<http://www.thedataanalysis.com/sql/sql-programming.html>

SQL: [**UPDATE**](http://www.thedataanalysis.com/sql/sql-programming/68-sql-update-statement-how-to-use-sql-update-statement.html) Statement

The **UPDATE** statement allows you to update a single record or multiple records in a table.

The syntax for the [**UPDATE**](http://www.thedataanalysis.com/sql/sql-programming/68-sql-update-statement-how-to-use-sql-update-statement.html) statement is:

**UPDATE** table

SET column = expression

**WHERE** predicates;

**Example #1 - Simple example**

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

**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');

**UPDATE** suppliers

SET name = 'HP'

**WHERE** name = 'IBM';

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

**Example #2 - More complex example**

You can also perform more complicated updates.

You may wish to update records in one table based on values in another table. Since you can't list more than one table in the **UPDATE** statement, you can use the EXISTS clause.

**For example:**

**UPDATE** suppliers

SET supplier\_name =

 ( **SELECT** customers.name

 **FROM** customers

 **WHERE** customers.customer\_id = suppliers.supplier\_id)

**WHERE** EXISTS

 ( **SELECT** customers.name

 **FROM** customers

 **WHERE** customers.customer\_id = suppliers.supplier\_id);

Whenever a supplier\_id matched a customer\_id value, the supplier\_name would be overwritten to the customer name from the customers table.

**Practice Exercise #1:**

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

*Solution:* The following SQL statement would perform this update.

**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:**

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:*

The following SQL statement would perform this update.

**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 |

**SQL: SUM Function**

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

*The syntax for the SUM function is*:

**SELECT** *SUM*(expression )

**FROM** tables

**WHERE** predicates;

Expression can be a numeric field or formula.

**Simple Example**

For example, you might wish to know how the combined total salary of all employees whose salary is above $25,000 / year.

**SELECT** *SUM*(salary) *AS* "Total Salary"

**FROM** employees

**WHERE** salary > 25000;

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

**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 $25,000 / year.

**SELECT** *SUM*(DISTINCT salary) *AS* "Total Salary"

**FROM** employees

**WHERE** salary > 25000;

If there were two salaries of $30,000/year, only one of these values would be used in the SUM function.

**Example using a 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 the net income for a business. Net Income is calculated as total income less total expenses.

**SELECT** *SUM*(income - expenses) *AS* "Net Income"

**FROM** gl\_transactions;

You might also want to perform a mathematical operation within a SUM function. For example, you might determine total commission as 10% of total sales.

**SELECT** *SUM*(sales \* 0.10) *AS* "Commission"

**FROM** order\_details;

**Example using GROUP BY**

In some cases, you will be required to use a **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.

**SQL: "IN" Function**

The IN function helps reduce the need to use multiple OR conditions.

The syntax for the IN function is:

**SELECT** columns

**FROM** tables

**WHERE** column1 in (value1, value2, .... value\_n);

This SQL statement will return the records where column1 is value1, value2..., or value\_n.

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

**Example #1**

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

**SELECT** \*

**FROM** suppliers

**WHERE** supplier\_name in ( 'IBM', 'Hewlett Packard', 'Microsoft');

This would return all rows where the supplier\_name is either IBM, Hewlett Packard, or Microsoft.

Because the \* is used in the select, all fields from the suppliers table would appear in the result set.

It is equivalent to the following statement:

SELECT \*

**FROM** suppliers

**WHERE** supplier\_name = 'IBM'

*OR* supplier\_name = 'Hewlett Packard'

*OR* supplier\_name = 'Microsoft';

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

**Example #2**

You can also use the IN function with numeric values.

**SELECT** \*

**FROM** orders

**WHERE** order\_id *IN* (10000, 10001, 10003, 10005);

This SQL statement would return all orders where the order\_id is either 10000, 10001, 10003, or 10005.

It is equivalent to the following statement:

**SELECT** \*

**FROM** orders

**WHERE** order\_id = 10000

*OR* order\_id = 10001 *OR* order\_id = 10003 *OR* order\_id = 10005;

**Example #3 using "NOT IN"**

The IN function can also be combined with the NOT operator.

For example,

**SELECT** \*

**FROM** suppliers

**WHERE** supplier\_name NOT IN ( 'IBM', 'Hewlett Packard', 'Microsoft');

This would return all rows where the supplier\_name is neither 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.

**SQL: HAVING Clause**

The HAVING clause is used in combination with the GROUP BY clause. It can be used in a SELECT statement to filter the records that a GROUP BY returns.

The syntax for the HAVING clause is:

**SELECT** column1, column2, column3,..col\_n, aggregate\_function (expression)

**FROM** tables

**WHERE** predicates

**GROUP BY** column1, column2, ... column\_n

**HAVING** condition1 ... condition\_n;

aggregate\_function can be a function such as SUM, [**Count**](http://www.thedataanalysis.com/sql/sql-programming/62-sql-count-function-how-to-use-sql-count-function-.html), [**MIN**](http://www.thedataanalysis.com/sql/sql-programming/54-sql-min-function-how-to-use-sql-min-function-.html)or [**MAX**](http://www.thedataanalysis.com/sql/sql-programming/53-sql-max-function-how-to-use-sql-max-function-.html).

**Example using the SUM function**

For example, you could also use the SUM function to return the name of the department and the total sales (in the associated department). The HAVING clause will filter the results so that only departments with sales greater than $1000 will be returned.

**SELECT** department, *SUM*(sales) as "Total sales"

**FROM** order\_details

**GROUP BY** department

**HAVING** *SUM*(sales) > 1000;

**Example using the COUNT function**

For example, you could use the COUNT function to return the name of the department and the number of employees (in the associated department) that make over $25,000 / year. The 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 > 25000

**GROUP BY** department

**HAVING** *COUNT*(\*) > 10;

**Example using the MIN function**

For example, you could also use the MIN function to return the name of each department and the minimum salary in the department. The HAVING clause will return only those departments where the starting salary is $35,000.

**SELECT** department, *MIN*(salary) as "Lowest salary"

**FROM** employees

**GROUP BY** department

**HAVING** *MIN*(salary) = 35000;

**Example using 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 HAVING clause will return only those departments whose maximum salary is less than $50,000.

**SELECT** department, *MAX*(salary) as "Highest salary"

**FROM** employees

**GROUP BY** department

**HAVING** *MAX*(salary) < 50000;

**SQL: GROUP BY Clause**

The [**GROUP BY**](http://www.thedataanalysis.com/sql/sql-programming/64-sql-group-by-clause-how-to-use-sql-group-clause-.html) clause can be used in a [**SELECT**](http://www.thedataanalysis.com/sql/sql-programming/50-sql-select-statement-how-to-use-sql-select-statement-.html)statement to collect data across multiple records and group the results by one or more columns. The syntax for the GROUP BY clause is:

**SELECT** column1, column2,column\_n, aggregate\_function (expression)

**FROM** tables

**WHERE** predicates

**GROUP BY** column1, column2, ... column\_n;

aggregate\_function can be a function such as SUM, [**Count**](http://www.thedataanalysis.com/sql/sql-programming/62-sql-count-function-how-to-use-sql-count-function-.html), [**MIN**](http://www.thedataanalysis.com/sql/sql-programming/54-sql-min-function-how-to-use-sql-min-function-.html)or [**MAX**](http://www.thedataanalysis.com/sql/sql-programming/53-sql-max-function-how-to-use-sql-max-function-.html).

**Example using the SUM function**

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

**SELECT** department, *SUM*(sales) as "Total sales"

**FROM** order\_details

**GROUP BY** department;

Because you have listed one column in your [**SELECT**](http://www.thedataanalysis.com/sql/sql-programming/50-sql-select-statement-how-to-use-sql-select-statement-.html)statementthat is not encapsulated in the SUM function, you must use a GROUP BY clause.

The department field must, therefore, be listed in the GROUP BY section.

**Example using the COUNT function**

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

**SELECT** department, *COUNT*(\*) as "Number of employees"

**FROM** employees

**WHERE** salary > 25000

**GROUP BY** department;

**Example using the** [**MIN**](http://www.thedataanalysis.com/sql/sql-programming/54-sql-min-function-how-to-use-sql-min-function-.html) **function**

For example, you could also use 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;

**Example using the** [**MAX**](http://www.thedataanalysis.com/sql/sql-programming/53-sql-max-function-how-to-use-sql-max-function-.html) **function**

For example, you could also use 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;

**SQL: EXISTS Condition**

The EXISTS condition is considered "to be met" if the subquery returns at least one row. The syntax for the EXISTS condition is:

SELECT columns

FROM tables

WHERE EXISTS **(** subquery **)**;

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

**Example #1**

Let's take a look at a simple example.

*The following is an SQL statement that uses the EXISTS condition:*

SELECT \*

FROM suppliers

WHERE EXISTS

 **(**select \*

 from orders

 where suppliers.supplier\_id = orders.supplier\_id**)**;

This select statement will return all records from the suppliers table where there is at least one record in the orders table with the same supplier\_id.

**Example #2 - NOT EXISTS**

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

For example,

SELECT \*

FROM suppliers

WHERE not exists **(**select \* from orders

 Where suppliers.supplier\_id = orders.supplier\_id**)**;

This will return all records from the suppliers table where there are no records in the orders table for the given supplier\_id.

**Example #3 - DELETE Statement**

The following is an example of a delete statement that utilizes the EXISTS condition:

DELETE FROM suppliers

WHERE EXISTS

 **(select** \*

 **from** orders

 **where** suppliers.supplier\_id = orders.supplier\_id**)**;

**Example #4 - UPDATE Statement**

The following is an example of an update statement that utilizes the 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**)**;

**Example #5 - INSERT Statement**

The following is an example of an insert statement that utilizes the 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);

**SQL: COUNT Function**

The COUNT function returns the number of rows in a query.

The syntax for the COUNT function is:

SELECT COUNT(expression)

FROM tables

WHERE predicates;

**Note:**The COUNT function will only count those records in which the field in the brackets is NOT NULL.

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

|  |  |  |
| --- | --- | --- |
| **Supplier\_ID** | **Supplier\_Name** | **State** |
| 1 | IBM | CA |
| 2 | Microsoft |  |
| 3 | NVIDIA |  |

The result for this query will return 3.

Select COUNT(Supplier\_ID) from suppliers;

While the result for the next query will only return 1, since there is only one row in the suppliers table where the State field is NOT NULL.

Select COUNT(State) from suppliers;

For example, you might wish to know how many employees have a salary that is above $25,000 / year.

SELECT COUNT(\*) as "Number of employees"

FROM employees

WHERE salary > 25000;

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

**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 $25,000 / year.

SELECT COUNT(DISTINCT department) as "Unique departments"

FROM employees

WHERE salary > 25000;

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

**Example using GROUP BY**

In some cases, you will be required to use a GROUP BY clause with the COUNT function. For example, you could use the COUNT function to return the name of the department and the number of employees (in the associated department) that make over $25,000 / year.

SELECT department, COUNT(\*) as "Number of employees"

FROM employees

WHERE salary > 25000

GROUP BY department;

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.

TIP: Performance Tuning !!

Since the COUNT function will return the same results regardless of what NOT NULL field(s) you include as the COUNT function parameters (ie: within the brackets), you can change the syntax of the COUNT function to COUNT(1) to get better performance as the database engine will not have to fetch back the data fields.

For example, based on the example above, the following syntax would result in better performance:

SELECT department, COUNT(1) as "Number of employees"

FROM employees

WHERE salary > 25000

GROUP BY department;

Now, the COUNT function does not need to retrieve all fields from the employees table as it had to when you used the COUNT(\*) syntax. It will merely retrieve the numeric value of 1 for each record that meets your criteria.

**Practice Exercise #1:**

Based on the employees table populated with the following data, count the number of employees whose salary is over $55,000 per year.

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:

Although inefficient in terms of performance, the following SQL statement would return the number of employees whose salary is over $55,000 per year.

SELECT COUNT(\*) as "Number of employees"

FROM employees

WHERE salary > 55000;

It would return the following result set:

|  |
| --- |
| **Number of employees** |
| 3 |

A more efficient implementation of the same solution would be the following SQL statement:

SELECT COUNT(1) as "Number of employees"

FROM employees

WHERE salary > 55000;

Now, the COUNT function does not need to retrieve all of the fields from the table (ie: employee\_number, employee\_name, and salary), but rather whenever the condition is met, it will retrieve the numeric value of 1. Thus, increasing the performance of the SQL statement.

**Practice Exercise #2:**

Based on the suppliers table populated with the following data, count the number of distinct cities in the suppliers 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 (5004, 'NVIDIA', 'New York');

INSERT INTO suppliers (supplier\_id, supplier\_name, city)

VALUES (5005, 'NVIDIA', 'LA');

Solution:

The following SQL statement would return the number of distinct cities in the suppliers table:

SELECT COUNT(DISTINCT city) as "Distinct Cities"

FROM suppliers;

It would return the following result set:

|  |
| --- |
| **Distinct Cities** |
| 4 |

**Practice Exercise #3:**

Based on the customers table populated with the following data, count the number of distinct cities for each customer\_name in the 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)

);

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7001, 'Microsoft', 'New York');

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7002, 'IBM', 'Chicago');

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7003, 'Red Hat', 'Detroit');

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7004, 'Red Hat', 'New York');

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7005, 'Red Hat', 'San Francisco');

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7006, 'NVIDIA', 'New York');

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7007, 'NVIDIA', 'LA');

INSERT INTO customers (customer\_id, customer\_name, city)

VALUES (7008, 'NVIDIA', 'LA');

Solution:

The following SQL statement would return the number of distinct cities for each customer\_name in the customers table:

SELECT customer\_name, COUNT(DISTINCT city) as "Distinct Cities"

FROM customers

GROUP BY customer\_name;

It would return the following result set:

|  |  |
| --- | --- |
| **CUSTOMER\_NAME** | **Distinct Cities** |
| IBM | 1 |
| Microsoft | 1 |
| NVIDIA | 2 |
| Red Hat | 3 |

**SQL: Combining the "AND" and "OR" Conditions**

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

**Example #1**

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

SELECT \*

FROM suppliers

WHERE (city = 'New York' and name = 'IBM')

 or (city = 'Newark');

This would return all suppliers that reside in New York whose name is IBM and all suppliers that reside in Newark. The brackets determine what order the AND and OR conditions are evaluated in.

**Example #2**

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

For example:

SELECT supplier\_id

FROM suppliers

WHERE (name = 'IBM')

 or (name = 'Hewlett Packard' and city = 'Atlantic City')

 or (name = 'Gateway' and status = 'Active' and city = 'Burma');

This SQL statement would return all supplier\_id values where the supplier's name is IBM or the name is Hewlett Packard and the city is Atlantic City or the name is Gateway, the status is Active, and the city is Burma.

**SQL: BETWEEN Condition**

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

SELECT columns

FROM tables

WHERE column1 between value1 and value2;

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

**Example #1 - Numbers**

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

SELECT \*

FROM suppliers

WHERE supplier\_id between 5000 AND 5010;

This would return all rows where the supplier\_id is between 5000 and 5010, inclusive. It is equivalent to the following SQL statement:

SELECT \*

FROM suppliers

WHERE supplier\_id >= 5000

AND supplier\_id <= 5010;

**Example #2 - Dates**

You can also use the BETWEEN function with dates.

SELECT \*

FROM orders

WHERE order\_date between to\_date ('2003/01/01', 'yyyy/mm/dd')

 AND to\_date ('2003/12/31', 'yyyy/mm/dd');

This SQL statement would return all orders where the order\_date is between Jan 1, 2003 and Dec 31, 2003 (inclusive).

It would be equivalent to the following SQL statement:

SELECT \*

FROM orders

WHERE order\_date >= to\_date('2003/01/01', 'yyyy/mm/dd')

 AND order\_date <= to\_date('2003/12/31','yyyy/mm/dd');

**Example #3 - NOT BETWEEN**

The BETWEEN function can also be combined with the NOT operator. For example,

SELECT \*

FROM suppliers

WHERE supplier\_id not between 5000 and 5500;

This would be equivalent to the following SQL:

SELECT \*

FROM suppliers

WHERE supplier\_id < 5000

OR supplier\_id > 5500;

In this example, the result set would exclude all supplier\_id values between the range of 5000 and 5500 (inclusive).

**SQL: "AND" Condition**

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

SELECT columns

FROM tables

WHERE column1 = 'value1'

 and column2 = 'value2';

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

**Example #1**

The first example that we'll take a look at involves a very simple example using the AND condition.

SELECT \*

FROM suppliers

WHERE city = 'New York'

 and type = 'PC Manufacturer';

This would return all suppliers that reside in New York and are PC Manufacturers. Because the \* is used in the select, all fields from the supplier table would appear in the result set.

**Example #1**

Our next example demonstrates how the AND condition can be used to "join" multiple tables in an SQL statement.

SELECT orders.order\_id, suppliers.supplier\_name

FROM suppliers, orders

WHERE suppliers.supplier\_id = orders.supplier\_id

 and suppliers.supplier\_name = 'IBM';

This would return all rows where the supplier\_name is IBM. And the suppliers 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 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.).

**SQL: WHERE Clause**

The WHERE clause allows you to filter the results from an SQL statement - select, insert, update, or delete statement. It is difficult to explain the basic syntax for the WHERE clause, so instead, we'll take a look at some examples.

**Example #1**

SELECT \*

FROM suppliers

WHERE supplier\_name = 'IBM';

In this first example, we've used the WHERE clause to filter our results from the suppliers table. The SQL statement above would return all rows from the suppliers table where the supplier\_name is IBM. Because the \* is used in the select, all fields from the suppliers table would appear in the result set.

**Example #2**

SELECT supplier\_id

FROM suppliers

WHERE supplier\_name = 'IBM'

 or supplier\_city = 'Newark';

We can define a WHERE clause with multiple conditions. This SQL statement would return all supplier\_id values where the supplier\_name is IBM or the supplier\_city is Newark.

**Example #3**

SELECT suppliers.suppler\_name, orders.order\_id

FROM suppliers, orders

WHERE suppliers.supplier\_id = orders.supplier\_id

and suppliers.supplier\_city = 'Atlantic City';

We can also use the WHERE clause to join multiple tables together in a single SQL statement. This SQL statement would return all supplier names and order\_ids where there is a matching record in the suppliers and orders tables based on supplier\_id, and where the supplier\_city is Atlantic City.

**SQL: ORDER BY Clause**

The ORDER BY clause allows you to sort the records in your result set. The ORDER BY clause can only be used in SELECT statements.

The syntax for the ORDER BY clause is:

SELECT columns

FROM tables

WHERE predicates

ORDER BY column ASC/DESC;

The 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.

**Example #1**

SELECT supplier\_city

FROM suppliers

WHERE supplier\_name = 'IBM'

ORDER BY supplier\_city;

This would return all records sorted by the supplier\_city field in ascending order.

**Example #2**

SELECT supplier\_city

FROM suppliers

WHERE supplier\_name = 'IBM'

ORDER BY supplier\_city DESC;

This would return all records sorted by the supplier\_city field in descending order.

**Example #3**

You can also 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.

SELECT supplier\_city

FROM suppliers

WHERE supplier\_name = 'IBM'

ORDER BY 1 DESC;

This 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.

**Example #4**

SELECT supplier\_city, supplier\_state

FROM suppliers

WHERE supplier\_name = 'IBM'

ORDER BY supplier\_city DESC, supplier\_state ASC;

This would return all records sorted by the supplier\_city field in descending order, with a secondary sort by supplier\_state in ascending order.

**SQL: "OR" Condition**

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

The syntax for the OR condition is:

SELECT columns

FROM tables

WHERE column1 = 'value1'

 or column2 = 'value2';

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

**Example #1**

The first example that we'll take a look at involves a very simple example using the OR condition.

SELECT \*

FROM suppliers

WHERE city = 'New York'

 or city = 'Newark';

This would return all suppliers that reside in either New York or Newark. Because the \* is used in the select, all fields from the suppliers table would appear in the result set.

**Example #2**

The next example takes a look at three conditions. If any of these conditions is met, the record will be included in the result set.

SELECT supplier\_id

FROM suppliers

WHERE name = 'IBM'

 or name = 'Hewlett Packard'

 or name = 'Gateway';

This SQL statement would return all supplier\_id values where the supplier's name is either IBM, Hewlett Packard or Gateway.

**SQL: MIN Function**

The MIN function returns the minimum value of an expression. The syntax for the MIN function is:

SELECT MIN(expression )

FROM tables

WHERE predicates;

**Simple Example**

For example, you might wish to know the minimum salary of all employees.

SELECT MIN(salary) as "Lowest salary"

FROM employees;

In this example, we've aliased the min(salary) field as "Lowest salary". As a result, "Lowest salary" will display as the field name when the result set is returned.

**Example using GROUP BY**

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

For example, you could also use 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;

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

**SQL: MAX Function**

The MAX function returns the maximum value of an expression. The syntax for the MAX function is:

SELECT MAX(expression )

FROM tables

WHERE predicates;

**Simple Example**

For example, you might wish to know the maximum salary of all employees.

SELECT MAX(salary) as "Highest salary"

FROM employees;

In this example, we've aliased the max(salary) field as "Highest salary". As a result, "Highest salary" will display as the field name when the result set is returned.

**Example using GROUP BY**

In some cases, you will be required to use a GROUP BY 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.

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**

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, 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 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 in an 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 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 should work:

SELECT Name, Score

FROM Scoring

WHERE Score = (select Max(Score) from Scoring);

Question:

I need help in an 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 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 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

**SQL: LIKE Condition**

The LIKE condition allows you to use wildcards in the where clause of an SQL statement. This allows you to perform pattern matching. The LIKE condition can be used in any valid SQL statement - select, insert, update, or delete. The patterns that you can choose from are: % allows you to match any string of any length (including zero length) \_ allows you to match on a single character

**Examples using % wildcard**

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

SELECT \* FROM suppliers

WHERE supplier\_name like 'Hew%';

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

SELECT \* FROM suppliers

WHERE supplier\_name like '%bob%';

In this example, we are looking for all suppliers whose name contains the characters 'bob'. You could also use the LIKE condition to find suppliers whose name does not start with 'T'. For example,

SELECT \* FROM suppliers

WHERE supplier\_name not like 'T%';

By placing the not keyword in front of the LIKE condition, you are able to retrieve all suppliers whose name does not start with 'T'. Examples using \_ wildcard Next, let's explain how the \_ wildcard works. Remember that the \_ is looking for only one character. For example,

SELECT \* FROM suppliers

WHERE supplier\_name like 'Sm\_th';

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

SELECT \* FROM suppliers

WHERE account\_number like '12317\_';

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:

123170

123171

123172

123173

123174

123175

123176

123177

123178

123179

Examples using Escape Characters Next, in Oracle, let's say you wanted to search for a % or a \_ character in a LIKE condition. You can do this using an Escape character. Please note that you can define an escape character as a single character (length of 1) ONLY. For example,

SELECT \* FROM suppliers

WHERE supplier\_name LIKE '!%' escape '!';

This SQL statement identifies the ! character as an escape character. This statement will return all suppliers whose name is %. Here is another more complicated example:

SELECT \* FROM suppliers

WHERE supplier\_name LIKE 'H%!%' escape '!';

This 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. For example,

SELECT \* FROM suppliers

WHERE supplier\_name LIKE 'H%!\_' escape '!';

This 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**

Question:

How do you incorporate the Oracle upper function with the 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 take a 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 SQL statements:

select \* from suppliers

where upper(supplier\_name) like ('TEST%');

or

select \* from suppliers

where upper(supplier\_name) like upper('test%')

These SQL statements use a combination of the 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:**

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: The following SQL statement would 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:**

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: The following SQL statement would 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:**

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 ('5001', 'Cisco', 'Jersey City');

INSERT INTO suppliers (supplier\_id, supplier\_name, city)

VALUES ('5007', 'EMC', 'Boston');

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: The following SQL statement would 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** |
| 5001 | Cisco | Jersey City |
| 5007 | EMC | Boston |
| 5008 | Microsoft | New York |
| 5009 | IBM | Chicago |

**SQL: INSERT Statement**

The INSERT statement allows you to insert a single record or multiple records into a table. The syntax for the INSERT statement is:

INSERT INTO table

 (column-1, column-2, ... column-n)

VALUES

 (value-1, value-2, ... value-n);

**Example #1 - Simple example**

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

INSERT INTO suppliers (supplier\_id, supplier\_name)

VALUES (24553, 'IBM');

This 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.

**Example #2 - More complex example**

You can also perform more complicated inserts 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 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 statement before performing the insert.

SELECT count(\*)

FROM customers

WHERE city = 'Newark';

**Frequently Asked Questions**

Question:

I am setting up a database with clients. I know that you use the "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 EXISTS condition. For example, if you had a table named clients with a primary key of client\_id, you could use the following 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 statement inserts multiple records with a subselect. If you wanted to insert a single record, you could use the following 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.

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;

**SQL Basics**

* SQL Stands for Structured Query Language
* Common Language For Variety of Databases
* SQL lets you access and manipulate databases

Types of SQL / Classification of SQL Clauses:

DML - Data Manipulation Language (SELECT)

It is used to retrieve, store, delete, insert and update data in/from the database.

*DML Commands*:

|  |  |
| --- | --- |
| [**SELECT**](http://www.thedataanalysis.com/sql/sql-programming/50-sql-select-statement-how-to-use-sql-select-statement-.html)- extracts data from the database[**INSERT**](http://www.thedataanalysis.com/sql/sql-programming/51-sql-insert-statement-how-to-use-sql-insert-clause-.html) - inserts new data into the database[**UPDATE**](http://www.thedataanalysis.com/sql/sql-programming/68-sql-update-statement-how-to-use-sql-update-statement.html) - updates data in the database[**DELETE**](http://www.thedataanalysis.com/sql/sql-programming/49-sql-delete-statement-how-to-use-sql-delete-statement-.html) - deletes data from the database | **MERGE****CALL****EXPLAIN PLAN****LOCK TABLE** |

DDL - Data Definition Language (CREATE TABLE)

It is used to create and modify the structure of database objects in the database.

*DDL Commands*:

|  |  |
| --- | --- |
| **CREATE** – creates a new database table or index.**ALTER** - alters or changes the database table.**DROP** - deletes the database table | **TRUNCATE****COMMENT****RENAME** |

DCL - Data Control Language

It is used to create roles, permissions, control access and referential integrity to the database.

*DCL Commands*:

**GRANT** - gives access privileges to users of the database.

**REVOKE** – withdraws access privileges to users of the database.

TCL - Transactional Control Language

It is used to manage different transactions occurring within a database.

*TCL Commands*:

|  |  |
| --- | --- |
| **COMMIT** - Saves the work done so that the changes or additions are reflected in future. | **SAVE POINT** – allows us to break the sql code in to blocks and then name that blocks as save point. |
| **ROLLBACK** - restores the database values to original state since the last commit. | **SET TRANSACTION** |

Pros and Cons of SQL:

|  |  |
| --- | --- |
| Pros: | Cons: |
| Very flexibleUniversal (Oracle, Access, Paradox, Microsoft, etc)Relatively Few Commands to Learn | Requires Detailed Knowledge of the Structure of the DatabaseCan Provide Misleading Results |