http://www.techonthenet.com/sql/

SQL stands for "*Structured Query Language*".
It is used by relational database technologies such as Oracle, Microsoft Access, and Sybase, among others. We've categorized SQL into the following topics:

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)
integer	integer	
smallint	smallint	
numeric	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.
decimal	decimal(p,s)	Where ***p*** is a precision value; ***s*** is a scale value.
real	real	Single-precision floating point number
double precision	double precision	Double-precision floating point number
float	float(p)	Where ***p*** is a precision value.
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.

bit varying	bit varying(x)	Where ***x*** is the number of bits to store. The length can vary up to x.
date	date	Stores year, month, and day values.
time	time	Stores the hour, minute, and second values.
timestamp	timestamp	Stores year, month, day, hour, minute, and second values.
time with time zone	time with time zone	Exactly the same as time, but also stores an offset from UTC of the time specified.
timestamp with time zone	timestamp with time zone	Exactly the same as timestamp, but also stores an offset from UTC of the time specified.
year-month interval		Contains a year value, a month value, or both.
day-time interval		Contains a day value, an hour value, a minute value, and/or a second value.

SQL: SELECT Statement

The SELECT statement allows you to retrieve records from one or more tables in your database.
SYNTAX:
SELECT columns
FROM tables
WHERE predicates;

Example #1
Let's take a look at how to select all fields from a table.

```
SELECT *
FROM suppliers
WHERE city = 'Newark';
```

In our example, we've used * to signify that we wish to view all fields from the suppliers table where the supplier resides in Newark.

Example #2
You can also choose to select individual fields as opposed to all fields in the table. For example:

```
SELECT name, city, state
FROM suppliers
```

```
WHERE supplier_id > 1000;
```

This select statement would return all name, city, and state values from the suppliers table where the supplier_id value is greater than 1000.

Example #3
You can also use the select statement to retrieve fields from multiple tables.

```
SELECT orders.order_id, suppliers.name
FROM suppliers, orders
WHERE suppliers.supplier_id = orders.supplier_id;
```

The result set would display the order_id and suppier name fields where the supplier_id value existed in both the suppliers and orders table.

SQL: DISTINCT Clause

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

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

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

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

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

Example #2
The DISTINCT clause can be used with more than one field.
For example:

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

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

SQL: 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;

Simple Example
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: 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: 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.

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: 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: "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 #2

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 *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 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: "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: 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: 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 ('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
5008	Microsoft	New York
5009		

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: 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: 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: GROUP BY Clause

The 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.

The syntax for the GROUP BY clause is:

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

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

Example using the SUM function

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

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

```
GROUP BY department;
```

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

Example using the COUNT function

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

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

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.

```
SELECT department, MIN(salary) as "Lowest salary"
FROM employees
GROUP BY department;
```

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.

```
SELECT department, MAX(salary) as "Highest salary"
FROM employees
GROUP BY department;
```

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, ... column_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, MIN, or MAX.

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: 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: Joins

A **join** is used to combine rows from multiple tables. A join is performed whenever two or more tables is listed in the FROM clause of an SQL statement. There are different kinds of joins. Let's take a look at a few examples.

Inner Join (simple join)

Chances are, you've already written an SQL statement that uses an inner join. It is the most common type of join. Inner joins return all rows from multiple tables where the join condition is met.
For example,

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

This SQL statement 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 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

If we run the SQL statement below:

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

Outer Join
Another type of join is called an outer join. This type of join returns all rows from one table and **only** those rows from a secondary table where the joined fields are equal (join condition is met).
For example,

```
select suppliers.supplier_id, suppliers.supplier_name, orders.order_date
```

```
from suppliers, orders
where suppliers.supplier_id = orders.supplier_id(+);
```

This SQL statement would return all rows from the suppliers table and only those rows from the orders table where the joined fields are equal.
The (+) after the orders.supplier_id field indicates that, 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.

The above SQL statement could also be written as follows:

```
select suppliers.supplier_id, suppliers.supplier_name, orders.order_date
from suppliers, orders
where orders.supplier_id(+) = suppliers.supplier_id
```

Let's look at some data to explain how 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 SQL statement below:

```
select suppliers.supplier_id, suppliers.supplier_name, orders.order_date
from suppliers, orders
where 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 an outer join was used. However, you will notice that the order_date field for those records contains a <null> value.

SQL: UNION Query

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

```
select field1, field2, . field_n
from tables
UNION
select field1, field2, . field_n
from tables;
```

Example #1

The following is an example of a UNION query:

```
select supplier_id
from suppliers
UNION
select supplier_id
from orders;
```

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

Example #2 - With ORDER BY Clause

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

```
select supplier_id, supplier_name
from suppliers
where supplier_id > 2000
```

```
UNION
select company_id, company_name
from companies
where company_id > 1000
ORDER BY 2;
```

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

Frequently Asked Questions

Question: I need to compare two dates and return the *count* of a field based on the date values. For example, I have a date field in a table called last updated date. I have to check if trunc(last_updated_date >= trun(sysdate-13).

Answer: Since you are using the COUNT function which is an aggregate function, we'd recommend using a UNION query. For example, you could try the following:

```
SELECT a.code as Code, a.name as Name, count(b.Ncode)
    FROM cdmaster a, nmmaster b
    WHERE a.code = b.code
    and a.status = 1
    and b.status = 1
    and b.Ncode <> 'a10'
    and trunc(last_updated_date) <= trunc(sysdate-13)
    group by a.code, a.name
UNION
SELECT a.code as Code, a.name as Name, count(b.Ncode)
    FROM cdmaster a, nmmaster b
    WHERE a.code = b.code
    and a.status = 1
    and b.status = 1
    and b.Ncode <> 'a10'
    and trunc(last_updated_date) > trunc(sysdate-13)
    group by a.code, a.name;
```

The UNION query allows you to perform a COUNT based on one set of criteria.

trunc(last_updated_date) <= trunc(sysdate-13)
As well as perform a COUNT based on another set of criteria.
trunc(last_updated_date) > trunc(sysdate-13)

SQL: UNION ALL Query

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

The syntax for a UNION ALL query is:

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

Example #1

The following is an example of a UNION ALL query:

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

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

Example #2 - With ORDER BY Clause

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

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

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

SQL: INTERSECT Query

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

The syntax for an INTERSECT query is:

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

Example #1

The following is an example of an INTERSECT query:

```
select supplier_id
from suppliers
INTERSECT
select supplier_id
from orders;
```

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

Example #2 - With ORDER BY Clause

The following is an INTERSECT query that uses an ORDER BY clause:

```
select supplier_id, supplier_name
from suppliers
where supplier_id > 2000
```

```
INTERSECT
select company_id, company_name
from companies
where company_id > 1000
ORDER BY 2;
```

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

SQL: MINUS Query

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

```
select field1, field2, . field_n
from tables
MINUS
select field1, field2, . field_n
from tables;
```

Example #1

The following is an example of an MINUS query:

```
select supplier_id
from suppliers
MINUS
select supplier_id
from orders;
```

In this example, the SQL would return all supplier_id values that are in the suppliers table and not in the orders table. What this means is that if a supplier_id value existed in the suppliers table and also existed in the orders table, the supplier_id value would not appear in this result set.

Example #2 - With ORDER BY Clause

The following is an MINUS query that uses an ORDER BY clause:

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

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

SQL: UPDATE Statement

The UPDATE statement allows you to update a single record or multiple records in a table.
The syntax for the UPDATE statement is:

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

Example #1 - Simple example

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

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

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

Example #2 - More complex example

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

For example:

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

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

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".

```
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:

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		

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: DELETE Statement

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

The syntax for the DELETE statement is:

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

Example #1 - Simple example

Let's take a look at a simple example:

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

This would delete all records from the suppliers table where the supplier_name is IBM.

You may wish to check for the number of rows that will be deleted. You can determine the number of rows that will be deleted by running the following SQL statement **before** performing the delete.

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

Example #2 - More complex example

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

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

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

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

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

Frequently Asked Questions

Question: How would I write an SQL 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:

```
DELETE FROM TableA
WHERE NOT EXISTS
    ( select *
    from TableB
    where TableA .field1 = TableB.fieldx
    and TableA .field2 = TableB.fieldz );
```

SQL: CREATE TABLE Statement

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

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

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.

For example:

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

Practice Exercise #1:
Create a *customers* table that stores customer ID, name, and address information. The customer ID should be the primary key for the table.

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

```
CREATE TABLE customers
(    customer_id     number(10)     not null,
     customer_name     varchar2(50)    not null,
     addressvarchar2(50),
     city varchar2(50),
     state    varchar2(25),
     zip_code    varchar2(10),
     CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Practice Exercise #2:
Based on the *departments* table below, create an *employees* table 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:

The 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: CREATE a table from another table

You can also 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 SELECT Statement).

Syntax #1 - Copying all columns from another table

The basic syntax 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.

Syntax #2 - Copying selected columns from another table
The basic syntax is:

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

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.

Syntax #3 - Copying selected columns from multiple tables
The basic syntax is:

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

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.

Acknowledgements: We'd like to thank Dave M. for contributing to this solution!

Frequently Asked Questions

Question: How can I create a table from another table without copying any values from the old table?

Answer: To do this, the basic 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!

SQL: ALTER TABLE Statement

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

Renaming a table

The basic syntax for renaming a table is:

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

For example:

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

This will rename the *suppliers* table to *vendors*.

Adding column(s) to a table

Syntax #1

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

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

For example:

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

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

Syntax #2

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

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

For example:

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

This will add two columns (*supplier_name* and *city*) to the *supplier* table.

Modifying column(s) in a table

Syntax #1

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

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

For example:

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

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

Syntax #2

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

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

For example:

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

This will modify both the *supplier_name* and *city* columns.

Drop column(s) in a table

Syntax #1

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

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

For example:

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

This will drop the column called *supplier_name* from the table called *supplier*.

Rename column(s) in a table

(NEW in Oracle 9i Release 2)

Syntax #1

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

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

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

For example:

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

This will rename the column called *supplier_name* to *sname*.

Acknowledgements: Thanks to Dave M., Craig A., and Susan W. for contributing to this solution!

Practice Exercise #1:

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

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

Practice Exercise #2:
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:
The following ALTER TABLE statement would add a *salary* column to the *employees* table:

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

Practice Exercise #3:
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,
    addressvarchar2(50),
    city varchar2(50),
    state    varchar2(25),
    zip_code    varchar2(10),
    CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Solution:
The following 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:
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:
The following 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:
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),
    addressvarchar2(50),
    city varchar2(50),
    state    varchar2(25),
    zip_code    varchar2(10),
    CONSTRAINT customers_pk PRIMARY KEY (customer_id)
);
```

Solution:
The following 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:

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:

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

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

Practice Exercise #7:

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:

The following 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: DROP TABLE Statement

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

The basic syntax for the DROP TABLE statement is:

```
DROP TABLE table_name;
```

For example:
DROP TABLE supplier;

This would drop table called *supplier*.

SQL: Global Temporary tables

Global temporary tables are distinct within SQL sessions.
The basic syntax is:
CREATE GLOBAL TEMPORARY TABLE table_name (...);

For example:
```
CREATE GLOBAL TEMPORARY TABLE supplier
(    supplier_id numeric(10)     not null,
     supplier_name  varchar2(50)    not null,
     contact_name   varchar2(50)
)
```
This would create a global temporary table called *supplier* .

SQL: Local Temporary tables

Local temporary tables are distinct within modules and embedded SQL programs within SQL sessions.
The basic syntax is:
DECLARE LOCAL TEMPORARY TABLE table_name (...);

SQL: VIEWS

A view is, in essence, a virtual table. It does not physically exist. Rather, it is created by a query joining one or more tables.

Creating a VIEW

The syntax for creating a VIEW is:
```
CREATE VIEW view_name AS
     SELECT columns
     FROM table
     WHERE predicates;
```

For example:
CREATE VIEW sup_orders AS

```
    SELECT suppliers.supplier_id, orders.quantity, orders.price
    FROM suppliers, orders
    WHERE suppliers.supplier_id = orders.supplier_id
    and suppliers.supplier_name = 'IBM';
```

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

```
SELECT *
    FROM sup_orders;
```

Updating a VIEW

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

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

For example:

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

Dropping a VIEW

The syntax for dropping a VIEW is:

```
DROP VIEW view_name;
```

For example:

```
DROP VIEW sup_orders;
```

Frequently Asked Questions

Question: Can you update the data in a view?

Answer: A view is created by joining one or more tables. When you update record(s) in a view, it updates the records in the underlying tables that make up the view.

So, yes, you can update the data in a view providing you have the proper privileges to the underlying tables.

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