Consider,

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sample Table: empinfo** | | | | | |
| **first** | **last** | **id** | **age** | **city** | **state** |
| John | Jones | 99980 | 45 | Payson | Arizona |
| Mary | Jones | 99982 | 25 | Payson | Arizona |
| Eric | Edwards | 88232 | 32 | San Diego | California |
| Mary Ann | Edwards | 88233 | 32 | Phoenix | Arizona |
| Ginger | Howell | 98002 | 42 | Cottonwood | Arizona |
| Sebastian | Smith | 92001 | 23 | Gila Bend | Arizona |
| Gus | Gray | 22322 | 35 | Bagdad | Arizona |
| Mary Ann | May | 32326 | 52 | Tucson | Arizona |
| Erica | Williams | 32327 | 60 | Show Low | Arizona |
| Leroy | Brown | 32380 | 22 | Pinetop | Arizona |
| Elroy | Cleaver | 32382 | 22 | Globe | Arizona |

Display everyone's first name and their age for everyone that's in table.

select first,

age

from empinfo;

Display the first name, last name, and city for everyone that's not from Payson.

select first,

last,

city

from empinfo

where city <>

'Payson';

Display all columns for everyone that is over 40 years old.

select \* from empinfo

where age > 40;

Display the first and last names for everyone whose last name ends in an "ay".

select first, last from empinfo

where last LIKE '%ay';

Display all columns for everyone whose first name equals "Mary".

select \* from empinfo

where first = 'Mary';

Display all columns for everyone whose first name contains "Mary".

select \* from empinfo

# Creating Tables

The **create table** statement is used to create a new table. Here is the format of a simple **create table** statement:

create table "tablename"

("column1" "data type",

"column2" "data type",

"column3" "data type");

Format of create table if you were to use optional constraints:

create table "tablename"

("column1" "data type"

[constraint],

"column2" "data type"

[constraint],

"column3" "data type"

[constraint]);

[ ] = optional

**Note:** You may have as many columns as you'd like, and the constraints are optional.

**Example:**

create table employee

(first varchar(15),

last varchar(20),

age number(3),

address varchar(30),

city varchar(20),

state varchar(20));

To create a new table, enter the keywords **create table** followed by the table name, followed by an open parenthesis, followed by the first column name, followed by the data type for that column, followed by any optional constraints, and followed by a closing parenthesis. It is important to make sure you use an open parenthesis before the beginning table, and a closing parenthesis after the end of the last column definition. Make sure you seperate each column definition with a comma. All SQL statements should end with a ";".

The table and column names must start with a letter and can be followed by letters, numbers, or underscores - not to exceed a total of 30 characters in length. Do not use any SQL reserved keywords as names for tables or column names (such as "select", "create", "insert", etc).

Data types specify what the type of data can be for that particular column. If a column called "Last\_Name", is to be used to hold names, then that particular column should have a "varchar" (variable-length character) data type.

Here are the most common Data types:

|  |  |
| --- | --- |
| char(size) | Fixed-length character string. Size is specified in parenthesis. Max 255 bytes. |
| varchar(size) | Variable-length character string. Max size is specified in parenthesis. |
| number(size) | Number value with a max number of column digits specified in parenthesis. |
| date | Date value |
| number(size,d) | Number value with a maximum number of digits of "size" total, with a maximum number of "d" digits to the right of the decimal. |

What are constraints? When tables are created, it is common for one or more columns to have **constraints** associated with them. A constraint is basically a rule associated with a column that the data entered into that column must follow. For example, a "unique" constraint specifies that no two records can have the same value in a particular column. They must all be unique. The other two most popular constraints are "not null" which specifies that a column can't be left blank, and "primary key". A "primary key" constraint defines a unique identification of each record (or row) in a table. All of these and more will be covered in the future Advanced release of this Tutorial. Constraints can be entered in this SQL interpreter, however, they are not supported in this Intro to SQL tutorial & interpreter. They will be covered and supported in the future release of the Advanced SQL tutorial - that is, if "response" is good.

It's now time for you to design and create your own table. You will use this table throughout the rest of the tutorial. If you decide to change or redesign the table, you can either **drop** it and recreate it or you can create a completely different one. The SQL statement **drop** will be covered later.

### Create Table Exercise

You have just started a new company. It is time to hire some employees. You will need to create a table that will contain the following information about your new employees: firstname, lastname, title, age, and salary. After you create the table, you should receive a small form on the screen with the appropriate column names. If you are missing any columns, you need to double check your SQL statement and recreate the table. Once it's created successfully, go to the "Insert" lesson.

**IMPORTANT**: When selecting a table name, it is important to select a unique name that no one else will use or guess. Your table names should have an underscore followed by your initials and the digits of your birth day and month. For example, Tom Smith, who was born on November 2nd, would name his table myemployees\_ts0211 Use this convention for all of the tables you create. Your tables will remain on a shared database until you drop them, or they will be cleaned up if they aren't accessed in 4-5 days. If "support" is good, I hope to eventually extend this to at least one week. When you are finished with your table, it is important to drop your table (covered in last lesson).

# Creating Tables Answer

Your create statement should resemble:

create table

myemployees\_ts0211

(firstname varchar(30),

lastname varchar(30),

title varchar(30),

age number(2),

salary number(8,2));

# Inserting into a Table

The **insert** statement is used to insert or add a row of data into the table.

To insert records into a table, enter the key words **insert into** followed by the table name, followed by an open parenthesis, followed by a list of column names separated by commas, followed by a closing parenthesis, followed by the keyword **values**, followed by the list of values enclosed in parenthesis. The values that you enter will be held in the rows and they will match up with the column names that you specify. Strings should be enclosed in single quotes, and numbers should not.

insert into "tablename"  
 (first\_column,...last\_column)  
 values (first\_value,...last\_value);

In the example below, the column name first will match up with the value 'Luke', and the column name state will match up with the value 'Georgia'.

**Example:**

insert into employee  
 (first, last, age, address, city, state)  
 values ('Luke', 'Duke', 45, '2130 Boars Nest',   
 'Hazard Co', 'Georgia');

**Note:** All strings should be enclosed between **single** quotes: 'string'

### Insert statement exercises

It is time to insert data into your new employee table.

Your first three employees are the following:

Jonie Weber, Secretary, 28, 19500.00  
Potsy Weber, Programmer, 32, 45300.00  
Dirk Smith, Programmer II, 45, 75020.00

Enter these employees into your table first, and then insert at least 5 more of your own list of employees in the table.

After they're inserted into the table, enter select statements to:

1. Select all columns for everyone in your employee table.
2. Select all columns for everyone with a salary over 30000.
3. Select first and last names for everyone that's under 30 years old.
4. Select first name, last name, and salary for anyone with "Programmer" in their title.
5. Select all columns for everyone whose last name contains "ebe".
6. Select the first name for everyone whose first name equals "Potsy".
7. Select all columns for everyone over 80 years old.
8. Select all columns for everyone whose last name ends in "ith".

Create at least 5 of your own select statements based on specific information that you'd like to retrieve

# Inserting into a Table Answers

Your Insert statements should be similar to: (note: use your own table name that you created)

insert into

myemployees\_ts0211

(firstname, lastname,

title, age, salary)

values ('Jonie', 'Weber',

'Secretary', 28,

19500.00);

Select all columns for everyone in your employee table.

select \* from

myemployees\_ts0211

Select all columns for everyone with a salary over 30000.

select \* from

myemployees\_ts0211

where salary > 30000

Select first and last names for everyone that's under 30 years old.

select firstname, lastname

from myemployees\_ts0211

where age < 30

Select first name, last name, and salary for anyone with "Programmer" in their title.

select firstname, lastname, salary

from myemployees\_ts0211

where title LIKE '%Programmer%'

Select all columns for everyone whose last name contains "ebe".

select \* from

myemployees\_ts0211

where lastname LIKE '%ebe%'

Select the first name for everyone whose first name equals "Potsy".

select firstname from

myemployees\_ts0211

where firstname = 'Potsy'

Select all columns for everyone over 80 years old.

select \* from  
myemployees\_ts0211  
 where age > 80

Select all columns for everyone whose last name ends in "ith".

select \* from

myemployees\_ts0211

where lastname LIKE '%ith'

# Deleting Records

The **delete** statement is used to delete records or rows from the table.

delete from "tablename"  
where "columnname"   
 OPERATOR "value"   
[and|or "column"   
 OPERATOR "value"];  
  
[ ] = optional

**Examples:**

delete from employee;

**Note:** if you leave off the where clause, **all records will be deleted!**

delete from employee  
 where lastname = 'May';  
  
delete from employee  
 where firstname = 'Mike' or firstname = 'Eric';

To delete an entire record/row from a table, enter "delete from" followed by the table name, followed by the where clause which contains the conditions to delete. If you leave off the where clause, all records will be deleted.

### Delete statement exercises

(Use the select statement to verify your deletes):

1. Jonie Weber-Williams just quit, remove her record from the table.
2. It's time for budget cuts. Remove all employees who are making over 70000 dollars.

Create at least two of your own delete statements, and then issue a command to delete all records from the table.

# Deleting Records Answers

Jonie Weber-Williams just quit, remove her record from the table:

delete

from myemployees\_ts0211

where lastname =

'Weber-Williams';

It's time for budget cuts. Remove all employees who are making over 70000 dollars.

delete

from myemployees\_ts0211

where salary >

70000;

# Updating Records

The **update** statement is used to update or change records that match a specified criteria. This is accomplished by carefully constructing a where clause.

update "tablename"  
set "columnname" =   
 "newvalue"  
 [,"nextcolumn" =   
 "newvalue2"...]  
where "columnname"   
 OPERATOR "value"   
 [and|or "column"   
 OPERATOR "value"];  
  
 [] = optional

Examples:

update phone\_book  
 set area\_code = 623  
 where prefix = 979;  
  
update phone\_book  
 set last\_name = 'Smith', prefix=555, suffix=9292  
 where last\_name = 'Jones';  
  
update employee  
 set age = age+1  
 where first\_name='Mary' and last\_name='Williams';

### Update statement exercises

After each update, issue a select statement to verify your changes.

1. Jonie Weber just got married to Bob Williams. She has requested that her last name be updated to Weber-Williams.
2. Dirk Smith's birthday is today, add 1 to his age.
3. All secretaries are now called "Administrative Assistant". Update all titles accordingly.
4. Everyone that's making under 30000 are to receive a 3500 a year raise.
5. Everyone that's making over 33500 are to receive a 4500 a year raise.
6. All "Programmer II" titles are now promoted to "Programmer III".
7. All "Programmer" titles are now promoted to "Programmer II".

# Updating Records Answers

Jonie Weber just got married to Bob Williams. She has requested that her last name be updated to Weber-Williams.

update

myemployees\_ts0211

set lastname=

'Weber-Williams'

where firstname=

'Jonie'

and lastname=

'Weber';

Dirk Smith's birthday is today, add 1 to his age.

update myemployees\_ts0211

set age=age+1

where firstname='Dirk' and lastname='Smith';

All secretaries are now called "Administrative Assistant". Update all titles accordingly.

update myemployees\_ts0211

set title = 'Administrative Assistant'

where title = 'Secretary';

Everyone that's making under 30000 are to receive a 3500 a year raise.

update myemployees\_ts0211

set salary = salary + 3500

where salary < 30000;

Everyone that's making over 33500 are to receive a 4500 a year raise.

update myemployees\_ts0211

set salary = salary + 4500

where salary > 33500;

All "Programmer II" titles are now promoted to "Programmer III".

update myemployees\_ts0211

set title = 'Programmer III'

where title = 'Programmer II'

All "Programmer" titles are now promoted to "Programmer II".

update myemployees\_ts0211

set title = 'Programmer II'

where title = 'Programmer'

# Drop a Table

The **drop table** command is used to delete a table and all rows in the table.

To delete an entire table including all of its rows, issue the **drop table** command followed by the tablename. **drop table** is different from deleting all of the records in the table. Deleting all of the records in the table leaves the table including column and constraint information. Dropping the table removes the table definition as well as all of its rows.

drop table "tablename"

**Example:**

drop table myemployees\_ts0211;

# SELECT Statement

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

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

 Here is the format of the SELECT statement:

 SELECT [ALL | DISTINCT] column1[,column2]

FROM table1[,table2]

[WHERE "conditions"]

[GROUP BY "column-list"]

[HAVING "conditions]

[ORDER BY "column-list" [ASC | DESC] ]

Example:

SELECT name, age, salary

FROM employee

WHERE age > 50;

The above statement will select all of the values in the name, age, and salary columns from the employee table whose age is greater than 50.

**Comparison Operators**

|  |  |
| --- | --- |
| = | Equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| <> or != | Not equal to |
| LIKE | String comparison test |

Example:

SELECT name, title, dept FROM employee WHERE title LIKE 'Pro%';

The above statement will select all of the rows/values in the name, title, and dept columns from the employee table whose title starts with 'Pro'. This may return job titles including Programmer or Pro-wrestler.

**ALL** and **DISTINCT** are keywords used to select either ALL (default) or the "distinct" or unique records in your query results. If you would like to retrieve just the unique records in specified columns, you can use the "DISTINCT" keyword. DISTINCT will discard the duplicate records for the columns you specified after the "SELECT" statement: For example:

SELECT DISTINCT age

FROM employee\_info;

This statement will return all of the unique ages in the employee\_info table.

ALL will display "all" of the specified columns including all of the duplicates. The ALL keyword is the default if nothing is specified.

# Aggregate Functions

|  |  |
| --- | --- |
| MIN | returns the smallest value in a given column |
| MAX | returns the largest value in a given column |
| SUM | returns the sum of the numeric values in a given column |
| AVG | returns the average value of a given column |
| COUNT | returns the total number of values in a given column |
| COUNT(\*) | returns the number of rows in a table |

Aggregate functions are used to compute against a "returned column of numeric data" from your SELECT statement. They basically summarize the results of a particular column of selected data. We are covering these here since they are required by the next topic, "GROUP BY". Although they are required for the "GROUP BY" clause, these functions can be used without the "GROUP BY" clause. For example:

SELECT AVG(salary)  
FROM employee;

This statement will return a single result which contains the average value of everything returned in the salary column from the employee table.

Another example:

SELECT AVG(salary)  
FROM employee;  
WHERE title = 'Programmer';

This statement will return the average salary for all employees whose title is equal to 'Programmer'

Example:

SELECT Count(\*)  
FROM employees;

# GROUP BY clause

The GROUP BY clause will gather all of the rows together that contain data in the specified column(s) and will allow aggregate functions to be performed on the one or more columns. This can best be explained by an example:

**GROUP BY** clause syntax:

SELECT column1,   
SUM(column2)  
 FROM "list-of-tables"  
 GROUP BY "column-list";

Let's say you would like to retrieve a list of the highest paid salaries in each dept:

SELECT max(salary), dept  
 FROM employee   
 GROUP BY dept;

This statement will select the maximum salary for the people in each unique department. Basically, the salary for the person who makes the most in each department will be displayed. Their, salary and their department will be returned.

# HAVING clause

The HAVING clause allows you to specify conditions on the rows for each group - in other words, which rows should be selected will be based on the conditions you specify. The HAVING clause should follow the GROUP BY clause if you are going to use it.

**HAVING** clause syntax:

SELECT column1,   
SUM(column2)  
FROM "list-of-tables"  
GROUP BY "column-list"  
HAVING "condition";

HAVING can best be described by example. Let's say you have an employee table containing the employee's name, department, salary, and age. If you would like to select the average salary for each employee in each department, you could enter:

SELECT dept, avg(salary)  
FROM employee  
GROUP BY dept;

But, let's say that you want to ONLY calculate & display the average if their salary is over 20000:

SELECT dept, avg(salary)  
FROM employee  
GROUP BY dept  
HAVING avg(salary) > 20000;

# ORDER BY clause

ORDER BY is an optional clause which will allow you to display the results of your query in a sorted order (either ascending order or descending order) based on the columns that you specify to order by.

**ORDER BY** clause syntax:

 SELECT column1, SUM(column2) FROM "list-of-tables" ORDER BY "column-list" [ASC | DESC];

[ ] = optional

This statement will select the employee\_id, dept, name, age, and salary from the employee\_info table where the dept equals 'Sales' and will list the results in Ascending (default) order based on their Salary.

ASC = Ascending Order - default  
DESC = Descending Order

For example:

SELECT employee\_id, dept, name, age, salary FROM employee\_info WHERE dept = 'Sales' ORDER BY salary;

If you would like to order based on multiple columns, you must seperate the columns with commas. For example:

SELECT employee\_id, dept, name, age, salary  
FROM employee\_info  
WHERE dept = 'Sales'  
ORDER BY salary, age DESC;

# Combining Conditions & Boolean Operators

The AND operator can be used to join two or more conditions in the WHERE clause. Both sides of the AND condition must be true in order for the condition to be met and for those rows to be displayed.

SELECT column1,   
SUM(column2)  
FROM "list-of-tables"  
WHERE "condition1" AND   
"condition2";

The OR operator can be used to join two or more conditions in the WHERE clause also. However, **either** side of the OR operator can be true and the condition will be met - hence, the rows will be displayed. With the OR operator, either side can be true or both sides can be true.

For example:

SELECT employeeid, firstname, lastname, title, salary  
FROM employee\_info  
WHERE salary >= 50000.00 AND title = 'Programmer';

This statement will select the employeeid, firstname, lastname, title, and salary from the employee\_info table where the salary is greater than or equal to 50000.00 AND the title is equal to 'Programmer'. Both of these conditions must be true in order for the rows to be returned in the query. If either is false, then it will not be displayed.

Although they are not required, you can use paranthesis around your conditional expressions to make it easier to read:

SELECT employeeid, firstname, lastname, title, salary  
FROM employee\_info  
WHERE (salary >= 50000.00) AND (title = 'Programmer');

Another Example:

SELECT firstname, lastname, title, salary FROM employee\_info WHERE (title = 'Sales') OR (title = 'Programmer');

This statement will select the firstname, lastname, title, and salary from the employee\_info table where the title is either equal to 'Sales' OR the title is equal to 'Programmer'.

# IN & BETWEEN

SELECT col1, SUM(col2)  
FROM "list-of-tables"  
WHERE col3 IN   
 (list-of-values);  
  
SELECT col1, SUM(col2)  
FROM "list-of-tables"  
WHERE col3 BETWEEN value1   
AND value2;

The IN conditional operator is really a set membership test operator. That is, it is used to test whether or not a value (stated before the keyword IN) is "in" the list of values provided after the keyword **IN**.

For example:

SELECT employeeid, lastname, salary  
FROM employee\_info  
WHERE lastname IN ('Hernandez', 'Jones', 'Roberts', 'Ruiz');

 This statement will select the employeeid, lastname, salary from the employee\_info table where the lastname is equal to either: Hernandez, Jones, Roberts, or Ruiz. It will return the rows if it is ANY of these values.

The IN conditional operator can be rewritten by using compound conditions using the equals operator and combining it with OR - with exact same output results:

SELECT employeeid, lastname, salary  
FROM employee\_info  
WHERE lastname = 'Hernandez' OR lastname = 'Jones' OR lastname = 'Roberts'  
OR lastname = 'Ruiz';

 As you can see, the IN operator is much shorter and easier to read when you are testing for more than two or three values.

You can also use **NOT IN** to exclude the rows in your list.

The BETWEEN conditional operator is used to test to see whether or not a value (stated before the keyword BETWEEN) is "between" the two values stated after the keyword BETWEEN.

For example:

 SELECT employeeid, age, lastname, salary  
FROM employee\_info  
 WHERE age BETWEEN 30 AND 40;

 This statement will select the employeeid, age, lastname, and salary from the employee\_info table where the age is between 30 and 40 (including 30 and 40).

This statement can also be rewritten without the BETWEEN operator:

SELECT employeeid, age, lastname, salary  
FROM employee\_info  
WHERE age >= 30 AND age <= 40;

 You can also use **NOT BETWEEN** to exclude the values between your range.

# Mathematical Functions

Standard ANSI SQL-92 supports the following first four basic arithmetic operators:

(+) addition; (-) subtraction; (\*) multiplication; (/) division; (%) modulo.

 The modulo operator determines the integer remainder of the division. This operator is not ANSI SQL supported, however, most databases support it. The following are some more useful mathematical functions to be aware of since you might need them. These functions are not standard in the ANSI SQL-92 specs, therefore they may or may not be available on the specific RDBMS that you are using.

|  |  |
| --- | --- |
| ABS(x) | returns the absolute value of x |
| SIGN(x) | returns the sign of input x as -1, 0, or 1 (negative, zero, or positive respectively) |
| MOD(x,y) | modulo - returns the integer remainder of x divided by y (same as x%y) |
| FLOOR(x) | returns the largest integer value that is less than or equal to x |
| CEILING(x) or CEIL(x) | returns the smallest integer value that is greater than or equal to x |
| POWER(x,y) | returns the value of x raised to the power of y |
| ROUND(x) | returns the value of x rounded to the nearest whole integer |
| ROUND(x,d) | returns the value of x rounded to the number of decimal places specified by the value d |
| SQRT(x) | returns the square-root value of x |

For example:

SELECT round(salary), firstname  
FROM employee\_info

This statement will select the salary rounded to the nearest whole value and the firstname from the employee\_info table.

# Table Joins, a must

All of the queries up until this point have been useful with the exception of one major limitation - that is, you've been selecting from only one table at a time with your SELECT statement. It is time to introduce you to one of the most beneficial features of SQL & relational database systems - the "**Join**". To put it simply, the "Join" makes relational database systems "relational".

Joins allow you to link data from two or more tables together into a single query result--from one single SELECT statement.

A "Join" can be recognized in a SQL SELECT statement if it has more than one table after the FROM keyword.

For example:

SELECT "list-of-columns"  
FROM table1,table2  
WHERE "search-condition(s)"

Joins can be explained easier by demonstrating what would happen if you worked with one table only, and didn't have the ability to use "joins". This single table database is also sometimes referred to as a "flat table". Let's say you have a one-table database that is used to keep track of all of your customers and what they purchase from your store:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **id** | **first** | **last** | **address** | **city** | **state** | **zip** | **date** | **item** | **price** |
|  |  |  |  |  |  |  |  |  |  |

Every time a new row is inserted into the table, all columns will be updated, thus resulting in unnecessary "redundant data". For example, every time Wolfgang Schultz purchases something, the following rows will be inserted into the table:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **id** | **first** | **last** | **address** | **city** | **state** | **zip** | **date** | **item** | **price** |
| 10982 | Wolfgang | Schultz | 300 N. 1st Ave | Yuma | AZ | 85002 | 032299 | snowboard | 45.00 |
| 10982 | Wolfgang | Schultz | 300 N. 1st Ave | Yuma | AZ | 85002 | 082899 | snow shovel | 35.00 |
| 10982 | Wolfgang | Schultz | 300 N. 1st Ave | Yuma | AZ | 85002 | 091199 | gloves | 15.00 |
| 10982 | Wolfgang | Schultz | 300 N. 1st Ave | Yuma | AZ | 85002 | 100999 | lantern | 35.00 |
| 10982 | Wolfgang | Schultz | 300 N. 1st Ave | Yuma | AZ | 85002 | 022900 | tent | 85.00 |

An ideal database would have two tables:

1. One for keeping track of your customers
2. And the other to keep track of what they purchase:

"Customer\_info" table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **customer\_number** | **firstname** | **lastname** | **address** | **city** | **state** | **zip** |
|  |  |  |  |  |  |  |

 "Purchases" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **customer\_number** | **date** | **item** | **price** |

Now, whenever a purchase is made from a repeating customer, the 2nd table, "Purchases" only needs to be updated! We've just eliminated useless redundant data, that is, we've just [**normalized**](http://www.sqlcourse2.com/normalization.html) this database!

Notice how each of the tables have a common "cusomer\_number" column. This column, which contains the unique customer number will be used to **JOIN** the two tables. Using the two new tables, let's say you would like to select the customer's name, and items they've purchased. Here is an example of a join statement to accomplish this:

SELECT customer\_info.firstname, customer\_info.lastname, purchases.item  
FROM customer\_info, purchases  
WHERE customer\_info.customer\_number = purchases.customer\_number;

This particular "Join" is known as an "Inner Join" or "Equijoin". This is the most common type of "Join" that you will see or use.

Notice that each of the columns are always preceded with the table name and a period. This isn't always required; however, it IS good practice so that you won’t confuse which columns go with what tables. It is required if the name column names are the same between the two tables. I recommend preceding all of your columns with the table names when using joins.

**Note: The syntax described above will work with most Database Systems -including the one with this tutorial. However, in the event that this doesn't work with yours, please check your specific database documentation.**

Although the above will probably work, here is the ANSI SQL-92 syntax specification for an Inner Join using the preceding statement above that you might want to try:

SELECT customer\_info.firstname, customer\_info.lastname, purchases.item  
FROM customer\_info INNER JOIN purchases  
ON customer\_info.customer\_number = purchases.customer\_number;

Another example:

SELECT employee\_info.employeeid, employee\_info.lastname, employee\_sales.comission  
FROM employee\_info, employee\_sales  
WHERE employee\_info.employeeid = employee\_sales.employeeid;

This statement will select the employeeid, lastname (from the employee\_info table), and the comission value (from the employee\_sales table) for all of the rows where the employeeid in the employee\_info table matches the employeeid in the employee\_sales table.