

This Oracle "OR" Condition example would return all supplier_id values where the supplier's name is either IBM, city is New York, or offices is greater than 5.

Oracle "OR" Example - INSERT

The Oracle "OR" Condition can be used in the Oracle INSERT statement.

For example:

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

This Oracle "OR" example would insert into the *suppliers* table, all *account_no* and *name* records from the *customers* table that reside in either New York or Newark.

Oracle "OR" Example - UPDATE

The Oracle "OR" Condition can be used in the Oracle UPDATE statement.

For example:

```
UPDATE suppliers
SET supplier_name = 'Apple'
WHERE supplier_name = 'RIM'
OR available_products < 10;
```

This Oracle "OR" Condition example would update all *supplier_name* values in the *suppliers* table to Apple where the *supplier_name* was RIM or its availabe_products was less than 10.

Oracle "OR" Example - DELETE

The Oracle "OR" Condition can be used in the Oracle DELETE statement.

For example:

```
DELETE FROM suppliers
WHERE supplier_name = 'HP'
OR employees >= 60;
```

This Oracle "OR" Condition example would delete all suppliers from the *suppliers* table whose *supplier_name* was HP or its employees was greater than or equal to 60.

Oracle/PLSQL: "AND" & "OR" Conditions

The Oracle "AND" Condition and Oracle "OR" Condition can be combined in a SELECT, INSERT, UPDATE, or DELETE statement.

When combining these conditions, it is important to use brackets so that the database knows what order to evaluate each condition. (Just like when you were learning the order of operations in Math class!)

Oracle "AND" & "OR" Syntax

The syntax for the **Oracle "AND" Condition** is:

```
SELECT columns
FROM tables
WHERE condition1
AND condition2
OR condition3;
```

But don't forget the order of operation brackets!

Oracle "AND" & "OR" Example - SELECT Statement

Let's look at an example that combines the "AND" and "OR" conditions in an Oracle SELECT statement.

For example:

```
SELECT *
FROM suppliers
WHERE (state = 'California' AND supplier_name = 'IBM')
OR (supplier_id < 5000);
```

This "AND" & "OR" example would return all suppliers that reside in the *state* of California whose *supplier_name* is IBM and all suppliers whose supplier_id is less than 5000. The brackets determine the order that the AND and OR conditions are evaluated. Just like you learned in the order of operations in Math class!

The next example takes a look at a more complex statement.

For example:

```
SELECT supplier_id
FROM suppliers
WHERE (supplier_name = 'IBM')
OR (supplier_name = 'Apple' AND state = 'Florida')
OR (supplier_name = 'Best Buy' AND status = 'Active' AND state = 'California');
```

This "AND" & "OR" example would return all *supplier_id* values where the *supplier_name* is IBM OR the *supplier_name* is Apple and the *state* is Florida OR the *supplier_name* is Best Buy, the *status* is Active and the *state* is California.

Oracle "AND" & "OR" Example - INSERT Statement

This next "AND" & "OR" example demonstrates how the "AND" Condition and "OR" Condition can be combined in the INSERT statement.

For example:

```
INSERT INTO suppliers
(supplier_id, supplier_name)
SELECT account_no, customer_name
FROM customers
WHERE (customer_name = 'Apple' OR customer_name = 'Samsung')
AND customer_id > 20;
```

This Oracle "AND" and "OR" example would insert into the *suppliers* table, all *account_no* and *customer_name* records from the *customers* table whose *customer_name* is either Apple or Samsung and where the customer_id is greater than 20.

Oracle "AND" & "OR" Example - UPDATE Statement

This "AND" & "OR" example shows how the "AND" and "OR" conditions can be used in the UPDATE statement.

For example:

```
UPDATE suppliers
SET supplier_name = 'Samsung'
WHERE supplier_name = 'RIM'
AND state = 'California';
```

This Oracle "AND" & "OR" Condition example would update all *supplier_name* values in the *suppliers* table to Samsung where the supplier_name was RIM and resides in the state of California.

Oracle "AND" & "OR" Example - DELETE Statement

Finally, this last "AND" & "OR" example demonstrates how the "AND" and "OR" conditions can be used in the DELETE statement.

For example:

```
DELETE FROM suppliers
WHERE state = 'Florida'
AND (product = 'PC computers' OR supplier_name = 'Dell');
```

This Oracle "AND" and "OR" Condition example would delete all records from the *suppliers* table whose state was Florida and either the product was PC computers or the supplier name was Dell.

Oracle/PLSQL: "IN" Condition

The **Oracle "IN" Condition** is used to help reduce the need to use multiple "OR" Conditions in a SELECT, INSERT, UPDATE, or DELETE statement.

Oracle "IN" Syntax

The syntax for the **Oracle "IN" Condition** is:

```
expression IN (value1, value2, .... value_n);
```

The **Oracle "IN" Condition** will return the records where *expression* is *value1*, *value2*..., or *value_n*.

Oracle "IN" Example - Character

Let's look at an Oracle "IN" Condition example using character values.

The following is an Oracle SELECT statement that uses the "IN" Condition to compare character values:

```
SELECT *
FROM customers
WHERE customer_name IN ('IBM', 'Hewlett Packard', 'Microsoft');
```

This Oracle "IN" Condition example would return all rows where the *customer_name* is either IBM, Hewlett Packard, or Microsoft. Because the * is used in the SELECT, all fields from the *customers* table would appear in the result set.

The above "IN" example is equivalent to the following SELECT statement:

```
SELECT *
FROM customers
WHERE customer_name = 'IBM'
OR customer_name = 'Hewlett Packard'
OR customer_name = 'Microsoft';
```

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

Oracle "IN" Example - Numeric

Next, let's look at an Oracle "IN" Condition example using numeric values.

For example:

```
SELECT *
FROM orders
WHERE order_id IN (10000, 10001, 10003, 10005);
```

This Oracle "IN" Condition example would return all orders where the *order_id* is either 10000, 10001, 10003, or 10005.

The above "IN" example is equivalent to the following SELECT statement:

```
SELECT *
FROM orders
WHERE order_id = 10000
OR order_id = 10001
OR order_id = 10003
OR order_id = 10005;
```

Oracle "IN" Example - Using NOT operator

Finally, let's look at an "IN" Condition example using the Oracle NOT operator.

For example:

```
SELECT *
FROM customers
WHERE customer_name NOT IN ( 'IBM', 'Hewlett Packard', 'Microsoft');
```

This Oracle "IN" Condition example would return all rows where the *customer_name* is **not** IBM, Hewlett Packard, or Microsoft. Sometimes, it is more efficient to list the values that you do **not** want, as opposed to the values that you do want.

Oracle/PLSQL: BETWEEN Condition

The **Oracle BETWEEN Condition** is used to retrieve values within a range in a SELECT, INSERT, UPDATE, or DELETE statement.

Oracle BETWEEN Syntax

The syntax for the **Oracle BETWEEN Condition** is:

```
expression BETWEEN value1 AND value2;
```

The **Oracle BETWEEN Condition** will return the records where *expression* is within the range of *value1* and *value2* (inclusive).

Oracle BETWEEN Example - Numeric

Let's look at some Oracle BETWEEN Condition examples using numeric values. The following numeric example uses the BETWEEN Condition to retrieve values within a numeric range.

For example:

```
SELECT *
FROM customers
WHERE customer_id BETWEEN 4000 AND 4999;
```

This Oracle BETWEEN example would return all rows from the *customers* table where the *customer_id* is between 4000 and 4999 (inclusive). It is equivalent to the following SELECT statement:

```
SELECT *
FROM customers
WHERE customer_id >= 4000
AND customer_id <= 4999;
```

Oracle BETWEEN Example - Date

Next, let's look at how you would use the Oracle BETWEEN Condition with Dates. The following date example uses the BETWEEN Condition to retrieve values within a date range.

For example:

```
SELECT *
FROM order_details
WHERE order_date BETWEEN TO_DATE ('2014/02/01', 'yyyy/mm/dd')
AND TO_DATE ('2014/02/28', 'yyyy/mm/dd');
```

This Oracle BETWEEN condition example would return all records from the *order_details* table where the *order_date* is between Feb 1, 2014 and Feb 28, 2014 (inclusive). It would be equivalent to the following SELECT statement:

```
SELECT *
FROM order_details
```

```
WHERE order_date >= TO_DATE('2014/02/01', 'yyyy/mm/dd')
AND order_date <= TO_DATE('2014/02/28','yyyy/mm/dd');
```

Oracle BETWEEN Example - Using NOT Operator

The Oracle BETWEEN Condition can also be combined with the Oracle NOT operator. Here is an example of how you would combine the BETWEEN Condition with the NOT Operator.

For example:

```
SELECT *
FROM customers
WHERE customer_id NOT BETWEEN 3000 AND 3500;
```

This Oracle BETWEEN example would return all rows from the *customers* table where the *customer_id* was **NOT** between 3000 and 3500, inclusive. It would be equivalent to the following SELECT statement:

```
SELECT *
FROM customers
WHERE customer_id < 3000
OR customer_id > 3500;
```

Oracle/PLSQL: "NOT" Condition

The **Oracle "NOT" Condition** (also known as the Oracle NOT Operator) is used to negate a condition in a SELECT, INSERT, UPDATE, or DELETE statement.

Oracle "NOT" Syntax

The syntax for the **Oracle "NOT" Condition** is:

```
NOT condition
```

The **Oracle "NOT" condition** requires that the opposite of the *condition* be must be met for the record to be included in the result set.

Oracle "NOT" Example - Combine With IN

The Oracle "NOT" Condition can be combined with the Oracle "IN" Condition.

For example:

```
SELECT *
FROM customers
WHERE customer_name NOT IN ( 'IBM', 'Hewlett Packard', 'Microsoft' );
```

This Oracle "NOT" example would return all rows from the *customers* table where the *customer_name* is **not** IBM, Hewlett Packard, or Microsoft. Sometimes, it is more efficient to list the values that you do **not** want, as opposed to the values that you do want.

Oracle "NOT" Example - Combine With IS NULL

The Oracle "NOT" Condition can also be combined with the Oracle IS NULL Condition.

For example,

```
SELECT *
FROM customers
WHERE last_name IS NOT NULL;
```

This Oracle "NOT" example would return all records from the *customers* table where the *last_name* does not contain a NULL value.

Oracle "NOT" Example - Combine With LIKE

The Oracle "NOT" Condition can also be combined with the Oracle LIKE Condition.

For example:

```
SELECT customer_name
FROM customers
```

```
WHERE customer_name NOT LIKE 'S%';
```

By placing the Oracle "NOT" Operator in front of the LIKE condition, you are able to retrieve all customers whose *customer_name* does **not** start with 'S'.

Oracle "NOT" Example - Combine With BETWEEN

The Oracle "NOT" Condition can also be combined with the Oracle BETWEEN Condition. Here is an example of how you would combine the NOT Operator with the BETWEEN Condition.

For example:

```
SELECT *
FROM customers
WHERE customer_id NOT BETWEEN 4000 AND 4100;
```

This Oracle "NOT" example would return all rows where the *customer_id* was **NOT** between 4000 and 4100, inclusive. It would be equivalent to the following Oracle SELECT statement:

```
SELECT *
FROM customers
WHERE customer_id < 4000
OR customer_id > 4100;
```

Oracle "NOT" Example - Combine With EXISTS

The Oracle "NOT" Condition can also be combined with the Oracle EXISTS Condition.

For example,

```
SELECT *
FROM suppliers
WHERE NOT EXISTS (SELECT *
                  FROM orders
                  WHERE suppliers.supplier_id = orders.supplier_id);
```

This Oracle "NOT" example would return all records from the *suppliers* table where there are **no** records in the *orders* table for the given supplier_id

Oracle/PLSQL: IS NULL

IS NULL is a comparison that evaluates to TRUE if the expression is a null value. You can use the **IS NULL** condition in either a SQL statement or in a block of PLSQL code.

SQL Statement

You can use the **IS NULL** condition in the following types of SQL statements:

- SELECT statement
- INSERT statement
- UPDATE statement
- DELETE statement

SELECT Statement

Here is an example of how you could use the **IS NULL** condition in a SELECT statement:

```
SELECT *
FROM suppliers
WHERE supplier_name IS NULL;
```

This will return all records from the *suppliers* table where the *supplier_name* contains a null value.

INSERT Statement

Here is an example of how you could use the **IS NULL** condition in an INSERT statement:

```
INSERT INTO suppliers
(supplier_id, supplier_name)
SELECT account_no, name
FROM customers
WHERE city IS NULL;
```

This will insert records into the *suppliers* table where the *city* contains a null value.

UPDATE Statement

Here is an example of how you could use the **IS NULL** condition in an UPDATE statement:

```
UPDATE suppliers
SET name = 'Apple'
WHERE name IS NULL;
```

This will update records in the *suppliers* table where the *name* contains a null value.

DELETE Statement

Here is an example of how you could use the **IS NULL** condition in a DELETE statement:

```
DELETE FROM suppliers
WHERE supplier_name IS NULL;
```

This will delete all records from the *suppliers* table where the *supplier_name* contains a null value.

Oracle/PLSQL: IS NOT NULL

IS NOT NULL is a comparison that evaluates to TRUE if the expression is not a null value. You can use the **IS NOT NULL** condition in either a SQL statement or in a block of PLSQL code.

SQL Statement

You can use the **IS NOT NULL** condition in the following types of SQL statements:

- SELECT statement
- INSERT statement
- UPDATE statement
- DELETE statement

SELECT Statement

Here is an example of how you could use the **IS NOT NULL** condition in a SELECT statement:

```
SELECT *
FROM customers
WHERE customer_name IS NOT NULL;
```

This will return all records from the *customers* table where the *customer_name* does not contain a null value.

INSERT Statement

Here is an example of how you could use the **IS NOT NULL** condition in an INSERT statement:

```
INSERT INTO customers
(customer_id, customer_name)
SELECT account_no, name
FROM customers
WHERE account_no IS NOT NULL;
```

This will insert records into the *customers* table where the *account_no* does not contain a null value.

UPDATE Statement

Here is an example of how you could use the **IS NOT NULL** condition in an UPDATE statement:

```
UPDATE customers
SET status = 'Active'
WHERE customer_name IS NOT NULL;
```

This will update records in the *customers* table where the *customer_name* does not contain a null value.

DELETE Statement

Here is an example of how you could use the **IS NOT NULL** condition in a DELETE statement:

```
DELETE FROM customers
WHERE status IS NOT NULL;
```

This will delete all records from the *customers* table where the *status* does not contain a null value.

Oracle/PLSQL: LIKE Condition

The **Oracle LIKE condition** allows wildcards to be used in the WHERE clause of a SELECT, INSERT, UPDATE, or DELETE statement. This allows you to perform pattern matching.

Oracle LIKE Syntax

The syntax for the **Oracle LIKE Condition** is:

```
expression LIKE pattern;
```

expression is a character expression such as a column or field.

pattern is a character expression that contains pattern matching. 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

Oracle LIKE Example - Using % wildcard (percent sign wildcard)

The first Oracle LIKE example that we will look at involves using the % wildcard (percent sign wildcard).

Let's explain how the % wildcard works in the Oracle LIKE condition. We want to find all of the customers whose last_name begins with 'Ap'.

```
SELECT customer_name
FROM customers
WHERE last_name LIKE 'Ap%';
```

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

```
SELECT customer_name
FROM customers
WHERE last_name LIKE '%er%';
```

In this Oracle LIKE condition example, we are looking for all *customers* whose *last_name* contains the characters 'er'.

Oracle LIKE Example - Using _ wildcard (underscore wildcard)

Next, let's explain how the _ wildcard (underscore wildcard) works in the Oracle LIKE condition. Remember that _ wildcard is looking for only one character.

For example:

```
SELECT supplier_name
FROM suppliers
WHERE supplier_name LIKE 'Sm_th';
```

This Oracle LIKE condition example would return all suppliers whose *supplier_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 *supplier_name* is 'Smith', 'Smyth', 'Smath', 'Smeth', etc.

Here is another example:

```
SELECT *
FROM suppliers
WHERE account_number LIKE '92314_';
```

You might find that you are looking for an account number, but you only have 5 of the 6 digits. The example above, would retrieve potentially 10 records back (where the missing value could equal anything from 0 to 9). For example, it could return suppliers whose account numbers are:

923140, 923141, 923142, 923143, 923144, 923145, 923146, 923147, 923148, 923149

Oracle LIKE Example - Using NOT Operator

Next, let's look at how you would use the Oracle NOT Operator with wildcards.

Let's use the % wilcard with the NOT Operator. You could also use the Oracle LIKE condition to find suppliers whose name does **not** start with 'T'.

For example:

```
SELECT supplier_name
FROM suppliers
WHERE supplier_name NOT LIKE 'W%';
```

By placing the **NOT Operator** in front of the Oracle LIKE condition, you are able to retrieve all suppliers whose *supplier_name* does **not** start with 'W'.

Oracle LIKE Example - Using Escape Characters

It is important to understand how to "Escape Characters" when pattern matching. These examples deal specifically with escaping characters in Oracle.

Let's say you wanted to search for a % or a _ character in the Oracle LIKE condition. You can do this using an Escape character.

Please note that you can only define an escape character as a single character (length of 1).

For example:

```
SELECT *
FROM suppliers
WHERE supplier_name LIKE '!%' escape '!';
```

This Oracle LIKE condition example identifies the ! character as an escape character. This statement will return all suppliers whose name is %.

Here is another more complicated example using escape characters in the Oracle LIKE condition.

```
SELECT *
FROM suppliers
WHERE supplier_name LIKE 'H%!%' escape '!';
```

This Oracle LIKE condition example returns all suppliers whose name starts with H and ends in %. For example, it would return a value such as 'Hello%'.

You can also use the escape character with the _ character in the Oracle LIKE condition.

For example:

```
SELECT *
FROM suppliers
WHERE supplier_name LIKE 'H%!_' escape '!';
```

This Oracle LIKE condition example returns all suppliers whose name starts with H and ends in _. For example, it would return a value such as 'Hello_'.

Frequently Asked Questions (Oracle LIKE)

Question: How do you incorporate the Oracle UPPER function with the Oracle LIKE condition? I'm trying to query against a free text field for all records containing the word "test". The problem is that it can be entered in the following ways: TEST, Test, or test.

Answer: To answer this question, let's look at an example.

Let's say that we have a *suppliers* table with a field called *supplier_name* that contains the values TEST, Test, or test.

If we wanted to find all records containing the word "test", regardless of whether it was stored as TEST, Test, or test, we could run either of the following SELECT statements:

```
SELECT *
FROM suppliers
WHERE UPPER(supplier_name) LIKE ('TEST%');
```

or

```
SELECT *
FROM suppliers
WHERE UPPER(supplier_name) LIKE UPPER('test%')
```

These SELECT statements use a combination of the Oracle UPPER function and the LIKE condition to return all of the records where the *supplier_name* field contains the word "test", regardless of whether it was stored as TEST, Test, or test.

Practice Exercise #1 - Oracle LIKE Condition:

Based on the *employees* table populated with the following data, find all records whose *employee_name* ends with the letter "h".

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);

INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1001, 'John Smith', 62000);

INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1002, 'Jane Anderson', 57500);

INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1003, 'Brad Everest', 71000);

INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1004, 'Jack Horvath', 42000);
```

Solution for Practice Exercise #1 - Oracle LIKE Condition:

The following SELECT statement uses the Oracle LIKE condition to return the records whose *employee_name* ends with the letter "h".

```
SELECT *
FROM employees
WHERE employee_name LIKE '%h';
```

It would return the following result set:

EMPLOYEE_NUMBER	EMPLOYEE_NAME	SALARY
1001	John Smith	62000
1004	Jack Horvath	42000

Practice Exercise #2 - Oracle LIKE Condition:

Based on the *employees* table populated with the following data, find all records whose *employee_name* contains the letter "s".

```
CREATE TABLE employees
( employee_number number(10) not null,
  employee_name varchar2(50) not null,
  salary number(6),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

```
INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1001, 'John Smith', 62000);

INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1002, 'Jane Anderson', 57500);

INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1003, 'Brad Everest', 71000);

INSERT INTO employees (employee_number, employee_name, salary)
VALUES (1004, 'Jack Horvath', 42000);
```

Solution for Practice Exercise #2 - Oracle LIKE Condition:

The following Oracle SELECT statement would use the Oracle LIKE condition to return the records whose *employee_name* contains the letter "s".

```
SELECT *
FROM employees
WHERE employee_name LIKE '%s%';
```

It would return the following result set:

EMPLOYEE_NUMBER	EMPLOYEE_NAME	SALARY
1002	Jane Anderson	57500
1003	Brad Everest	71000

Practice Exercise #3 - Oracle LIKE Condition:

Based on the *suppliers* table populated with the following data, find all records whose *supplier_id* is 4 digits and starts with "500".

```
CREATE TABLE suppliers
( supplier_id varchar2(10) not null,
  supplier_name varchar2(50) not null,
  city varchar2(50),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);

INSERT INTO suppliers(supplier_id, supplier_name, city)
VALUES ('5008', 'Microsoft', 'New York');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES ('5009', 'IBM', 'Chicago');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES ('5010', 'Red Hat', 'Detroit');

INSERT INTO suppliers (supplier_id, supplier_name, city)
VALUES ('5011', 'NVIDIA', 'New York');
```

Solution for Practice Exercise #3 - Oracle LIKE Condition:

The following Oracle SELECT statement would use the Oracle LIKE condition to return the records whose *supplier_id* is 4 digits and starts with "500".

```
SELECT *
FROM suppliers
WHERE supplier_id LIKE '500_';
```

It would return the following result set:

SUPPLIER_ID	SUPPLIER_NAME	CITY
5008	Microsoft	New York
5009	IBM	Chicago

Oracle/PLSQL: EXISTS Condition

The **Oracle EXISTS condition** is considered "to be met" if the subquery returns at least one row. It can be used in a SELECT, INSERT, UPDATE, or DELETE statement.

Oracle EXISTS Syntax

The syntax for the **Oracle EXISTS condition** is:

```
WHERE EXISTS ( subquery );
```

Note

Oracle SQL statements that use the **Oracle EXISTS Condition** are very inefficient since the sub-query is RE-RUN for EVERY row in the outer query's table. There are more efficient ways to write most queries, that do not use the EXISTS Condition.

Oracle EXISTS Example - SELECT Statement

Let's look at a simple example.

The following is an Oracle SELECT statement that uses the Oracle EXISTS condition:

```
SELECT *
FROM customers
WHERE EXISTS (SELECT *
              FROM order_details
              WHERE customers.customer_id = orders.customer_id);
```

This Oracle EXISTS condition example will return all records from the *customers* table where there is at least one record in the *order_details* table with the matching *customer_id*.

Oracle EXISTS Example - SELECT Statement using NOT EXISTS

The Oracle EXISTS condition can also be combined with the Oracle NOT operator.

For example,

```
SELECT *
FROM customers
WHERE NOT EXISTS (SELECT *
                  FROM order_details
                  WHERE customers.customer_id = orders.customer_id);
```

This Oracle EXISTS example will return all records from the *customers* table where there are **no** records in the *order_details* table for the given customer_id.

Oracle EXISTS Example - INSERT Statement

The following is an example of an Oracle INSERT statement that uses the Oracle EXISTS condition:

```
INSERT INTO suppliers
(supplier_id, supplier_name)
SELECT account_no, name
FROM suppliers
WHERE EXISTS (SELECT *
              FROM orders
              WHERE suppliers.supplier_id = orders.supplier_id);
```

Oracle EXISTS Example - UPDATE Statement

The following is an example of an Oracle UPDATE statement that uses the Oracle EXISTS condition:

```
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);
```

Oracle EXISTS Example - DELETE Statement

The following is an example of an Oracle DELETE statement that uses the Oracle EXISTS condition:

```
DELETE FROM suppliers
WHERE EXISTS (SELECT *
              FROM orders
              WHERE suppliers.supplier_id = orders.supplier_id);
```

Oracle/PLSQL: DISTINCT Clause

The Oracle DISTINCT Clause is used to remove duplicates from the result set. The DISTINCT Clause can only be used with SELECT statements.

Oracle DISTINCT Syntax

The syntax for the **Oracle DISTINCT Clause** is:

```
SELECT DISTINCT expressions
FROM tables
WHERE conditions;
```

Oracle DISTINCT Example - Single field

Let's look at the simplest Oracle DISTINCT Clause example. We can use the Oracle DISTINCT Clause to return a single field that removes the duplicates from the result set.

For example:

```
SELECT DISTINCT state
FROM customers;
```

This Oracle DISTINCT example would return all unique *state* values from the *customeres* table.

Oracle DISTINCT Example - Multiple fields

Let's look at how you might use the Oracle DISTINCT Clause to remove duplicates from more than one field in your SELECT statement.

For example:

```
SELECT DISTINCT city, state
FROM customers;
```

This Oracle DISTINCT Clause example would return each unique city and state combination from the *customers* table. In this case, the DISTINCT applies to each field listed after the DISTINCT keyword, and therefore returns distinct combinations.

Oracle/PLSQL: Joins

Oracle JOINS are used to retrieve data from multiple tables. An Oracle JOIN is performed whenever two or more tables are joined in a SQL statement.

There are 4 different types of Oracle joins:

- Oracle INNER JOIN (or sometimes called simple join)
- Oracle LEFT OUTER JOIN (or sometimes called LEFT JOIN)
- Oracle RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)
- Oracle FULL OUTER JOIN (or sometimes called FULL JOIN)

So let's discuss Oracle JOIN syntax, look at visual illustrations of Oracle JOINS, and explore Oracle JOIN examples.

Oracle INNER JOIN (simple join)

Chances are, you've already written a statement that uses an Oracle INNER JOIN. It is the most common type of join. Oracle INNER JOINS return all rows from multiple tables where the join condition is met.

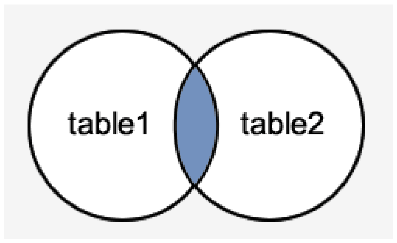
Oracle INNER JOIN Syntax

The syntax for the **Oracle INNER JOIN** is:

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.column = table2.column;
```

Visual Illustration of Oracle INNER JOIN

In this visual diagram, the **Oracle INNER JOIN** returns the shaded area:



The Oracle INNER JOIN would return the records where *table1* and *table2* intersect.

Oracle INNER JOIN Examples

Here is an example of an Oracle INNER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This Oracle INNER JOIN example would return all rows from the suppliers and orders tables where there is a matching supplier_id value in both the suppliers and orders tables.

Let's look at some data to explain how the INNER JOINS work:

We have a table called *suppliers* with two fields (supplier_id and supplier_ name). It contains the following data:

supplier_id	**supplier_name**
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have another table called *orders* with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	**supplier_id**	**order_date**
500125	10000	2003/05/12
500126	10001	2003/05/13
500127	10004	2003/05/14

If we run the Oracle SELECT statement (that contains an INNER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
INNER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	**name**	**order_date**
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13

The rows for *Microsoft* and *NVIDIA* from the supplier table would be omitted, since the supplier_id's 10002 and 10003 do not exist in both tables. The row for 500127 (order_id) from the orders table would be omitted, since the supplier_id 10004 does not exist in the suppliers table.

Old Oracle INNER JOIN Syntax

As a final note, it is worth mentioning that the Oracle INNER JOIN example above could be rewritten using the older implicit syntax as follows (but we still recommend using the INNER JOIN keyword syntax):

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id;
```

Oracle LEFT OUTER JOIN

Another type of join is called an **Oracle OUTER JOIN**. This type of join returns all rows from the LEFT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Oracle LEFT OUTER JOIN Syntax

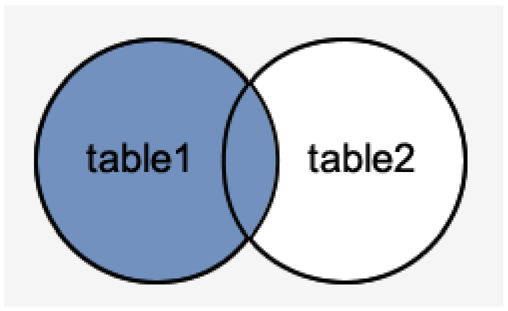
The syntax for the **Oracle LEFT OUTER JOIN** is:

```
SELECT columns
FROM table1
LEFT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the LEFT OUTER JOIN keywords are replaced with LEFT JOIN.

Visual Illustration of Oracle LEFT OUTER JOIN

In this visual diagram, the **Oracle LEFT OUTER JOIN** returns the shaded area:



The Oracle LEFT OUTER JOIN would return the all records from *table1* and only those records from *table2* that intersect with *table1*.

Oracle LEFT OUTER JOIN Examples

Here is an example of an Oracle LEFT OUTER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
LEFT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This LEFT OUTER JOIN example would return all rows from the suppliers table and only those rows from the orders table where the joined fields are equal.

If a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set.

Let's look at some data to explain how LEFT OUTER JOINS work:

We have a table called **suppliers** with two fields (supplier_id and name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have a second table called **orders** with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2003/05/12
500126	10001	2003/05/13

If we run the SELECT statement (that contains a LEFT OUTER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
LEFT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	supplier_name	order_date
10000	IBM	2003/05/12
10001	Hewlett Packard	2003/05/13
10002	Microsoft	<null>
10003	NVIDIA	<null>

The rows for *Microsoft* and *NVIDIA* would be included because a LEFT OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

Old Oracle LEFT OUTER JOIN Syntax

As a final note, it is worth mentioning that the LEFT OUTER JOIN example above could be rewritten using the older implicit syntax that utilizes the outer join operator (+) as follows (but we still recommend using the LEFT OUTER JOIN keyword syntax):

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id(+);
```

Oracle RIGHT OUTER JOIN

Another type of join is called an **Oracle RIGHT OUTER JOIN**. This type of join returns all rows from the RIGHT-hand table specified in the ON condition and **only** those rows from the other table where the joined fields are equal (join condition is met).

Oracle RIGHT OUTER JOIN Syntax

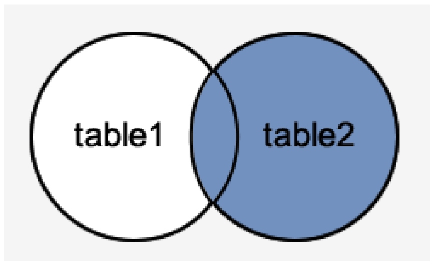
The syntax for the **Oracle RIGHT OUTER JOIN** is:

```
SELECT columns
FROM table1
RIGHT [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the RIGHT OUTER JOIN keywords are replaced with RIGHT JOIN.

Visual Illustration of Oracle RIGHT OUTER JOIN

In this visual diagram, the **Oracle RIGHT OUTER JOIN** returns the shaded area:



The Oracle RIGHT OUTER JOIN would return the all records from *table2* and only those records from *table1* that intersect with *table2*.

Oracle RIGHT OUTER JOIN Examples

Here is an example of an Oracle RIGHT OUTER JOIN:

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers
RIGHT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This RIGHT OUTER JOIN example would return all rows from the orders table and only those rows from the suppliers table where the joined fields are equal.

If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how RIGHT OUTER JOINS work:

We have a table called **suppliers** with two fields (supplier_id and name). It contains the following data:

supplier_id	**supplier_name**
10000	Apple
10001	Google

We have a second table called **orders** with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	**supplier_id**	**order_date**
500125	10000	2013/08/12

500126	10001	2013/08/13
500127	10002	2013/08/14

If we run the SELECT statement (that contains a RIGHT OUTER JOIN) below:

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers
RIGHT OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

order_id	**order_date**	**supplier_name**
500125	2013/08/12	Apple
500126	2013/08/13	Google
500127	2013/08/14	<null>

The row for *500127* (order_id) would be included because a RIGHT OUTER JOIN was used. However, you will notice that the supplier_name field for that record contains a <null> value.

Old Oracle RIGHT OUTER JOIN Syntax

As a final note, it is worth mentioning that the RIGHT OUTER JOIN example above could be rewritten using the older implicit syntax that utilizes the outer join operator (+) as follows (but we still recommend using the RIGHT OUTER JOIN keyword syntax):

```
SELECT orders.order_id, orders.order_date, suppliers.supplier_name
FROM suppliers, orders
WHERE suppliers.supplier_id(+) = orders.supplier_id;
```

Oracle FULL OUTER JOIN

Another type of join is called an **Oracle FULL OUTER JOIN**. This type of join returns all rows from the LEFT-hand table and RIGHT-hand table with nulls in place where the join condition is not met.

Oracle FULL OUTER JOIN Syntax

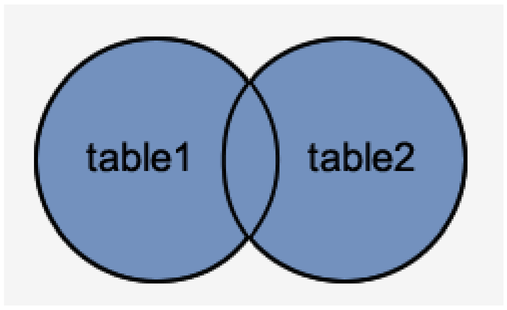
The syntax for the **Oracle FULL OUTER JOIN** is:

```
SELECT columns
FROM table1
FULL [OUTER] JOIN table2
ON table1.column = table2.column;
```

In some databases, the FULL OUTER JOIN keywords are replaced with FULL JOIN.

Visual Illustration of Oracle FULL OUTER JOIN

In this visual diagram, the **Oracle FULL OUTER JOIN** returns the shaded area:



The Oracle FULL OUTER JOIN would return the all records from both *table1* and *table2*.

Oracle FULL OUTER JOIN Examples

Here is an example of an Oracle FULL OUTER JOIN:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
FULL OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

This FULL OUTER JOIN example would return all rows from the suppliers table and all rows from the orders table and whenever the join condition is not met, <nulls> would be extended to those fields in the result set.

If a supplier_id value in the suppliers table does not exist in the orders table, all fields in the orders table will display as <null> in the result set. If a supplier_id value in the orders table does not exist in the suppliers table, all fields in the suppliers table will display as <null> in the result set.

Let's look at some data to explain how FULL OUTER JOINS work:

We have a table called **suppliers** with two fields (supplier_id and name). It contains the following data:

supplier_id	supplier_name
10000	IBM
10001	Hewlett Packard
10002	Microsoft
10003	NVIDIA

We have a second table called **orders** with three fields (order_id, supplier_id, and order_date). It contains the following data:

order_id	supplier_id	order_date
500125	10000	2013/08/12
500126	10001	2013/08/13
500127	10004	2013/08/14

If we run the SELECT statement (that contains a FULL OUTER JOIN) below:

```
SELECT suppliers.supplier_id, suppliers.supplier_name, orders.order_date
FROM suppliers
FULL OUTER JOIN orders
ON suppliers.supplier_id = orders.supplier_id;
```

Our result set would look like this:

supplier_id	supplier_name	order_date
10000	IBM	2013/08/12
10001	Hewlett Packard	2013/08/13
10002	Microsoft	<null>
10003	NVIDIA	<null>
<null>	<null>	2013/08/14

The rows for *Microsoft* and *NVIDIA* would be included because a FULL OUTER JOIN was used. However, you will notice that the order_date field for those records contains a <null> value.

The row for supplier_id 10004 would be also included because a FULL OUTER JOIN was used. However, you will notice that the supplier_id and supplier_name field for those records contain a <null> value.

Old Oracle FULL OUTER JOIN Syntax

As a final note, it is worth mentioning that the FULL OUTER JOIN example above could not have been written in the old syntax without using a UNION query.

Oracle/PLSQL: ORDER BY Clause

The **Oracle ORDER BY clause** is used to sort the records in your result set. The ORDER BY clause can only be used in SELECT statements.

Oracle ORDER BY Syntax

The syntax for the **Oracle ORDER BY clause** is:

```
SELECT columns
FROM tables
WHERE conditions
ORDER BY column ASC/DESC;
```

The Oracle ORDER BY clause sorts the result set based on the columns specified. If the ASC or DESC value is omitted, it is sorted by ASC.

ASC indicates ascending order. (default)
DESC indicates descending order.

Oracle ORDER BY Example - Sorting without using ASC/DESC attribute

The Oracle ORDER BY clause can be used without specifying the ASC or DESC value. When this attribute is omitted from the ORDER BY clause, the sort order is defaulted to ASC or ascending order.

For example:

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'Microsoft'
ORDER BY supplier_city;
```

This Oracle ORDER BY example would return all records sorted by the *supplier_city* field in ascending order and would be equivalent to the following ORDER BY clause:

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'Microsoft'
ORDER BY supplier_city ASC;
```

Most programmers omit the ASC attribute if sorting in ascending order.

Oracle ORDER BY Example - Sorting in descending order

When sorting your result set in descending order, you use the DESC attribute in your ORDER BY clause as follows:

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'Microsoft'
ORDER BY supplier_city DESC;
```

This Oracle ORDER BY example would return all records sorted by the *supplier_city* field in descending order.

Oracle ORDER BY Example - Sorting by relative position

You can also use the Oracle ORDER BY clause to sort by relative position in the result set, where the first field in the result set is 1. The next field is 2, and so on.

For example:

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'Microsoft'
ORDER BY 1 DESC;
```

This Oracle ORDER BY would return all records sorted by the *supplier_city* field in descending order, since the *supplier_city* field is in position #1 in the result set and would be equivalent to the following ORDER BY clause:

```
SELECT supplier_city
FROM suppliers
WHERE supplier_name = 'Microsoft'
ORDER BY supplier_city DESC;
```

Oracle ORDER BY Example - Using both ASC and DESC attributes

When sorting your result set using the Oracle ORDER BY clause, you can use the ASC and DESC attributes in a single SELECT statement.

For example:

```
SELECT supplier_city, supplier_state
FROM suppliers
WHERE supplier_name = 'Microsoft'
ORDER BY supplier_city DESC, supplier_state ASC;
```

This Oracle ORDER BY would return all records sorted by the *supplier_city* field in descending order, with a secondary sort by *supplier_state* in ascending order.

Oracle/PLSQL: COUNT Function

The Oracle/PLSQL **COUNT function** returns the number of rows in a query.

Oracle COUNT Syntax

The syntax for the Oracle/PLSQL **COUNT function** is:

```
SELECT COUNT( expression )
FROM tables
WHERE conditions;
```

Parameters or Arguments

expression can be a numeric field or formula.

Only includes NOT NULL Values

Not everyone realizes this, but the **COUNT function** will only include the records in the count where the value of *expression* in COUNT(*expression*) is NOT NULL. When *expression* contains a NULL value, it is not included in the COUNT calculations.

Let's look at a COUNT function example that demonstrates how NULL values are evaluated by the COUNT function.

For example, if you have the following table called *suppliers*:

supplier_id	**supplier_name**	**state**
1	IBM	CA
2	Microsoft	
3	NVIDIA	

And if you ran the following SELECT statement that uses the COUNT function:

```
SELECT COUNT(supplier_id)
FROM suppliers;
```

This COUNT example will return 3 since all *supplier_id* values in the query's result set are NOT NULL.

However, if you ran the next SELECT statement that uses the COUNT function:

```
SELECT COUNT(state)
FROM suppliers;
```

This COUNT example will only return 1, since only one *state* value in the query's result set is NOT NULL. That would be the first row where the state = 'CA'. It is the only row that is included in the COUNT function calculation.

Applies To

The **COUNT function** can be used in the following versions of Oracle/PLSQL:

- Oracle 12c, Oracle 11g, Oracle 10g, Oracle 9i, Oracle 8i

Oracle COUNT Example - Single Field

Let's look at some Oracle COUNT function examples and explore how you would use the COUNT function in Oracle/PLSQL.

For example, you might wish to know how many employees have a salary above $75,000 / year.

```
SELECT COUNT(*) as "Number of employees"
FROM employees
WHERE salary > 75000;
```

In this COUNT function example, we've aliased the COUNT(*) expression as "Number of employees". As a result, "Number of employees" will display as the field name when the result set is returned.

Oracle COUNT Example - Using DISTINCT

You can use the DISTINCT clause within the **COUNT function**. For example, the SQL statement below returns the number of unique departments where at least one employee makes over $55,000 / year.

```
SELECT COUNT(DISTINCT department) as "Unique departments"
FROM employees
WHERE salary > 55000;
```

Again, the COUNT(DISTINCT department) field is aliased as "Unique departments". This is the field name that will display in the result set.

Oracle COUNT Example - Using GROUP BY

In some cases, you will be required to use the GROUP BY clause with the COUNT function.

For example, you could also use the COUNT function to return the name of the department and the number of employees (in the associated department) that are in the state of 'CA'.

```
SELECT department, COUNT(*) as "Number of employees"
FROM employees
WHERE state = 'CA'
GROUP BY department;
```

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

Oracle/PLSQL: SUM Function

The Oracle/PLSQL **SUM function** returns the summed value of an expression.

Oracle SUM Syntax

The syntax for the Oracle/PLSQL **SUM function** is:

```
SELECT SUM( expression )
FROM tables
WHERE conditions;
```

Parameters or Arguments

expression can be a numeric field or formula.

Applies To

The **SUM function** can be used in the following versions of Oracle/PLSQL:

- Oracle 12c, Oracle 11g, Oracle 10g, Oracle 9i, Oracle 8i

Oracle SUM Example - Single Field

Let's look at some Oracle SUM function examples and explore how you would use the SUM function in Oracle/PLSQL.

For example, you might wish to know how the combined total salary of all employees whose salary is above $50,000 / year.

```
SELECT SUM(salary) as "Total Salary"
FROM employees
WHERE salary > 50000;
```

In this SUM function example, we've aliased the SUM(salary) expression as "Total Salary". As a result, "Total Salary" will display as the field name when the result set is returned.

Oracle SUM Example - Using DISTINCT

You can use the DISTINCT clause within the **SUM function**. For example, the SQL statement below returns the combined total salary of unique salary values where the salary is above $50,000 / year.

```
SELECT SUM(DISTINCT salary) as "Total Salary"
FROM employees
WHERE salary > 50000;
```

If there were two salaries of $82,000/year, only one of these values would be used in the **SUM function**.

Oracle SUM Example - Using Formula

The *expression* contained within the **SUM function** does not need to be a single field. You could also use a formula. For example, you might want to calculate the total commission.

```
SELECT SUM(sales * 0.05) as "Total Commission"
FROM orders;
```

Oracle SUM Example - Using GROUP BY

In some cases, you will be required to use the GROUP BY clause with 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.

Oracle/PLSQL: MIN Function

The Oracle/PLSQL **MIN function** returns the minimum value of an expression.

Oracle MIN Syntax

The syntax for the Oracle/PLSQL **MIN function** is:

```
SELECT MIN( expression )
FROM tables
WHERE conditions;
```

Parameters or Arguments

expression can be a numeric field or formula.

Applies To

The **MIN function** can be used in the following versions of Oracle/PLSQL:

- Oracle 12c, Oracle 11g, Oracle 10g, Oracle 9i, Oracle 8i

Oracle MIN Example - Single Field

Let's look at some Oracle MIN function examples and explore how you would use the MIN function in Oracle/PLSQL.

For example, you might wish to know how the minimum salary of all employees.

```
SELECT MIN(salary) as "Lowest Salary"
FROM employees;
```

In this MIN function example, we've aliased the MIN(salary) expression as "Lowest Salary". As a result, "Lowest Salary" will display as the field name when the result set is returned.

Oracle MIN Example - Using GROUP BY

In some cases, you will be required to use the GROUP BY clause with the MIN function.

For example, you could also use the MIN function to return the name of the department and the minimum salary in the department.

```
SELECT department, MIN(salary) as "Lowest salary"
FROM employees
GROUP BY department;
```

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

Oracle/PLSQL: MAX Function

The Oracle/PLSQL **MAX function** returns the maximum value of an expression.

Oracle MAX Syntax

The syntax for the Oracle/PLSQL **MAX function** is:

```
SELECT MAX( expression )
FROM tables
WHERE conditions;
```

Parameters or Arguments

expression can be a numeric field or formula.

Applies To

The **MAX function** can be used in the following versions of Oracle/PLSQL:

- Oracle 12c, Oracle 11g, Oracle 10g, Oracle 9i, Oracle 8i

Oracle MAX Example - Single Field

Let's look at some Oracle MAX function examples and explore how you would use the MAX function in Oracle/PLSQL.

For example, you might wish to know how the maximum salary of all employees.

```
SELECT MAX(salary) as "Highest Salary"
FROM employees;
```

In this MAX function example, we've aliased the MAX(salary) expression as "Highest Salary". As a result, "Highest Salary" will display as the field name when the result set is returned.

Oracle MAX Example - Using GROUP BY

In some cases, you will be required to use the GROUP BY clause with the MAX function.

For example, you could also use the MAX function to return the name of the department and the maximum salary in the department.

```
SELECT department, MAX(salary) as "Highest salary"
FROM employees
GROUP BY department;
```

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

Frequently Asked Questions (Oracle MAX Function)

Question: I'm trying to pull some info out of a table. To simplify, let's say the table (report_history) has 4 columns: user_name, report_job_id, report_name, and report_run_date.

Each time a report is run in Oracle, a record is written to this table noting the above info. What I am trying to do is pull from this table when the last time each distinct report was run and who ran it last.

My initial query:

```
SELECT report_name, MAX(report_run_date)
FROM report_history
GROUP BY report_name
```

runs fine. However, it does not provide the name of the user who ran the report.

Adding user_name to both the select list and to the group by clause returns multiple lines for each report; the results show the last time each person ran each report in question. (i.e. User1 ran Report 1 on 01-JUL-03, User2 ran Report1 on 01-AUG-03). I don't want that....I just want to know who ran a particular report the last time it was run.

Any suggestions?

Answer: This is where things get a bit complicated. The SQL SELECT statement below will return the results that you want:

```
SELECT rh.user_name, rh.report_name, rh.report_run_date
FROM report_history rh,
  (SELECT MAX(report_run_date) AS maxdate, report_name
   FROM report_history
   GROUP BY report_name) maxresults
WHERE rh.report_name = maxresults.report_name
AND rh.report_run_date= maxresults.maxdate;
```

Let's take a few moments to explain what we've done.

First, we've aliased the first instance of the report_history table as rh.

Second, we've included two components in our FROM clause. The first is the table called report_history (aliased as rh). The second is a select statement:

```
(SELECT MAX(report_run_date) AS maxdate, report_name
 FROM report_history
 GROUP BY report_name) maxresults
```

We've aliased the max(report_run_date) as *maxdate* and we've aliased the entire result set as *maxresults*.

Now, that we've created this select statement within our FROM clause, Oracle will let us join these results against our original report_history table. So we've joined the report_name and report_run_date fields between the tables called *rh* and *maxresults*. This allows us to retrieve the report_name, max(report_run_date) as well as the user_name.

Question: I need help with a SQL query. I have a table in Oracle called *orders* which has the following fields: order_no, customer, and amount.

I need a query that will return the customer who has ordered the highest total amount.

Answer: The following SQL should return the customer with the highest total amount in the orders table.

```
SELECT query1.*
FROM (SELECT customer, SUM(orders.amount) AS total_amt
      FROM orders
      GROUP BY orders.customer) query1,

     (SELECT MAX(query2.total_amt) AS highest_amt
      FROM (SELECT customer, SUM(orders.amount) AS total_amt
            FROM orders
            GROUP BY orders.customer) query2) query3
WHERE query1.total_amt = query3.highest_amt;
```

This SQL SELECT statement will summarize the total orders for each customer and then return the customer with the highest total orders. This syntax is optimized for Oracle and may not work for other database technologies.

Question: I'm trying to retrieve some info from an Oracle database. I've got a table named *Scoring* with two fields - Name and Score. What I want to get is the highest score from the table and the name of the player.

Answer: The following SQL SELECT statement should work:

```
SELECT Name, Score
FROM Scoring
WHERE Score = (SELECT MAX(Score) FROM Scoring);
```

Question: I need help in a SQL query. I have a table in Oracle called *cust_order* which has the following fields: OrderNo, Customer_id, Order_Date, and Amount.

I would like to find the customer_id, who has Highest order count.

I tried with following query.

```
SELECT MAX(COUNT(*))
FROM CUST_ORDER
GROUP BY CUSTOMER_ID;
```

This gives me the max Count, But, I can't get the CUSTOMER_ID. Can you help me please?

Answer: The following SQL SELECT statement should return the customer with the highest order count in the cust_order table.

```
SELECT query1.*
FROM (SELECT Customer_id, Count(*) AS order_count
      FROM cust_order
      GROUP BY cust_order.Customer_id) query1,

     (SELECT max(query2.order_count) AS highest_count
      FROM (SELECT Customer_id, Count(*) AS order_count
            FROM cust_order
            GROUP BY cust_order.Customer_id) query2) query3
WHERE query1.order_count = query3.highest_count;
```

This SQL SELECT statement will summarize the total orders for each customer and then return the customer with the highest order count. This syntax is optimized for Oracle and may not work for other database technologies.

Question: I'm trying to get the employee with the maximum salary from department 30, but I need to display the employee's full information. I've tried the following query, but it returns the result from both department 30 and 80:

```
SELECT *
FROM employees
WHERE salary = (SELECT MAX(salary)
                FROM employees
                WHERE department_id=30);
```

Answer: The SQL SELECT statement that you have written will first determine the maximum salary for department 30, but then you select all employees that have this salary. In your case, you must have 2 employees (one in department 30 and another in department 80) that have this same salary. You need to make sure that you are refining your query results to only return employees from department 30.

Try using this SQL SELECT statement:

```
SELECT *
FROM employees
WHERE department_id=30
AND salary = (SELECT MAX(salary)
              FROM employees
              WHERE department_id=30);
```

This will return the employee information for only the employee in department 30 that has the highest salary.

Oracle/PLSQL: GROUP BY Clause

The **Oracle GROUP BY Clause** can be used in a SELECT statement to collect data across multiple records and group the results by one or more columns.

Oracle GROUP BY Syntax

The syntax for the **Oracle GROUP BY Clause** is:

```
SELECT expression1, expression2, ... expression_n,
       aggregate_function (expression)
FROM tables
WHERE conditions
GROUP BY expression1, expression2, ... expression_n;
```

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

expression1, *expression2*, ... *expression_n* are expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY Clause.

Oracle GROUP BY Example - Using SUM function

Let's look at an Oracle GROUP BY query example that uses the SUM function.

This Oracle GROUP BY example uses the SUM function to return the name of the product and the total sales (for the product).

```
SELECT product, SUM(sale) AS "Total sales"
FROM order_details
GROUP BY product;
```

Because you have listed one column (the product field) in your SELECT statement that is not encapsulated in the SUM function, you must use the GROUP BY Clause. The *product* field must, therefore, be listed in the GROUP BY clause.

Oracle GROUP BY Example - Using COUNT function

Let's look at how we could use the GROUP BY clause with the COUNT function.

This GROUP BY example uses the COUNT function returns the category and the number of suppliers (in that category) that have over 45 available_products.

```
SELECT category, COUNT(*) AS "Number of suppliers"
FROM suppliers
WHERE available_products > 45
GROUP BY category;
```

Oracle GROUP BY Example - Using MIN function

Let's next look at how we could use the GROUP BY clause with the MIN function.

This GROUP BY example uses the MIN function to return the name of each department and the minimum salary in the department.

```
SELECT department, MIN(salary) AS "Lowest salary"
FROM employees
GROUP BY department;
```

Oracle GROUP BY Example - Using MAX function

Finally, let's look at how we could use the GROUP BY clause with the MAX function.

This GROUP BY example uses the MAX function to return the name of each department and the maximum salary in the department.

```
SELECT department, MAX(salary) AS "Highest salary"
FROM employees
GROUP BY department;
```

Oracle/PLSQL: HAVING Clause

The **Oracle HAVING Clause** is used in combination with the GROUP BY Clause to restrict the groups of returned rows to only those whose the condition is TRUE.

Oracle HAVING Syntax

The syntax for the **Oracle HAVING Clause** is:

```
SELECT expression1, expression2, ... expression_n,
       aggregate_function (expression)
FROM tables
WHERE conditions
GROUP BY expression1, expression2, ... expression_n
HAVING condition;
```

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

expression1, *expression2*, *... expression_n* are expressions that are not encapsulated within an aggregate function and must be included in the GROUP BY Clause.

condition is the condition that is used to restrict the groups of returned rows. Only those groups whose condition evaluates to TRUE will be included in the result set.

Oracle HAVING Example - Using SUM function

Let's look at an Oracle HAVING Clause example that uses the SUM function.

You could also use the SUM function to return the name of the department and the total sales (in the associated department). The Oracle HAVING Clause will filter the results so that only departments with sales greater than $25,000 will be returned.

```
SELECT department, SUM(sales) AS "Total sales"
FROM order_details
GROUP BY department
HAVING SUM(sales) > 25000;
```

Oracle HAVING Example - Using COUNT function

Let's look at how we could use the HAVING clause with the COUNT function.

You could use the COUNT function to return the name of the department and the number of employees (in the associated department) that make under $49,500 / year. The Oracle HAVING Clause will filter the results so that only departments with more than 10 employees will be returned.

```
SELECT department, COUNT(*) AS "Number of employees"
FROM employees
WHERE salary < 49500
GROUP BY department
HAVING COUNT(*) > 10;
```

Oracle HAVING Example - Using MIN function

Let's next look at how we could use the HAVING clause with the MIN function.

You could also use the MIN function to return the name of each department and the minimum salary in the department. The Oracle HAVING Clause will return only those departments where the starting salary is $42,000.

```
SELECT department, MIN(salary) AS "Lowest salary"
FROM employees
GROUP BY department
HAVING MIN(salary) = 42000;
```

Oracle HAVING Example - Using MAX function

Finally, let's look at how we could use the HAVING clause with the MAX function.

For example, you could also use the MAX function to return the name of each department and the maximum salary in the department. The Oracle HAVING Clause will return only those departments whose maximum salary is less than $45,000.

```
SELECT department, MAX(salary) AS "Highest salary"
FROM employees
GROUP BY department
HAVING MAX(salary) > 45000;
```

SQL: CREATE TABLE AS Statement

You can also use the **SQL CREATE TABLE AS statement** to create a table from an existing table by copying the existing table's columns.

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 SQL SELECT Statement).

SQL CREATE TABLE AS Syntax - Copying all columns from another table

The syntax for the **SQL CREATE TABLE AS statement** copying all of the columns is:

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

For Example:

```
CREATE TABLE suppliers
AS (SELECT *
    FROM companies
    WHERE id > 1000);
```

This would create a new table called **suppliers** that included all columns from the **companies** table.

If there were records in the **companies** table, then the new suppliers table would also contain the records selected by the SELECT statement.

SQL CREATE TABLE AS Syntax - Copying selected columns from another table

The syntax for the **CREATE TABLE AS statement** copying the selected columns is:

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

For Example:

```
CREATE TABLE suppliers
  AS (SELECT id, address, city, state, zip
      FROM companies
      WHERE id > 1000);
```

This would create a new table called **suppliers**, but the new table would only include the specified columns from the **companies** table.

Again, if there were records in the **companies** table, then the new suppliers table would also contain the records selected by the SELECT statement.

SQL CREATE TABLE AS Syntax - Copying selected columns from multiple tables

The syntax for the **CREATE TABLE AS statement** copying columns from multiple tables is:

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

For Example:

```
CREATE TABLE suppliers
  AS (SELECT companies.id, companies.address, categories.cat_type
      FROM companies, categories
      WHERE companies.id = categories.id
      AND companies.id > 1000);
```

This would create a new table called **suppliers** based on columns from both the **companies** and **categories** tables.

Frequently Asked Questions (SQL CREATE table AS)

Question: How can I create a SQL table from another table without copying any values from the old table?

Answer: To do this, the SQL CREATE TABLE syntax is:

```
CREATE TABLE new_table
  AS (SELECT *
      FROM old_table WHERE 1=2);
```

For example:

```
CREATE TABLE suppliers
  AS (SELECT *
      FROM companies WHERE 1=2);
```

This would create a new table called *suppliers* that included all columns from the *companies* table, but no data from the *companies* table.

Acknowledgements: We'd like to thank Daniel W. for providing this solution!