CONTENTS :

The WHERE Clause

The WHERE Clause

Overview

"I have a dream that my four little children will one day live in a nation where they will not be judged by the color of their skin, but by the content of their character."
- Martin Luther King, Jr.

The WHERE Clause limits Returning Rows

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT  First_Name, Last_Name, Class_Code,  Grade_Pt
FROM      Student_Table
WHERE First_Name = 'Henry';
```

First_Name	Last_Name	Class_Code	Grade_Pt
Henry	Hanson	FR	2.88

The WHERE Clause filters how many ROWS are coming back on the report. So, not all rows will return, just the rows that qualify. In this example, I am asking for the report to bring back only rows WHERE the first name is Henry.

Using a Column ALIAS throughout the SQL

```
SELECT  First_Name  AS Fname
        ,Last_Name      Lname
        ,Class_Code     "Class Code"
        ,Grade_Pt       AS "AVG"
        ,Student_ID
FROM      Student_Table
WHERE Fname = 'Henry' ;
```

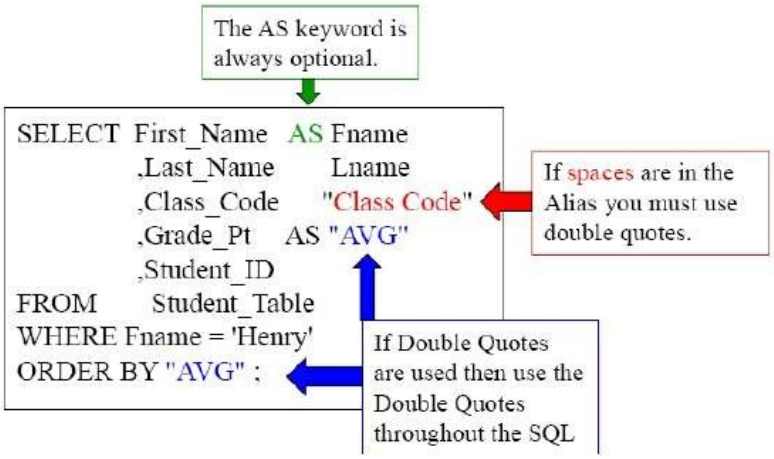
Aliasing a column

Use the ALIAS again in your WHERE Clause!

First_Name	Last_Name	Class_Code	Grade_Pt	Student_ID
Henry	Hanson	FR	2.88	125634

When you ALIAS a column, you give it a new name for the report header. A good rule of thumb is to refer to the column by the alias throughout the query. Notice we use the Alias in the WHERE clause also. When you alias the AS keyword is always optional. Any alias that has spaces in the middle must have double quotes, and an alias such as AVG must have double quotes because it is a reserved word.

Double Quoted Aliases are for Reserved Words and Spaces



When you ALIAS a column, you give it a new name for the report header. A good rule of thumb is to refer to the column by the alias throughout the query. If you use double quotes to define the Alias (because there are spaces or it is a reserved word), then you must be consistent and use the double quotes each time you refer to the alias.

Character Data needs Single Quotes in the WHERE Clause

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT First_Name, Last_Name, Class_Code Grade_Pt
FROM Student_Table
WHERE First_Name = 'Henry';
```

Character Data needs Single Quotes in the WHERE Clause.

In the WHERE clause, if you search for Character data such as first name, you need single quotes around it but remember that you don't single-quote integers.

Character Data needs Single Quotes, but Numbers Don't

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95

333450	Smith	Andy	SO	2.00

```
SELECT *
FROM   Student_Table
WHERE First_Name = 'Henry' ;
```

```
SELECT *
FROM   Student_Table
WHERE Grade_Pt = 0.00 ;
```

Character data (letters) need single quotes, but you need NO Single Quotes for Integers (numbers). Remember, you never use double quotes except for aliasing.

NULL means UNKNOWN DATA so Equal (=) won't Work

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT *
FROM   Student_Table
WHERE Class_Code = NULL ;
```

Error

The first thing you need to know about a NULL is it is unknown data. It is NOT a zero. It is missing data. Since we don't know what is in a NULL, you can't use an = sign. You must use IS NULL or IS NOT NULL.

Use IS NULL or IS NOT NULL when dealing with NULLs

```
SELECT *
FROM   Student_Table
WHERE Class_Code IS NULL ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
260000	Johnson	Stanley	?	?

If you are looking for a row that holds NULL value, you need to put 'IS NULL'. This will only bring back the rows with a NULL value in it.

NULL is UNKNOWN DATA so NOT Equal won't Work

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt

423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT *
FROM    Student_Table
WHERE Class_Code =  NOT NULL ;
```

Error

The same goes with = NOT NULL. We can't compare a NULL with any equal sign. We can only deal with NULL values with IS NULL and IS NOT NULL.

Use IS NULL or IS NOT NULL when dealing with NULLs

```
SELECT  *
FROM    Student_Table
WHERE  Class_Code IS NOT NULL;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

Much like before, when you want to bring back the rows that do not have NULLs in them, you put an 'IS NOT NULL' in the WHERE Clause, and only rows that are not null for the specified column will return.

Using Greater Than OR Equal To (>=)

```
SELECT  * FROM Student_Table
WHERE  Grade_Pt >= 3.0 ;
```

Greater than or Equal to

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95

The WHERE Clause doesn't just deal with 'Equals', but other options too. These include GREATER or LESSER THAN, along with asking for things that are GREATER/LESSER THAN or EQUAL to.

Using GE as Greater Than or Equal To (>=)

```
SELECT  * FROM Student_Table
WHERE  Grade_Pt GE 3.0 ;
```

Greater than or Equal to

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95

The syntax above uses a Teradata extension (GE) for Greater Than or Equal To!

AND in the WHERE Clause

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT    *
FROM      Student_Table
WHERE     Class_Code = 'FR'
AND       First_Name = 'Henry' ;
```

Notice the WHERE statement and the word AND. In this example, qualifying rows must have a Class_Code = 'FR' and must have a First_Name of 'Henry'.

Troubleshooting AND

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT    *
FROM      Student_Table
WHERE Grade_Pt = 3.0  AND Grade_Pt = 4.0;
```

What is going wrong here? You are using an AND to check the same column. What you are basically asking with this syntax, is to see the rows that have BOTH a Grade_Pt of 3.0 and a 4.0. That is impossible, so no rows will be returned.

OR in the WHERE Clause

```
SELECT *
FROM    Student_Table
WHERE  Grade_Pt = 3.0  OR Grade_Pt = 4.0;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
234121	Thomas	Wendy	FR	4.00
123250	Phillips	Martin	SR	3.00

Notice above in the WHERE Clause we use OR. Or allows for either of the parameters to be TRUE in order for the data to qualify and return.

Troubleshooting OR

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT    *
FROM      Student_Table
WHERE Grade_Pt  = 3.0 OR  4.0;
```

Error

A common mistake

This causes an error! Why? You need to state the column name again before the 4.0.

OR must utilize the Column Name Each Time

```
SELECT    *
FROM      Student_Table
WHERE Grade_Pt  = 3.0 OR  Grade_Pt  = 4.0;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt

234121	Thomas	Wendy	FR	4.00
123250	Phillips	Martin	SR	3.00

Notice that you must always state the COLUMN NAME along with the parameter. Even if you are using the same Column Name, you must specify it over again.

Troubleshooting Character Data

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT   *                                  Error
FROM     Student_Table
WHERE Grade_Pt = 3.0 AND Class_Code = SR ;
```

This query errors, but what is WRONG with this syntax? No Single quotes around SR.

Using Different Columns in an AND Statement

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT * FROM Student Table
WHERE Grade_Pt = 3.0 AND Class_Code = 'SR' ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
123250	Phillips	Martin	SR	3.00

Notice that AND separates two different columns, and the data will come back if both are TRUE.

Quiz – How many rows will return?

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90

260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT *
FROM   Student_Table
WHERE Grade_Pt = 4.0 OR Grade_Pt = 3.0
AND    Class_Code = 'SR';
```

Which Seniors have a 3.0 or a 4.0 Grade_Pt average. How many rows will return?

A) 2 C) Error

B) 1 D) 3

Answer to Quiz – How many rows will return?

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT     * FROM Student_Table
WHERE     Grade_Pt = 4.0 OR Grade_Pt = 3.0
AND  Class_Code = 'SR' ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
234121	Thomas	Wendy	FR	4.00
123250	Phillips	Martin	SR	3.00

We had two rows return? Isn't that a mystery! Why?

A) 2 C) Error

B) 1 D) 3

What is the Order of Precedence?


```
SELECT   *
FROM     Student_Table
WHERE  Grade_Pt = 4.0 OR Grade_Pt = 3.0
AND      Class_Code = 'SR';
```

Syntax has an ORDER OF PRECEDENCE. It will read anything with parentheses around it first. Then, it will read the NOT statements. Then, it will read the AND statements. FINALLY, it will read the OR Statements. This is why the last query came out odd. Let's fix it and bring back the right answer set.

Using Parentheses to change the Order of Precedence

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT  * FROM  Student_Table
WHERE  (Grade_Pt = 3.0 OR Grade_Pt = 4.0)
AND     Class_Code = 'SR' ;
```

Parentheses Evaluated First!

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
123250	Phillips	Martin	SR	3.00

This is the proper way of looking for rows that have both a Grade_Pt of 3.0 or 4.0, AND also having a Class_Code of 'SR'. Only ONE row comes back. Parentheses are evaluated first, so this allows you to direct exactly what you want to work first.

Using an IN List in place of OR

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95

333450	Smith	Andy	SO	2.00

```
SELECT * FROM Student_Table
WHERE  Grade_Pt IN (3.0, 4.0)
AND    Class_Code = 'SR' ;
```

The IN List

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
123250	Phillips	Martin	SR	3.00

Using an IN List is a great way of looking for rows that have both a Grade_Pt of 3.0 or 4.0, AND also have a Class_Code of 'SR'. Only ONE row comes back.

The IN List is an Excellent Technique

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT * FROM Student Table
WHERE Grade_Pt IN (2.0, 3.0, 4.0) ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
234121	Thomas	Wendy	FR	4.00
333450	Smith	Andy	SO	2.00
123250	Phillips	Martin	SR	3.00

The IN Statement is an excellent way to look for multiple values for a column.

IN List vs. OR brings the same Results

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT * FROM Student_Table
WHERE  Grade_Pt IN (2.0, 3.0, 4.0) ;
```

Better Technique

```
SELECT*
FROM    Student_Table
WHERE   Grade_Pt = 2.0
OR      Grade_Pt = 3.0
OR      Grade_Pt = 4.0 ;
```

Both Produce the same results

The IN Statement avoids retyping the same column name separated by an OR. The IN allows you to search the same column for a list of values. Both queries above are equal, but the IN list is a nice way to keep things easy and organized.

Using a NOT IN List

```
SELECT *
FROM    Student_Table
WHERE  Grade_Pt NOT IN (2.0, 3.0, 4.0);
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
231222	Wilson	Susie	SO	3.80
280023	McRoberts	Richard	JR	1.90
322133	Bond	Jimmy	JR	3.95
125634	Hanson	Henry	FR	2.88
324652	Delaney	Danny	SR	3.35

You can also ask to see the results that ARE NOT IN your parameter list. That requires the column name and a NOT IN. Neither the IN nor NOT IN can search for NULLs!

A Technique for Handling Nulls with a NOT IN List

```
SELECT *
FROM    Student_Table
WHERE  Grade_Pt NOT IN (2.0, 3.0, 4.0)
OR      Grade_Pt IS NULL ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
231222	Wilson	Susie	SO	3.80
280023	McRoberts	Richard	JR	1.90
322133	Bond	Jimmy	JR	3.95
125634	Hanson	Henry	FR	2.88
324652	Delaney	Danny	SR	3.35
260000	Johnson	Stanley	?	?

This is a great technique to look for a NULL when using a NOT IN List.

An IN List with the Keyword ANY

```
SELECT *
FROM    Student_Table
WHERE  Grade_Pt = ANY (2.0, 3.0, 4.0);
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
234121	Thomas	Wendy	FR	4.00
333450	Smith	Andy	SO	2.00
123250	Phillips	Martin	SR	3.00

This is the same thing as using an IN. It's just another way of writing your SQL.

A NOT IN List with the Keywords NOT = ALL

```
SELECT  *
FROM     Student_Table
WHERE   Grade_Pt NOT = ALL (2.0, 3.0, 4.0) ;
```

Student_ID	Last_Name	First Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
231222	Wilson	Susie	SO	3.80
280023	McRoberts	Richard	JR	1.90
322133	Bond	Jimmy	JR	3.95
125634	Hanson	Henry	FR	2.88
324652	Delaney	Danny	SR	3.35

This is another way of doing a NOT IN. Notice the NOT = ALL and then the list.

BETWEEN is Inclusive

```
SELECT *
FROM   Student_Table
WHERE  Grade_Pt BETWEEN 2.0 AND 4.0;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
125634	Hanson	Henry	FR	2.88
231222	Wilson	Susie	SO	3.80
324652	Delaney	Danny	SR	3.35
322133	Bond	Jimmy	JR	3.95
234121	Thomas	Wendy	FR	4.00
333450	Smith	Andy	SO	2.00
123250	Phillips	Martin	SR	3.00

This is a BETWEEN. What this allows you to do is see if a column falls in a range. It is inclusive, meaning that in our example, we will be getting the rows that also have a 2.0 and 4.0 in their column!

BETWEEN Works for Character Data

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT     *
FROM     Student Table
WHERE     Last_Name BETWEEN 'L' AND 'LZ' ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00

The BETWEEN isn't just used with numbers. You can look to see if words falls between certain letters.

LIKE uses Wildcards Percent '%' and Underscore '_'

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT * FROM Student_Table
WHERE Last_Name LIKE 'SM%' ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
333450	Smith	Andy	SO	2.00

The wildcard percentage sign (%) is a wildcard for any number of characters. We are looking for anyone whose name starts with SM! In this example, the only row that would come back is 'Smith'.

LIKE command Underscore is Wildcard for one Character

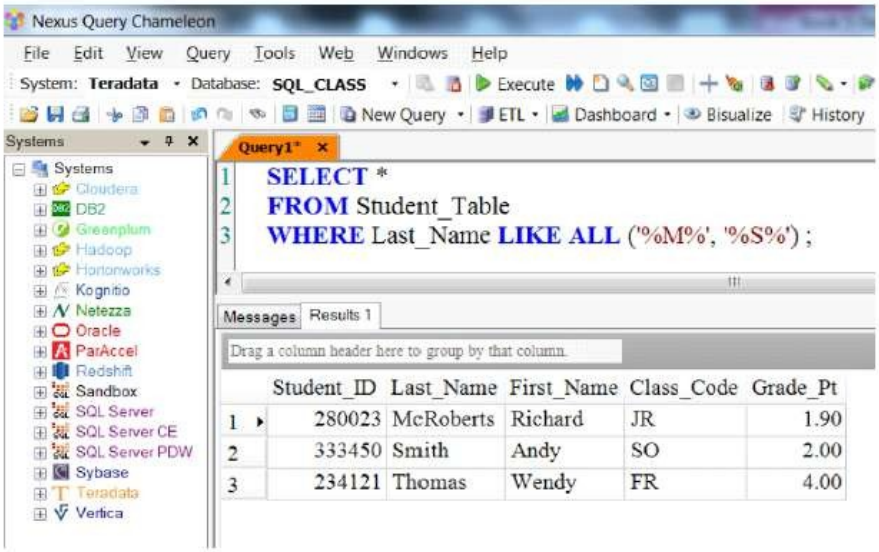
Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
SELECT * FROM Student_Table
WHERE Last_Name LIKE '_a%' ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88

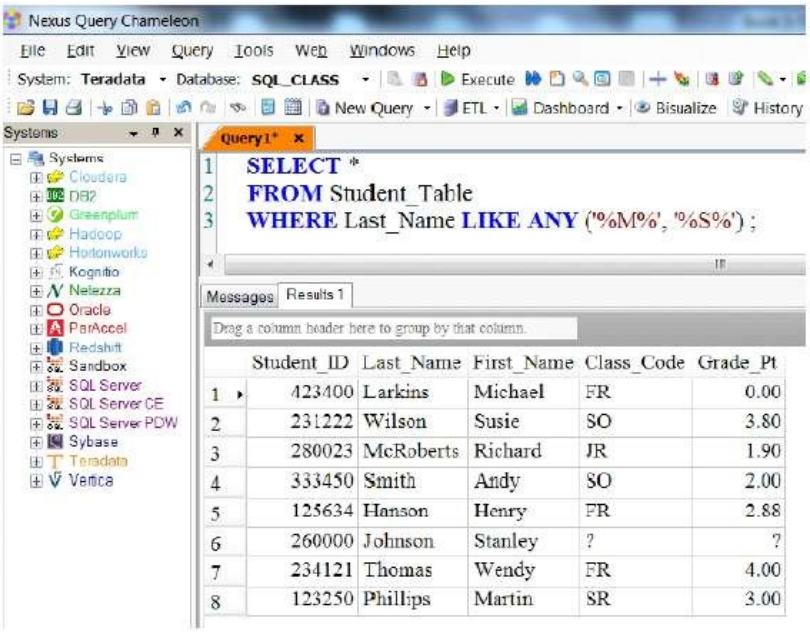
The second wild card is a '_' (underscore). An underscore represents a one character wildcard. Our search finds anyone with an 'a' in the second letter of their last name.

LIKE ALL means ALL conditions must be Met



What this syntax is looking for is any row that has a Last_Name with an 'S' AND an 'M' in it. It isn't looking for these in any order. As long as the Last_Name has an 'S' and an 'M' somewhere, it'll come back.

LIKE ANY means ANY of the Conditions can be Met



The word ANY means either an 'S' OR an 'M' in the Last_Name in any order.

IN ANSI Transaction Mode Case Matters

```
Student_ID      Last_Name      First_Name      Class_Code      Grade_Pt
----------      ---------      ----------      ----------      --------
```

423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
/* This query is in ANSI Transaction mode */
SELECT *                                   Capitol S  Small m
FROM     Student_Table                        ↓          ↓
WHERE  Last_Name LIKE ALL ('%S%', '%m%') ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
333450	Smith	Andy	SO	2.00

When in ANSI Transaction Mode, the system is CASE SENSITIVE, but it is not case sensitive in Teradata mode. Teradata mode is also referred to as BTET for Begin and End Transaction.

In Teradata Transaction Mode Case Doesn't Matter

```
/* This query is in Teradata (BT/ET) Transaction mode */
SELECT *                                   Capitol S  Small m
FROM     Student_Table                        ↓          ↓
WHERE  Last_Name LIKE ALL ('%S%', '%m%') ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
280023	McRoberts	Richard	JR	1.90
234121	Thomas	Wendy	FR	4.00
333450	Smith	Andy	SO	2.00

Case Sensitivity in Teradata (BTET) Transaction mode is not an issue.

LIKE Command Works Differently on Char Vs. Varchar

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
/* First_Name has a
Data Type
of VARCHAR (20) */
SELECT * FROM Student Table
WHERE First_Name LIKE '%y' ;
```

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
125634	Hanson	Henry	FR	2.88
322133	Bond	Jimmy	JR	3.95
324652	Delaney	Danny	SR	3.35

```
333450    Smith        Andy          SO              2.00
260000    Johnson      Stanley        ?                ?
234121    Thomas       Wendy         FR              4.00
```

It is important that you know the data type of the column you are using with your LIKE command. VARCHAR and CHAR data differ slightly.

Troubleshooting LIKE Command on Character Data

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

```
/* Last_Name has a Data Type of CHAR (20) */

SELECT * FROM Student_Table
WHERE Last_Name LIKE '%n' ;
```

```
Student_ID   Last_Name   First_Name   Class_Code   Grade_Pt

No Rows Returned
```

This is a CHAR (20) data type. That means that any words under 20 characters will pad spaces behind them until they reach 20 characters. You will not get any rows back from this example because, technically, no row ends in an 'N', but instead ends in a space. The good news is that there is a special technique to take care of this.

Introducing the TRIM Command

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

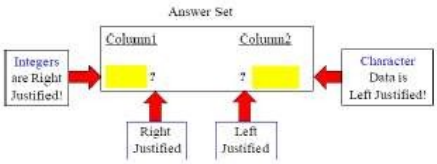
```
/* Last_Name has a Data Type of CHAR(20) */

SELECT Last_Name FROM Student_Table
WHERE TRIM (Last_Name) LIKE '%n' ;
```

```
Last_Name
Hanson
Wilson
Johnson
```

This is a CHAR (20) data type. That means that any words under 20 characters will pad spaces behind them until they reach 20 characters. That is why you TRIM the spaces. The spaces are removed and now rows return.

Quiz – Which Data is Left Justified and Which is Right?

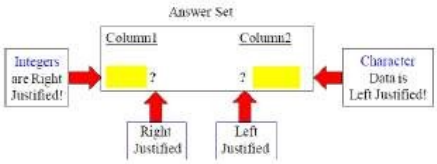
```
SELECT   *
FROM      Student_Table
WHERE    Column1 IS NULL
AND      Column2 IS NULL;
```



Which Column from the Answer Set could have a DATA TYPE of INTEGER and which could have Character Data?

Numbers are Right Justified and Character Data is Left

```
SELECT   *
FROM      Student_Table
WHERE    Column1 IS NULL
AND      Column2 IS NULL;
```



All Integers will start from the right and move left. Thus Col1 was defined during the table create statement to hold an INTEGER. The next page shows a clear example.

Answer – Which Data is Left Justified and Which is Right?

```
SELECT Employee_No, First_Name
FROM    Employee_Table
WHERE  Employee_No = 2000000;
```

All Integers will start from the right and move left. All Character data will start from the left and move to the right.

An Example of Data with Left and Right Justification

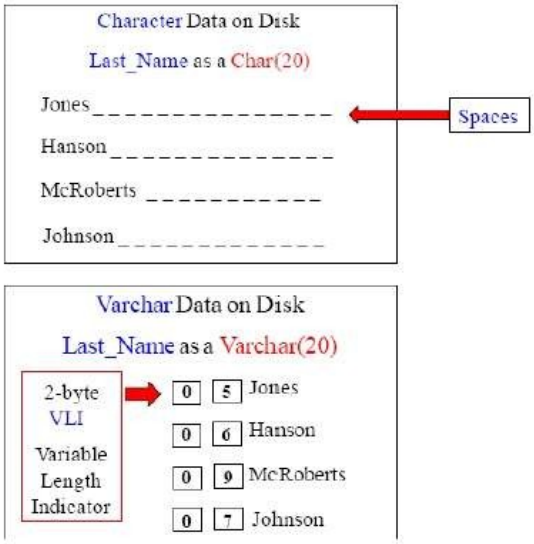
```
SELECT Student_ID, Last_Name
FROM     Student_Table ;
```

Integers are Right Justified!	Student_ID	Last_Name	Character Data is Left Justified!
	423400	Larkins	
	125634	Hanson	
	280023	McRoberts	
	260000	Johnson	
	231222	Wilson	
	234121	Thomas	

```
324652      Delaney
123250      Phillips
322133      Bond
333450      Smith
```

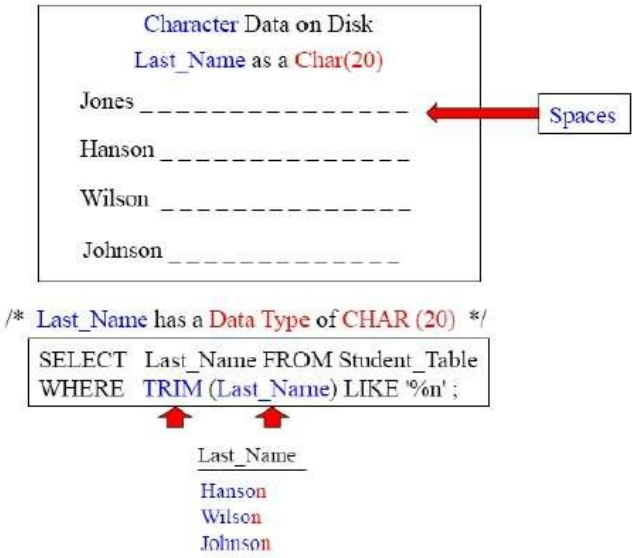
This is how a standard result set will look. Notice that the integer type in Student_ID starts from the right and goes left. Character data type in Last_Name moves left to right like we are used to seeing while reading English.

A Visual of CHARACTER Data vs. VARCHAR Data



Character data pads spaces to the right and Varchar uses a 2-byte VLI instead.

Use the TRIM command to remove spaces on CHAR Data




```
/* Last_Name has a Data Type of CHAR (20) */
SELECT  Last_Name FROM Student_Table
WHERE   TRIM (Last_Name) LIKE '%n' ;

Last_Name
Hanson
Wilson
Johnson
```

By using the TRIM command on the Last_Name column, you are able to trim off any spaces from the end. Now you can

find all names that end in 'n'.

TRIM Eliminates Leading and Trailing Spaces

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

/* Last_Name has a Data Type of CHAR(20) */

```
SELECT Last_Name FROM Student_Table
WHERE TRIM (Last_Name) LIKE '%n' ;
```

Last_Name
Hanson
Wilson
Johnson

Once we use the TRIM on Last_Name, we have eliminated any spaces at the end. So, now we are set to bring back anyone with a Last_Name that truly ends in 'n'!

Escape Character in the LIKE Command changes Wildcards

		Student_ID		
Student_Table	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00
999999	T_	S%	FR	1.90

/* We just pretended to add a new row to the Student_Table */

/* Can you use the LIKE command to find S% above? */

Here you will have to utilize a Wildcard Escape Character. Turn the page for more.

Escape Characters Turn off Wildcards in the LIKE Command

		Student_Table		
Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00

324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00
999999	T_	S%	FR	1.90

/* Can you use the LIKE command to find S% above? */

```
SELECT *
FROM Student_Table
WHERE First_Name LIKE 'S@%' Escape '@';
```

We can pick our Escape character, and we have chosen the @ sign. This turns the wildcard off for 1 character so we find 'S%', without bringing back Stanley or Susie.

Quiz – Turn off that Wildcard

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00
999999	T_	S%	FR	1.90

Can you use the LIKE command to find the Last_Name of T_(pronounced Tunderscore!)

This is a little trickier than you might think, so be on your toes…. And get a haircut!

ANSWER – To Find that Wildcard

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00
999999	T_	S%	FR	1.90

Can you use the LIKE command to find the Last_Name of T_(pronounced Tunderscore!)

```
SELECT * FROM Student_Table
WHERE TRIM(Last_Name) LIKE 'T@_' Escape '@';
```

You didn't really need to get a full haircut, but just a TRIM Command and the Escape!

Sample

Sample

Overview

"The universe extends beyond the mind of man, and is more complex than the small sample one can study."
- Kenneth L. Pike

The SAMPLE Function and Syntax

The syntax for the SAMPLE function:

```
SAMPLE [WITH REPLACEMENT]
[RANDOMIZED ALLOCATION]
              [WHEN <condition>  THEN]
              {<number-of-rows> | <percentage>}
[...,<number-of-rows> | <percentage>]
              [ELSE {<number-of-rows> |
<percentage }   END]
```

The Sampling function (SAMPLE) permits a SELECT to randomly return rows from a Teradata database table. It allows the request to specify either an absolute number of rows or a percentage of rows to return. Additionally, it provides an ability to return rows from multiple samples.

SAMPLE Function Examples

Bring back 5 rows

```
SELECT   *
FROM Student_Course_Table
SAMPLE 5 ;
```

Student_ID	Course_ID
125634	100
231222	220
125634	200
333450	400
260000	400

Bring back 25% of the rows

```
SELECT  *
FROM Student_Course_Table
SAMPLE .25 ;
```

Student_ID	Course_ID
125634	100
322133	300
260000	400
333450	400

The above example uses the SAMPLE to get a random sample of the sales table. Notice that five rows came back because we asked for a SAMPLE 5 in the first example. In the second example, we got 25% of the rows.

A SAMPLE Example that asks for Multiple Samples

Bring back two Samples of 25%, but no duplicates!

```
SELECT   *
FROM Student_Course_Table
SAMPLE .25, .25
ORDER BY 1,2 ;
```

Student_ID	Course_ID
125634	100
125634	220
231222	210
231222	220
260000	400
322133	220
322133	300
324652	200

Sometimes, a single sampling of the data is not sufficient. The SAMPLE function can be used to request more than one sample by listing either the number of rows or the percentage of the rows to be returned. The above example uses the SAMPLE function to request multiple samples.

A SAMPLE Example with the SAMPLEID

```
SELECT Student_ID
       ,Course_ID
       ,SAMPLEID
FROM Student_Course_Table
SAMPLE 5, 5, 5
ORDER BY 3, 1, 2 ;
```

Bring back 3 Samples with each sample having 5 rows.

Student_ID	Course_ID	SAMPLEID
125634	100	1
125634	200	1
231222	220	1
234121	100	1
322133	220	1
123250	100	2
125634	220	2
231222	210	2
280023	210	2
333450	400	2
260000	400	3
322133	300	3
324652	200	3
333450	500	3

Why did the last sample Only bring back 4 rows?

Although multiple samples were taken, the rows came back as a single answer set consisting of 5 rows, 5 rows, and then 4 rows of the data. The SAMPLEID column name can be used to distinguish between each sample. The last sample only brought back 4 rows because there are only 14 rows in the table, and there will be no duplicates.

A SAMPLE Example WITH REPLACEMENT

```
SELECT Student_ID
       ,Course_ID
       ,SAMPLEID
FROM Student_Course_Table
SAMPLE WITH REPLACEMENT 5, 5, 5
ORDER BY 3, 1, 2 ;
```

Bring back 3 Samples with each sample having 5 rows.
You can have duplicates now!

Student_ID	Course_ID	SAMPLEID
231222	210	1
231222	210	1
280023	210	1
280023	210	1
280023	210	1
125634	100	2
125634	220	2
125634	220	2
322133	220	2
322133	300	2
125634	220	3
234121	100	3
260000	400	3
322133	300	3
322133	300	3

At the same time, you may wish for rows to be available for all samples. The above example uses the SAMPLE WITH REPLACEMENT function with the SAMPLEID to request multiple samples and denote which sample each row came from.

A SAMPLE Example with Four 10% Samples

```
SELECT Student_ID
       ,Course_ID
       ,SAMPLEID
FROM Student_Course_Table
SAMPLE .1, .1, .1, .1
ORDER BY SAMPLEID ;
```

Bring back 4 Samples with each sample having 10% of the rows.

Student_ID	Course_ID	SAMPLEID
123250	100	1
125634	100	2
280023	210	3
234121	100	4

The above example uses the SAMPLE function with the SAMPLEID to request multiple samples as a percentage and denote which sample each row came from.Although 10% of 14 rows is 1.4, it can only return a whole row, and therefore, 1 row is returned per sample. Also, since SAMPLEID is a column, it can be used as the sort key.

A Randomized SAMPLE

```
SELECT Student_ID
      ,Course_ID
      ,SAMPLEID
FROM Student_Course_Table
SAMPLE RANDOMIZED ALLOCATION .1, .1, .1, .1;
```

Bring back 4 Samples with each sample having 10% of the rows, and do a random sample across the entire population.

Student_ID	Course_ID	SAMPLEID
125634	100	1
125634	200	3
125634	220	4
324652	200	2

By default, the SAMPLE function does a proportional sampling across all AMPs in the system. Therefore, it is not a simple random sample across the entire population of rows. If you want a random sample across the entire population, use the RANDOMIZED ALLOCATION as seen above.

A SAMPLE with Conditional Logic

```
SELECT Student_ID
      ,Course_ID
      ,SAMPLEID
FROM Student_Course_Table
SAMPLE RANDOMIZED ALLOCATION
           WHEN Course_ID >200 THEN .1, .1 ELSE .2, .2
           END
ORDER BY 3;
```

Bring back two Samples with one row per sample if the Course_ID is <= 200.

Bring back two Samples with two rows each if the Course_ID > 200.

Student_ID	Course_ID	SAMPLEID
260000	400	1
231222	220	2
125634	100	3
123250	100	4

The above query brings back two Samples with one row per sample if the Course_ID is <= 200. Else, it bring back two Samples with two rows each if the Course_ID > 200. This means it will attempt to bring back six records total in four different samples.

Aggregates and A SAMPLE using a Derived Table

```
SELECT  count(distinct(Course_ID))
       FROM (SEL Course_ID FROM Student_Course_Table
SAMPLE 5) DT ;
```

COUNT(Distinct(Course_ID))

```
4
```

A second run of the same SELECT might very well yield these results:

```
COUNT(Distinct(Course_ID))
5
```

Although they look like Aggregates, they are not normally compatible with them in the same SELECT list. As demonstrated here, aggregation can be performed. However, they must be calculated in a temporary or derived table.

The above example uses the SAMPLE function to request multiple samples to create a derived table (See Temporary Tables chapter). Then, the unique rows will be counted to show the random quality of the SAMPLE function.

Random Number Generator

```
The syntax for RANDOM is:

RANDOM(<low-literal-value>, <high-literal-value>)
```

The example below uses the RANDOM function to return a random number between 1 and 20:

```
SELECT RANDOM(1, 20);

Random(1,20)
14
```

The RANDOM function generates a random number that is inclusive for the numbers specified in the SQL that is greater than or equal to the first argument, and less than or equal to the second argument.

The RANDOM function may be used in the SELECT list, in a CASE, in a WHERE clause, in a QUALIFY, in a HAVING, and in an ORDER BY. The RANDOM function can be used creatively to provide some powerful functionality within an SQL statement.

Using Random to SELECT a Percentage of Rows

The next SELECT uses RANDOM to randomly select 5% of the rows from the table:

```
SELECT *
FROM Sales_Table
WHERE RANDOM (1, 100) = 5;

Product_ID      Sale_Date     Daily_Sales
1000            09/28/2000    48850,40
```

There is roughly a 5% (1 out of 100) chance that a row will be returned using RANDOM in the WHERE clause, completely at random. Since SAMPLE randomly selects rows out of spool, currently RANDOM will be faster than SAMPLE. However, SAMPLE will be more accurate regarding the number of rows being returned with both the percent and row count.

Using Random and Aggregations

This example uses RANDOM to randomly generate a number that will determine which rows from the aggregation will be returned:

```
SELECT Product_ID, COUNT(Daily_Sales)
FROM Sales_Table
GROUP BY 1
```

```
        HAVING COUNT(Daily_Sales) > RANDOM(1, 10) ;

 Product_ID    Count(Daily_Sales)
       2000                     7
       3000                     7
```

This last example uses RANDOM to randomly generate a number that will determine which rows from the aggregation will be returned. Whenever a random number is needed within the SQL, RANDOM is a great tool.

Set Operators Functions

Set Operators Functions

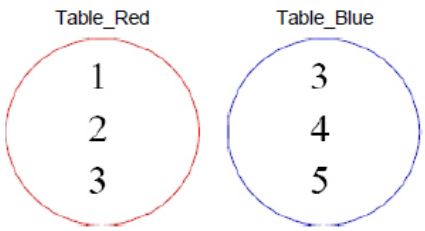
Overview

"Good advice is something a man gives when he is too old to set a bad example"
- Francois de la Rouchefoucauld

Rules of Set Operators

1. Each query will have two SELECT Statements separated by a SET Operator
2. SET Operators are UNION, INTERSECT, or EXCEPT/MINUS
3. Must specify the same number of columns from the same domain (data type/range)
4. If using Aggregates, both SELECTs much have their own GROUP BY
5. Both SELECTS must have a FROM Clause
6. The First SELECT is used for all ALIAS, TITLE, and FORMAT Statements
7. The Second SELECT will have the ORDER BY statement, which must be a number
8. When multiple operators the order of precedence is INTERSECT, UNION, and EXCEPT/MINUS
9. Parentheses can change the order of Precedence
10. Duplicate rows are eliminated in the spool, unless the ALL keyword is used

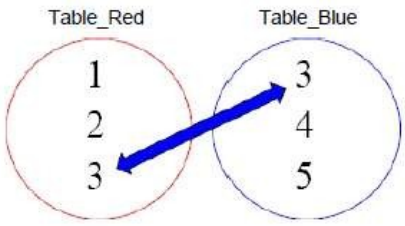
INTERSECT Explained Logically




```
SELECT * FROM Table_Red
INTERSECT
SELECT * FROM Table_Blue;
```

In this example, what numbers in the answer set would come from the query above?

INTERSECT Explained Logically



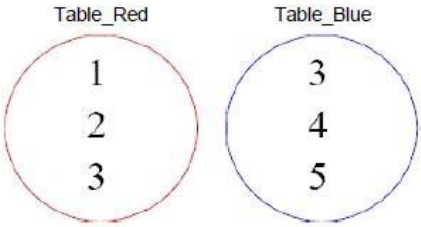

```
SELECT * FROM Table_Red
INTERSECT
```

```
SELECT * FROM Table_Blue;
```

3

In this example, only the number 3 was in both tables. So they INTERSECT.

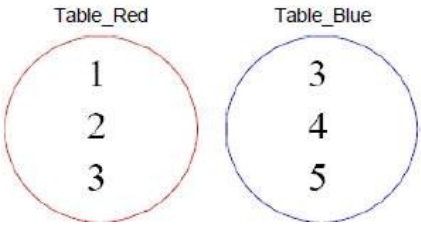
UNION Explained Logically




```
SELECT * FROM Table_Red
UNION
SELECT * FROM Table_Blue;
```

In this example, what numbers in the answer set would come from the query above?

UNION Explained Logically

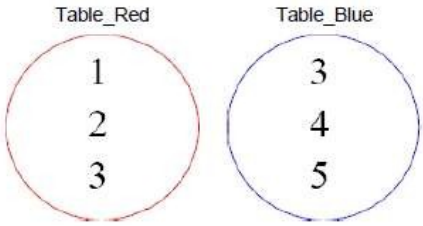



```
SELECT * FROM Table_Red
UNION
SELECT * FROM Table_Blue;
```

1 2 3 4 5

The top and bottom queries run simultaneously. Then, the two different spools files are merged to eliminate duplicates and place the remaining numbers in the answer set.

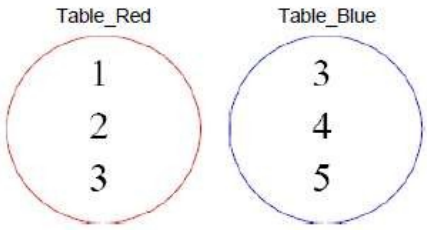
UNION ALL Explained Logically




```
SELECT * FROM Table_Red
UNION ALL
SELECT * FROM Table_Blue;
```

In this example, what numbers in the answer set would come from the query above?

UNION Explained Logically

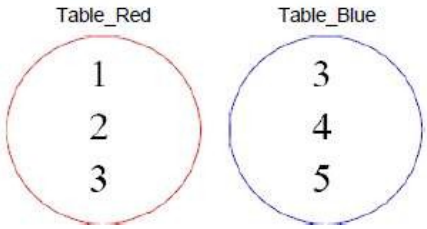



```
SELECT * FROM Table_Red
UNION ALL
SELECT * FROM Table_Blue;
```

1 2 3 3 4 5

Both top and bottom queries run simultaneously. Then, the two different spools files are merged together to build the answer set. The ALL prevents eliminating Duplicates.

EXCEPT Explained Logically

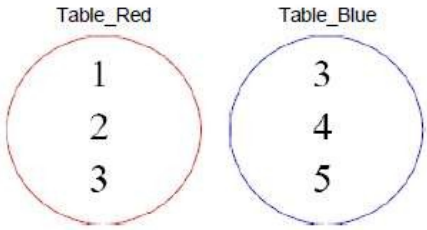



```
SELECT * FROM Table_Red
EXCEPT
SELECT * FROM Table_Blue;
```

EXCEPT and MINUS do the exact same thing so either word will work!

In this example, what numbers in the answer set would come from the query above?

EXCEPT Explained Logically



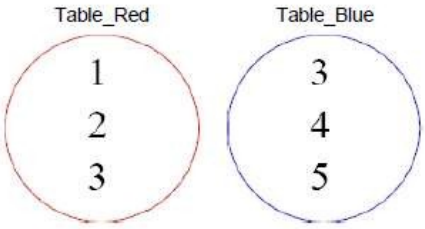

```
SELECT * FROM Table_Red
EXCEPT
```

```
SELECT * FROM Table_Blue;
```

1 2

The Top query SELECTED 1, 2, 3 from Table_Red. From that point on, only 1, 2, 3 at most could come back. The bottom query is run on Table_Blue and if there are any matches they are not ADDED to the 1, 2, 3 but instead take away either the 1, 2, or 3.

Minus Explained Logically

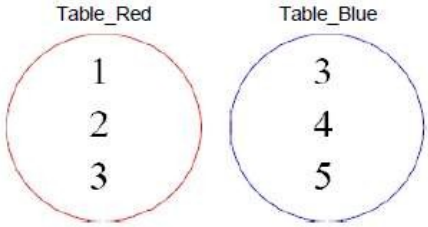



```
SELECT * FROM Table_Blue
MINUS
SELECT * FROM Table_Red;
```

EXCEPT and MINUS do the exact same thing so either word will work!

What will the answer set be? Notice, I changed the order of the tables in the query!

Minus Explained Logically



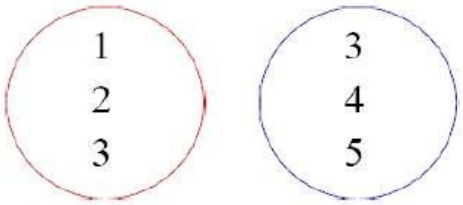

```
SELECT * FROM Table_Blue
MINUS
SELECT * FROM Table_Red;
```

4 5

The Top query SELECTED 3, 4, 5 from Table_Blue. From that point on, only 3, 4, 5 at most could come back. The bottom query is run on Table_Red and if there are any matches they are not ADDED to the 3, 4, 5, but instead take away either the 3, 4, or 5.

Testing Your Knowledge

Table_Red Table_Blue




```
SELECT *
FROM Table_Blue
EXCEPT
SELECT *
FROM Table_Red;
```

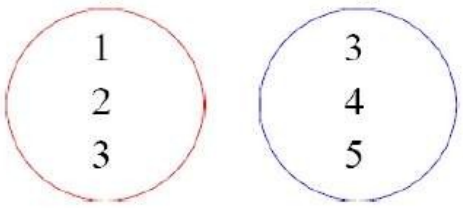
```
SELECT *
FROM Table_Blue
MINUS
SELECT *
FROM Table_Red;
```

Will the result set be the same for both queries above?

Both queries above are exactly the same to the system and produce the same result set.

Testing Your Knowledge

Table_Red Table_Blue




```
SELECT *
FROM Table_Blue
EXCEPT
SELECT *
FROM Table_Red;
```

```
SELECT *
FROM Table_Red
MINUS
SELECT *
FROM Table_Blue;
```

Will the result set be the same for both queries above?

No! The first query returns 4, 5 and the query on the right returns 1, 2.

An Equal Amount of Columns in both SELECT List

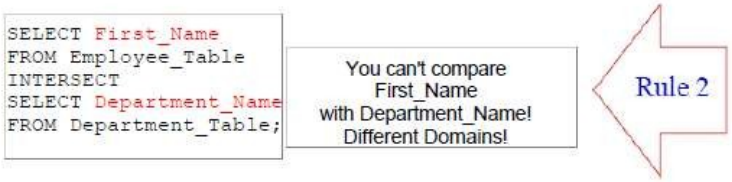
```
SELECT Dept_No
      ,Employee_No
FROM Employee_Table
INTERSECT
SELECT Dept_No
      ,Mgr_No
FROM Department_Table;
```

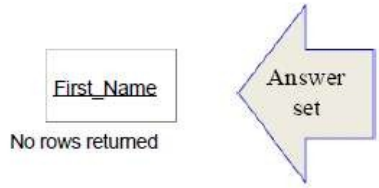
Both queries have the same number of columns in the SELECT list.

Rule 1

You must have an equal amount of columns in both SELECT lists. This is because data is compared from the two spool files and duplicates are eliminated. So, for comparison purposes, there must be an equal amount of columns in both queries.

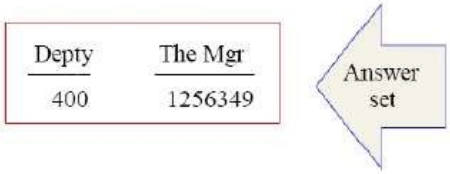
Columns in the SELECT list should be from the same Domain





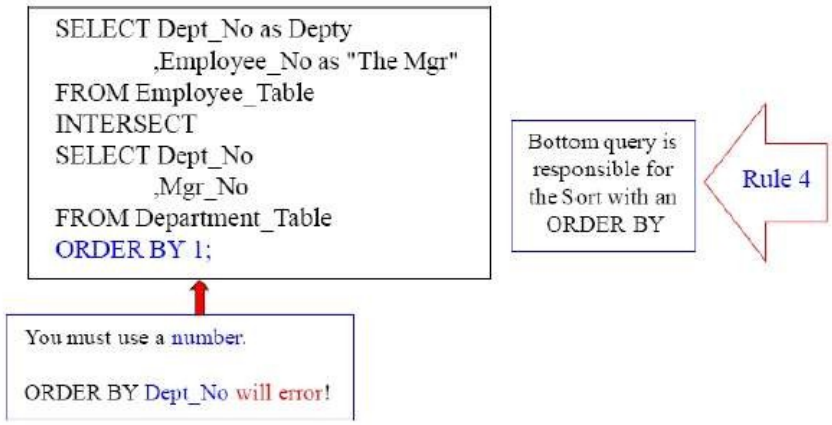
The above query works without error, but no data is returned. There are no First Names that are the same as Department Names. This is like comparing Apples to Oranges. That means they are NOT in the same Domain.

The Top Query handles all Aliases



The Top Query is responsible for ALIASING.

The Bottom Query does the ORDER BY (a Number)



The Bottom Query is responsible for sorting, but the ORDER BY statement must be a number which represents column1, column2, column3, etc.

Great Trick: Place your Set Operator in a Derived Table

```
SELECT Employee_No AS MANAGER
      ,Trim(Last_Name) || ',' || First_Name as "Name"
FROM Employee_Table
INNER JOIN
       (SELECT Employee_No FROM Employee_Table
        INTERSECT
        SELECT Mgr_No       FROM Department_Table)
          AS TeraTom (empno)
ON Employee_No = empno
ORDER BY "Name"
```

MANAGER	Name
1256349	Harrison, Herbert
1333454	Smith, John
1000234	Smythe, Richard
1121334	Strickling, Cletus

The Derived Table gave us the empno for all managers and we were able to join it.

UNION vs. UNION ALL

If it is expected that a UNION of two Answer sets will not create duplicate rows, or
the SQL merges them together anyway via a Group By, the use of Union All will provide the same result while consuming less resources.

```
Syntax:

Select Col1, Col2, Col3 from First_Table
UNION ALL
Select Col1, Col2, Col3 from Second_Table
Group By 1,2,3;
```

Real Example

```
SELECT Employee_No, Dept_No, First_Name, Last_Name, Salary
       FROM Hierarchy_Table
UNION ALL
SELECT Employee_No, Dept_No, First_Name, Last_Name, Salary
       FROM Employee_Table
GROUP BY 1, 2, 3, 4, 5
ORDER BY 2;
```

Unions will get better performance and use less system resources when using a Union ALL. Unless the ALL option is used, there is overhead to eliminate duplicate rows from each result set and from the final result.

UNION vs. UNION ALL Example

```
SELECT Department_Name, Dept_No from Department_Table
UNION ALL
SELECT Department_Name, Dept_No from Department_Table
ORDER BY 1;
```

UNION Answer Set

Department_Name	Dept_No
Customer Support	400
Human Resources	500
Marketing	100
Research and Development	200
Sales	300

UNION ALL Answer Set

Department_Name	Dept_No
Customer Support	400
Customer Support	400
Human Resources	500
Human Resources	500
Marketing	100
Marketing	100
Research and Development	200
Research and Development	200
Research and Development	300
Sales	300
Sales	

UNION eliminates duplicates, but UNION ALL does not.

Using UNION ALL and Literals

```
SELECT Dept_No AS Dept
  ,'Employee'      (TITLE ' ')
  ,First_Name ||''|| Last_Name
                     as "Name"
FROM Employee_Table
UNION ALL
SELECT Dept_No
  ,'Department'
  ,Department_Name
FROM Department_Table
ORDER BY 1, 2;
```

Dept	_____	Name
?	Employee	Squiggy Jones
10	Employee	Richard Smythe
100	Department	Marketing
100	Employee	Mandee Chambers
200	Department	Research and
200	Employee	Develop
200	Employee	Billy Coffing
300	Department	John Smith
300	Employee	Sales
400	Department	Loraine Larkins
400	Employee	Customer Support
400	Employee	Cletus Strickling
400	Employee	Herbert Harrison
500	Department	William Reilly
		Human Resources

Notice the 2nd SELECT column in that it is a literal 'Employee' (with two spaces) and the other Literal is 'Department'. These literals match up because now they are both 10 characters long exactly. The UNION ALL brings back all Employees and all Departments and shows the employees in each valid department.

A Great Example of how EXCEPT works

```
SELECT Dept_No as Department_Number
FROM Department_Table
EXCEPT
SELECT Dept_No
FROM Employee_Table
ORDER BY 1;
```

Department number
500

This query brought back all Departments without any employees.

USING Multiple SET Operators in a Single Request

```
SELECT Dept_No , Employee_No as empno
FROM Employee_Table
UNION ALL
SELECT Dept_No, Employee_No
FROM Employee_Table
INTERSECT ALL
SELECT Dept_No, Mgr_No
FROM Department_Table
MINUS
SELECT Dept_No, Mgr_No
FROM Department_Table
WHERE Department_Name LIKE '%Sales%'
ORDER BY 1, 2;
```

Dept_NO	Empno
?	2000000
10	1000234
100	1232578
200	1324657
200	1333454
300	2312225
400	1121334
400	1256349
400	2341218

Above, we use multiple SET Operators. They follow the natural Order of Precedence in that UNION is evaluated first, and then INTERSECT, and finally MINUS.

Changing the Order of Precedence with Parentheses

```
SELECT Dept_No , Employee_No as empno
FROM Employee_Table
UNION ALL
(SELECT Dept_No, Employee_No
FROM Employee_Table
INTERSECT ALL
(SELECT Dept_No, Mgr_No
FROM Department_Table
MINUS
SELECT Dept_No, Mgr_No
FROM Department_Table
WHERE Department_Name LIKE '%Sales%'))
ORDER BY 1, 2;
```

Dept_NO	Empno
?	2000000
10	1000234
100	1232578
200	1324657
200	1333454
300	2312225
400	1121334
400	1256349
400	1256349
400	2341218

Above, we use multiple SET Operators and Parentheses to change the order of precedence. Above, the EXCEPT runs first, then the INTERSECT, and lastly, the UNION. The natural Order of Precedence, without parentheses, is UNION, INTERSECT, and finally EXCEPT or MINUS.

Using UNION ALL for speed in Merging Data Sets

Cust_Table_East	Cust_Table_West	Combined_Custs

1,000,000 rows of Eastern Customers	1,000,000 rows of Western Customers	Completely empty

```
INSERT INTO Combined_Custs
  SEL * FROM Cust_Table_East
   UNION ALL
  SEL * FROM Cust_Table_West ;
```

Combined_Custs

2,000,000 rows of
East and West Customers

Because the Combined_Custs table started empty, there is no Transient Journal taking pictures for Rollback purposes. So, this dramatically increases the speed. This one transaction sees both SELECT statements run in parallel and then merge into one.

Using UNION to be same as GROUP BY GROUPING SETS

```
SELECT Product_ID as PROD_ID
       ,NULL as Yr
       ,NULL as Mth
       ,SUM(Daily_Sales)
FROM   Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT  NULL
       ,EXTRACT(Year from Sale_Date)
       ,NULL
       ,SUM(Daily_Sales)
FROM   Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT NULL
       ,NULL
       ,EXTRACT(Month from Sale_Date)
       ,SUM(Daily_Sales)
FROM   Sales_Table
GROUP BY 1, 2, 3
ORDER BY 1 DESC, 2, 3 ;
```

GROUP BY must be used in all three SELECTs

Prod_ID	Yr	Mth	Sum(Daily_Sales)
3000	?	?	224587.82
2000	?	?	306611.81
1000	?	?	331204.72
?	?	9	418769.36
?	?	10	443634.99
?	2000	?	862404.35

Using UNION to be same as GROUP BY ROLLUP

```
SEL Product_ID as PROD_ID
   ,EXTRACT(Year from Sale_Date) as Yr
```

Prod_ID	Yr	Mth	Total

```
   ,EXTRACT (Month from Sale_Date) as Mth
,SUM(Daily_Sales) as "Total" FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SEL Product_ID
   ,EXTRACT(Year from Sale_Date)
   ,NULL
   ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT Product_ID
      ,NULL
      ,NULL
      ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT NULL, NULL, NULL
      ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
ORDER BY 1 DESC, 2, 3;
```

3000	?	?	224587.82
3000	2000	?	224587.82
3000	2000	9	139679.76
3000	2000	10	84908.06
2000	?	?	306611.81
2000	2000	?	306611.81
2000	2000	9	139738.91
2000	2000	10	166872.90
1000	?	?	331204.72
1000	2000	?	331204.72
1000	2000	9	139350.69
1000	2000	10	191854.03
?	?	?	862404.35

Using UNION to be the same as GROUP BY Cube

```
SEL Product_ID as PROD_ID
    ,EXTRACT(Year from Sale_Date) as Yr
    ,EXTRACT (Month from Sale_Date) as Mth
  ,SUM(Daily_Sales) as "Total"
FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SEL Product_ID
    ,NULL
    ,EXTRACT(Year from Sale_Date)
  ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT Product_ID
      ,NULL
      ,NULL
   SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION

            CONTINUED on Right
```

```
SELECT NULL
    ,EXTRACT(Year from Sale_Date)
    ,EXTRACT (Month from Sale_Date)
    ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT NULL
   ,EXTRACT(Year from Sale_Date)
   ,NULL
   ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT NULL, NULL
   ,EXTRACT (Month from Sale_Date)
   ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT NULL, NULL, NULL
   ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
ORDER BY 1 DESC, 2 DESC, 3 DESC ;
```

Using UNION to be same as GROUP BY Cube

```
SELECT NULL
```

```
SEL ProductJD as PROD JD
  ,EXTRACT(Year from Sale_Date) as Yr
  ,EXTRACT (Month from Sale_Date) as Mth
,SUM(Daily_Sales) as "Total"
FROM Sales_Table
GROUP BY 1,2,3
UNION
SEL ProductJD
  ,NULL
  ,EXTRACT(Year from Sale_Date)
,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1,2,3
UNION
SELECT ProductJD
      ,NULL
      ,NULL
    ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1,2,3
UNION
```

CONTINUED on Right

```
 ,EXTRACT(Year from Sale_Date)
 ,EXTRACT (Month from Sale_Date)
 ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1,2, 3
UNION
SELECT NULL
 ,EXTRACT(Year from Sale_Date)
 ,NULL
 ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1,2,3
UNION
SELECT NULL, NULL
 ,EXTRACT (Month from Sale_Date)
 ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1,2,3
UNION
SELECT NULL, NULL, NULL
 ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1,2,3
ORDER BY 1 DESC, 2 DESC, 3 DESC ;
```

Using UNION to be same as GROUP BY Cube

```
SEL Product_ID as PROD_ID
   ,EXTRACT(Year from Sale_Date) as Yr
   ,EXTRACT (Month from Sale_Date) as Mth
,SUM(Daily_Sales) as "Total" FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SEL Product_ID
   ,EXTRACT(Year from Sale_Date)
   ,NULL
   ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT Product_ID
      ,NULL
      ,NULL
      ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
UNION
SELECT NULL, NULL, NULL
      ,SUM(Daily_Sales) FROM Sales_Table
GROUP BY 1, 2, 3
ORDER BY 1 DESC, 2, 3;
```

Prod_ID	Yr	Mth	Total
3000	?	?	224587.82
3000	2000	?	224587.82
3000	2000	9	139679.76
3000	2000	10	84908.06
2000	?	?	306611.81
2000	2000	?	306611.81
2000	2000	9	139738.91
2000	2000	10	166872.90
1000	?	?	331204.72
1000	2000	?	331204.72
1000	2000	9	139350.69
1000	2000	10	191854.03
?	?	?	862404.35

Statistical Aggregate Functions

Statistical Aggregate Functions

Overview

"It's not that figures lie, it's that liars figure."
- Anonymous

The Stats Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

Above is the Stats_Table data in which we will use in our statistical examples.

The KURTOSIS Function

```
SELECT KURTOSIS(col1) AS KofCol1
FROM Stats_Table;
```

The KURTOSIS function is used to return a number that represents the sharpness of a peak on a plotted curve of a probability function for a distribution compared with the normal distribution.

A high value result is referred to as leptokurtic, a medium result is referred to as mesokurtic, and a low result is referred to as platykurtic.

A positive value indicates a sharp or peaked distribution, and a negative number represents a flat distribution. A peaked distribution means that one value exists more often than the other values. A flat distribution means there is the same quantity values exist for each number.

If you compare this to the row distribution associated within Teradata, most of the time a flat distribution is best with the same number of rows stored on each AMP. Having skewed data represents more of a lumpy distribution.

A Kurtosis Example

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT KURTOSIS(col1) AS KofCol1
      ,KURTOSIS(col2) AS KofCol2
      ,KURTOSIS(col3) AS KofCol3
      ,KURTOSIS(col4) AS KofCol4
      ,KURTOSIS(col5) AS KofCol5
      ,KURTOSIS(col6) AS KofCol6
FROM Stats_Table;
```

KofCol1	KofCol2	KofCol3	KofCol4	KofCol5	KofCol6
-1	-1	1	-1	-1	-1

The SKEW Function

The SKEW Function

```
SELECT SKEW(col1) AS SKofCol1
FROM Stats_Table;
```

The Skew indicates that a distribution does not have equal probabilities above and below the mean (average). In a skew distribution, the median and the mean are not coincident or equal.

Where:

- A median value < mean value = a positive skew
- A median value > mean value = a negative skew
- A median value = mean value = no skew

Syntax for using SKEW:

SKEW(<column-name>)

A SKEW Example

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20

```
SELECT SKEW(col1) AS SKofCol1
      ,SKEW(col2) AS SKofCol2
      ,SKEW(col3) AS SKofCol3
      ,SKEW(col4) AS SKofCol4
      ,SKEW(col5) AS SKofCol5
      ,SKEW(col6) AS SKofCol6
FROM Stats_Table;
```

6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

SKofCol1	SKofCol2	SKofCol3	SKofCol4	SKofCol5	SKofCol6
0	-0	1	0	0	-0

A median value < mean value = a positive skew
A median value > mean value = a negative skew
A median value = mean value = no skew

The STDDEV_POP Function

The STDDEV_POP Function

```
SELECT STDDEV_POP(col1) AS SDPCol1
FROM Stats_Table;
```

The standard deviation function is a statistical measure of spread or dispersion of values. It is the root's square of the difference of the mean (average). This measure is to compare the amount by which a set of values differs from the arithmetical mean.

The STDDEV_POP function is one of two that calculates the standard deviation. The population is of all the rows included based on the comparison in the WHERE clause.

Syntax for using STDDEV_POP:

STDDEV_POP(<column-name>)

A STDDEV_POP Example

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40

```
SELECT STDDEV_POP(col1) AS SDPCol1
      ,STDDEV_POP(col2) AS SDPCol2
      ,STDDEV_POP(col3) AS SDPCol3
      ,STDDEV_POP(col4) AS SDPCol4
      ,STDDEV_POP(col5) AS SDPCol5
      ,STDDEV_POP(col6) AS SDPCol6
FROM Stats_Table;
```

13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

SDPCol1	SDPCol2	SDPCol3	SDPCol4	SDPCol5	SDPCol6
9	4	14	9	4	27

The standard deviation function is a statistical measure of spread or dispersion of values. It is the root's square of the difference of the mean (average). This measure is to compare the amount by which a set of values differs from the arithmetical mean.

The STDDEV_SAMP Function

```
SELECT STDDEV_SAMP(col1) AS SDSCol1
FROM Stats_Table;
```

The standard deviation function is a statistical measure of spread or dispersion of values. It is the root's square of the difference of the mean (average). This measure is to compare the amount by which a set of values differs from the arithmetical mean.

The STDDEV_SAMP function is one of two that calculates the standard deviation. The sample is a random selection of all rows returned based on the comparisons in the WHERE clause. The population is for all of the rows based on the WHERE clause.

Syntax for using STDDEV_SAMP:

STDDEV_SAMP(<column-name>)

A STDDEV_SAMP Example

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70

```
SELECT STDDEV_POP(col1) AS SDSCol1
      ,STDDEV_POP(col2) AS SDSCol2
      ,STDDEV_POP(col3) AS SDSCol3
      ,STDDEV_POP(col4) AS SDSCol4
      ,STDDEV_POP(col5) AS SDSCol5
      ,STDDEV_POP(col6) AS SDSCol6
FROM Stats_Table;
```

SDSCol1	SDSCol2	SDSCol3	SDSCol4	SDSCol5	SDSCol6
9	4	14	9	5	27

The STDDEV_SAMP function is one of two that calculates the standard deviation. The sample is a random selection of all rows returned based on the comparisons in the WHERE clause. The population is for all of the rows based on the WHERE clause.

25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

The VAR_POP Function

```
SELECT VAR_POP(col1) AS VPCol1
FROM Stats_Table;
```

The Variance function is a measure of dispersion (spread of the distribution) as the square of the standard deviation. There are two forms of Variance in Teradata. VAR_POP is for the entire population of data rows allowed by the WHERE clause.

Although standard deviation and variance are regularly used in statistical calculations, the meaning of variance is not easy to elaborate. Most often, variance is used in theoretical work where a variance of the sample is needed.

There are two methods for using variance. These are the Kruskal-Wallis one-way Analysis of Variance, and Friedman two-way Analysis of Variance by rank.

Syntax for using VAR_POP:

VAR_POP(<column-name>)

A VAR_POP Example

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT VAR_POP(col1) AS VPCol1
      ,VAR_POP(col2) AS VPCol2
      ,VAR_POP(col3) AS VPCol3
      ,VAR_POP(col4) AS VPCol4
      ,VAR_POP(col5) AS VPCol5
      ,VAR_POP(col6) AS VPCol6
FROM Stats_Table;
```

VPCol1	VPCol2	VPCol3	VPCol4	VPCol5	VPCol6
75	19	191	75	20	723

The Variance function is a measure of dispersion (spread of the distribution) as the square of the standard deviation. There are two forms of Variance in Teradata, VAR_POP is for the entire population of data rows allowed by the WHERE clause.

The VAR_SAMP Function

```
SELECT VAR_SAMP(col1) AS VSCol1
FROM Stats_Table;
```

The Variance function is a measure of dispersion (spread of the distribution) as the square of the standard deviation. There are two forms of Variance in Teradata, VAR_SAMP is used for a random sampling of the data rows allowed through by the WHERE clause.

Although standard deviation and variance are regularly used in statistical calculations, the meaning of variance is not easy to elaborate. Most often, variance is used in theoretical work where a variance of the sample is needed to look for consistency.

There are two methods for using variance. These are the Kruskal-Wallis one-way Analysis of Variance and Friedman two-way Analysis of Variance by rank.

Syntax for using VAR_SAMP:

VAR_SAMP(<column-name>)

A VAR_SAMP Example

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT VAR_SAMP(col1) AS VSCol1
      ,VAR_SAMP(col2) AS VSCol2
      ,VAR_SAMP(col3) AS VSCol3
      ,VAR_SAMP(col4) AS VSCol4
      ,VAR_SAMP(col5) AS VSCol5
      ,VAR_SAMP(col6) AS VSCol6
FROM Stats_Table;
```

VSCol1	VSCol2	VSCol3	VSCol4	VSCol5	VSCol6
78	20	198	78	20	748

The Variance function is a measure of dispersion (spread of the distribution) as the square of the standard deviation. There are two forms of Variance in Teradata, VAR_SAMP is used for a random sampling of the data rows allowed through by the WHERE clause.

The CORR Function

```
SELECT CORR(col1, col2) AS CCol1#2
FROM Stats_Table;
```

The CORR function is a binary function, meaning that two variables are used as input to it. It measures the association between 2 random variables. If the variables are such that, when one changes, the other does so in a related manner; they are correlated. Independent variables are not correlated because the change in one does not necessarily cause the other to change.

The correlation coefficient is a number between -1 and 1. It is calculated from a number of pairs of observations or linear points (X,Y).

Where:

1 = perfect positive correlation

0 = no correlation

-1 = perfect negative correlation

Syntax for using CORR:

CORR(<column-name>, <column-name>)

A CORR Example

Stats_Table

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT CORR(col2) AS CCol1#2
      ,CORR(col3) AS CCol1#3
      ,CORR(col4) AS CCol1#4
      ,CORR(col5) AS CCol1#5
      ,CORR(col6) AS CCol1#6
FROM Stats_Table;
```

CCol1#2	CCol1#3	CCol1#4	CCol1#5	CCol1#6
0.986480	0.885155	-1.000000	-0.151877	0.991612

Where:
1 = perfect positive correlation
0 = no correlation
-1 = perfect negative correlation

Another CORR Example so you can compare

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30

```
SELECT CORR(col2) AS CCol1#2
      ,CORR(col3) AS CCol1#3
      ,CORR(col1) AS CCol1#1
      ,CORR(col5) AS CCol1#5
      ,CORR(col6) AS CCol1#6
FROM Stats_Table;
```

9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

CCol1#2	CCol1#3	CCol1#4	CCol1#5	CCol1#6
-0.986480	-0.885155	-1.000000	0.151877	-0.991612

Where:
1 = perfect positive correlation
0 = no correlation
-1 = perfect negative correlation

The COVAR_POP Function

```
SELECT COVAR_POP(col1, col2) AS CCol1#2
FROM Stats_Table;
```

The covariance is a statistical measure of the tendency of two variables to change in conjunction with each other. It is equal to the product of their standard deviations and correlation coefficients.

The covariance is a statistic used for bivariate samples or bivariate distribution. It is used for working out the equations for regression lines and the product-moment correlation coefficient.

Syntax:

COVAR(<column-name>, <column-name>)

A COVAR_POP Example

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65

```
SELECT COVAR_POP(col2) AS CPCol1#2
      ,COVAR_POP(col3) AS CPCol1#3
      ,COVAR_POP(col4) AS CPCol1#4
      ,COVAR_POP(col5) AS CPCol1#5
      ,COVAR_POP(col6) AS CPCol1#6
FROM Stats_Table;
```

CPCol1#2	CPCol1#3	CPCol1#4	CPCol1#5	CPCol1#6
37.50	105.90	-74.92	-5.82	230.75

The covariance is a statistical measure of the tendency of two variables to change in conjunction with each other. It is equal to the product of their standard deviations and correlation coefficients.

23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

Another COVAR_POP Example so you can Compare

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT COVAR_POP(col2) AS CPCol1#2
      ,COVAR_POP(col3) AS CPCol1#3
      ,COVAR_POP(col1) AS CPCol1#1
      ,COVAR_POP(col5) AS CPCol1#5
      ,COVAR_POP(col6) AS CPCol1#6
FROM Stats_Table;
```

CPCol1#2	CPCol1#3	CPCol1#1	CPCol1#5	CPCol1#6
-37.50	-105.90	-74.92	5.82	-230.75

The covariance is a statistical measure of the tendency of two variables to change in conjunction with each other. It is equal to the product of their standard deviations and correlation coefficients.

The REGR_INTERCEPT Function

```
SELECT REGR_INTERCEPT(col1, col2) AS RIofCol1#2
FROM Stats_Table;
```

A regression line is a line of best fit, drawn through a set of points on a graph for X and Y coordinates. It uses the Y coordinate as the Dependent Variable and the X value as the Independent Variable.

Two regression lines always meet or intercept at the mean of the data points(x,y), where x=AVG(x) and y=AVG(y) and is not usually one of the original data points.

Syntax for using REGR_INTERCEPT:

REGR_INTERCEPT(dependent-expression, independent-expression)

A REGR_INTERCEPT Example

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT
 REGR_INTERCEPT(col2) AS RICol1#2
,REGR_INTERCEPT(col3) AS RICol1#3
,REGR_INTERCEPT(col4) AS RICol1#4
,REGR_INTERCEPT(col5) AS RICol1#5
,REGR_INTERCEPT(col6) AS RICol1#6
 FROM Stats_Table;
```

RICol1#2	RICol1#3	RICol1#4	RICol1#5	RICol1#6
-1	3	31	18	-1

A regression line is a line of best fit, drawn through a set of points on a graph for X and Y coordinates. It uses the Y coordinate as the Dependent Variable and the X value as the Independent Variable.

Two regression lines always meet or intercept at the mean of the data points(x,y), where x=AVG(x) and y=AVG(y) and is not usually one of the original data points.

Another REGR_INTERCEPT Example so you can compare

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95

```
SELECT
 REGR_INTERCEPT(col2) AS RICol1#2
,REGR_INTERCEPT(col3) AS RICol1#3
,REGR_INTERCEPT(col1) AS RICol1#1
,REGR_INTERCEPT(col5) AS RICol1#5
,REGR_INTERCEPT(col6) AS RICol1#6
 FROM Stats_Table;
```

RICol1#2	RICol1#3	RICol1#1	RICol1#5	RICol1#6
32	28	0	13	32

A regression line is a line of best fit, drawn through a set of points on a graph for X and Y coordinates. It uses the Y coordinate as the Dependent Variable and the X value as the Independent Variable.

Two regression lines always meet or intercept at the mean of the data points(x,y), where x=AVG(x) and y=AVG(y) and is not usually one of the original data points.

30	16	60	1	1	100

The REGR_SLOPE Function

```
SELECT REGR_SLOPE(col1, col2) AS RSCol1#2
FROM Stats_Table;
```

A regression line is a line of best fit, drawn through a set of points on a graph of X and Y coordinates. It uses the Y coordinate as the Dependent Variable and the X value as the Independent Variable.

The slope of the line is the angle at which it moves on the X and Y coordinates. The vertical slope is Y on X and the horizontal slope is X on Y.

Syntax for using REGR_SLOPE:

REGR_SLOPE(dependent-expression, independent-expression)

A REGR_SLOPE Example

Stats Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT
 REGR_SLOPE(col2) AS RSCol1#2
,REGR_SLOPE(col3) AS RSCol1#3
,REGR_SLOPE(col4) AS RSCol1#4
,REGR_SLOPE(col5) AS RSCol1#5
,REGR_SLOPE(col6) AS RSCol1#6
 FROM Stats_Table;
```

RSCol1#2	RSCol1#3	RSCol1#4	RSCol1#5	RSCol1#6
2	1	-1	-0	0

A regression line is a line of best fit, drawn through a set of points on a graph for X and Y coordinates. It uses the Y coordinate as the Dependent Variable and the X value as the Independent Variable.

The slope of the line is the angle at which it moves on the X and Y coordinates. The vertical slope is Y on X and the horizontal slope is X on Y.

Another REGR_SLOPE Example so you can compare

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20

```
SELECT
 REGR_SLOPE(col2) AS RSCol1#2
,REGR_SLOPE(col3) AS RSCol1#3
,REGR_SLOPE(col1) AS RSCol1#4
,REGR_SLOPE(col5) AS RSCol1#5
,REGR_SLOPE(col6) AS RSCol1#6
 FROM Stats_Table;
```

6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

RSCol1#2	RSCol1#3	RSCol1#1	RSCol1#5	RSCol1#6
-2	-1	1	0	-0

A regression line is a line of best fit, drawn through a set of points on a graph for X and Y coordinates. It uses the Y coordinate as the Dependent Variable and the X value as the Independent Variable.

The slope of the line is the angle at which it moves on the X and Y coordinates. The vertical slope is Y on X and the horizontal slope is X on Y.

Using GROUP BY

Stats_Table

Col1	Col2	Col3	Col4	Col5	Col6
1	1	1	30	1	0
2	1	1	29	2	5
3	3	10	28	3	10
4	3	10	27	4	15
5	3	10	26	5	20
6	4	10	25	6	30
7	5	10	24	7	30
8	5	10	23	8	30
9	5	10	22	9	35
10	5	20	21	10	35
11	7	20	20	22	40
12	7	20	19	12	40
13	9	20	18	13	45
14	9	20	17	14	45
15	9	20	16	15	50
16	9	20	15	14	55
17	10	20	14	13	55
18	10	20	13	12	60
19	10	20	12	11	60
20	10	20	11	9	65
21	10	20	10	8	65
22	10	20	9	7	65
23	13	20	8	6	70
24	13	30	7	5	70
25	13	30	6	4	80
26	14	40	5	3	85
27	15	40	4	2	90
28	15	50	3	1	90
29	16	50	2	1	95
30	16	60	1	1	100

```
SELECT col3
      ,count(*)          AS Cnt
      ,avg(col1)         AS Avg1
      ,stddev_pop(col1)  AS SD1
      ,var_pop(col1)     AS VP1
      ,avg(col4)         AS Avg4
      ,stddev_pop(col4)  AS SD4
      ,var_pop(col4)     AS VP4
      ,avg(col6)         AS Avg6
      ,stddev_pop(col6)  AS SD6
      ,var_pop(col6)     AS VP6
FROM Stats_Table
GROUP BY 1 ORDER BY 1;
```

Col3	Cnt	Avg1	SD1	VP1	Avg4	SD4	VP4	Avg6	SD6	VP6
1	2	2	0	0	30	0	0	2	2	6
10	7	6	2	4	25	2	4	24	9	74
20	14	16	4	16	14	4	16	54	11	116
30	2	24	0	0	6	0	0	75	5	25
40	2	26	0	0	4	0	0	88	2	6
50	2	28	0	0	2	0	0	92	2	6
60	1	30	0	0	1	0	0	100	0	0

No Having Clause vs. Use of HAVING

```
SELECT    col3
          ,count(*)              AS Cnt
          ,avg(col1)             AS Avg1
          ,stddev_pop(col1)      AS SD1
          ,var_pop(col1)         AS VP1
          ,avg(col4)             AS Avg4
          ,stddev_pop(col4)      AS SD4
          ,var_pop(col4)         AS VP4
          ,avg(col6)             AS Avg6
          ,stddev_pop(col6)      AS SD6
          ,var_pop(col6)         AS VP6
FROM  Stats_Table
GROUP BY 1 ORDER BY 1 ;
```

Col3	Cnt	Avg1	SD1	VP1	Avg4	SD4	VP4	Avg6	SD6	VP6
1	2	2	0	0	30	0	0	2	2	6
10	7	6	2	4	25	2	4	24	9	74
20	14	16	4	16	14	4	16	54	11	116
30	2	24	0	0	6	0	0	75	5	25
40	2	26	0	0	4	0	0	88	2	6
50	2	28	0	0	2	0	0	92	2	6
60	1	30	0	0	1	0	0	100	0	0

```
SELECT col3
          ,count(*)              AS Cnt
          ,avg(col1)             AS Avg1
          ,stddev_pop(col1)      AS SD1
          ,var_pop(col1)         AS VP1
FROM  Stats_Table
GROUP BY 1
ORDER BY 1
HAVING Cnt > 2 and VP1 < 20 ;
```

col3	Cnt	Avg1	SD1	VP1
10	7	6	2	4
20	14	16	4	16

The example above uses HAVING to perform a compound comparison on both the count and the covariance.

Stored Procedure Functions

Stored Procedure Functions

Overview

"Freedom from effort in the present merely means that there has been effort stored up in the past."
- Theodore Roosevelt

Stored Procedures vs. Macros

Macros	Stored Procedures
▪ Contains SQL ▪ May contain BTEQ Dot commands ▪ Parameter values can be passed ▪ May retrieve 1 or more rows ▪ Stored in DBC PERM Space ▪ Returns rows to the client	▪ Contains SQL ▪ Contains comprehensive SPL ▪ Parameter values can be passed to it ▪ Must use a cursor to retrieve > than 1 row ▪ Stored in DATABASE or USER PERM ▪ May return 1 or more values to client as parameter

Stored Procedures are a lot like Macros. However, they manipulate data a row at a time. Stored Procedures take up PERM Space, unlike Views and Macros that do NOT. Stored Procedures are actually compiled and will use a Cursor to retrieve or manipulate more than one row. Although Stored Procedures utilize SQL, they also utilize SPL, which stands for Stored Procedure Language, which provides loops, while, etc.

Creating a Stored Procedure

THE BEGIN and END statements are required in all Stored Procedures. Don't miss a semi-colon. They are everywhere. I have bolded them for your convenience.

How you CALL a Stored Procedure

1
```
CREATE PROCEDURE First_Procedure ( )
BEGIN
   INSERT INTO Customer_Table DEFAULT VALUES;
END;
```

2
```
CALL First_Procedure ( ) ;
```
Parentheses Needed

3
```
SELECT * FROM Customer_Table ORDER BY 1 ;
```

Customer_Number	Customer_Name	Phone_Number
?	?	?
11111111	Billy's Best Choice	555-1234
31313131	Acme Products	555-1111
31323134	Ace Consulting	555-1212
57896883	XYZ Plumbing	347-8954
87323456	Databases N-U	322-1012

Default Values row placed inside Table

You SELECT from a View, EXECUTE a Macro, and you CALL a Stored Procedure.

Label all BEGIN and END statements except the first ones

1
```
CREATE PROCEDURE Second_Procedure ( )
BEGIN
   INSERT INTO Customer_Table DEFAULT VALUES;
   SecondSection:BEGIN
   DELETE FROM Customer_Table WHERE Customer_Number is NULL;
    END SecondSection;
END;
```

2
```
CALL Second_Procedure ( ) ;
```

When you have multiple BEGIN and END statements, you have to label them all (except for the first BEGIN and END statements). We have labeled our next set of BEGIN and END SecondSection.

How to Declare a Variable

1

```
CREATE PROCEDURE Declare_Procedure ( )
 BEGIN
 DECLARE  var1   INTEGER   DEFAULT  11111111;
   DELETE FROM Customer_Table WHERE Customer_Number = :var1 ;
 END;
```

Colon

2

```
CALL Declare_Procedure ( ) ;
```

When you DECLARE a variable, and then reference that variable later, a colon is always in front of the Variable.

How to Declare a Variable and then SET the Variable

1

```
CREATE PROCEDURE SetVar_Procedure ( )
 BEGIN
   DECLARE  var1   INTEGER ;
 SET var1 = 31313131 ;
   DELETE FROM  Customer_Table WHERE Customer_Number = :var1 ;
 END ;
```

Semi-Colon

Colon

2

```
CALL SetVar_Procedure ( ) ;
```

Once a variable and the data type is defined, the value must be assigned. SET is the more flexible a method compared to DEFAULT.

An IN Variable is passed to the Procedure during the CALL

1

```
CREATE PROCEDURE PassInput_Procedure (IN  var1  INTEGER )
 BEGIN
   DELETE FROM  Customer_Table WHERE Customer_Number = :var1;
 END;
```

2

```
CALL PassInput_Procedure (31323134 ) ;
```

The Variable Var1 was not assigned with the DEFAULT or the SET, but instead passed as a parameter. There are three types of parameters (IN, OUT, INOUT). In this example, an IN is being used. Warning: You cannot add, subtract, or change an IN variable. You set it when you call the procedure and that value remains constant.

The IN, OUT and INOUT Parameters

1

```
CREATE PROCEDURE Test_Proc
        (IN  var1 BYTEINT, IN var2 BYTEINT, OUT Msg CHAR(20) )
BEGIN
     CASE  WHEN var1 = var2   THEN  Set Msg = 'They are equal' ;
           WHEN var1 < var2   THEN  Set Msg = 'Variable 1 less' ;
           ELSE                     Set Msg =  'Variable 1 greater' ;
      END CASE;
END;
```

2

```
CALL Test_Proc (1,2, Msg ) ;
```

Msg

There are three types of parameters (IN, OUT, INOUT). This is an example of an IN and an OUT parameter. What that means is this Stored Procedure will take a parameter in, and then spit something out. Notice that we named the OUT parameter Msg, and then we needed to put the name Msg in our Call statement.

Using IF inside a Stored Procedure

1

```
CREATE PROCEDURE TestIF_Proc
(IN var1 BYTEINT, INvar2 BYTEINT, OUT Msg CHAR(20) )
 BEGIN
  IF var1 = var2 THEN SET Msg = 'They are equal';
  END IF ;
   IF var1 < var2 THEN SET Msg = 'Variable 1 less';
   END IF ;
    IF var1 > var2 THEN SET Msg = 'Variable 1 greater';
    END IF ;
END;
```

```
      ELSE
SET Msg = 'Variable 1 greater';
   END IF;
END;
```

```
   IF var1 < var2 THEN
   SET Msg = 'Variable 1 less';
   END IF;
    IF var1 > var2 THEN
    SET Msg = 'Variable 1 greater';
   END IF;
END;
```

These queries do the SAME thing. However, the first one is more efficient because it only does TWO calculations instead of three.

Using Loops in Stored Procedures

1

```
CREATE Table My_Log_Tbl
(
 Cntr       Integer
,TheTime  Time
) Primary Index (Cntr);
```

2

```
CREATE PROCEDURE Inserter_Five ( )
LOOPER:BEGIN
  DECLARE Cntr INTEGER DEFAULT 0;
Loopit:LOOP
    SET Cntr = Cntr + 1;
     IF Cntr > 5 THEN  LEAVE Loopit;
     END IF;
    INSERT INTO My_Log_Tbl
                VALUES (:Cntr, TIME) ;
  END LOOP Loopit ;
END LOOPER ;
```

3

```
CALL Inserter_Five () ;
```

4

```
SELECT * FROM My_Log_Tbl
ORDER BY 1 ;
```

Cntr	TheTime
1	09:48:43
2	09:48:43
3	09:48:43
4	09:48:43

LOOPs require Labeling. Much like when you have more than one BEGIN/END.

You can Name the First Begin and End if you choose

```
CREATE Table My_Log_Tbl
(
 Cntr       Integer
,TheTime  Time
) Primary Index (Cntr);
```

1 First we have to CREATE the Table.

```
CREATE PROCEDURE Inserter_Five ( )
  LOOPER:BEGIN
    DECLARE Cntr INTEGER          DEFAULT 0;
    Loopit:LOOP
       SET Cntr = Cntr + 1;
       IF Cntr > 5 THEN  LEAVE Loopit;
       END IF;
       INSERT INTO My_Log_Tbl
       VALUES (:Cntr, TIME);
     END LOOP Loopit ;
  END LOOPER;
```

2 Then we build the Stored Procedure

3 Now we call the Stored Procedure

```
CALL Inserter_Five () ;
```

4

```
SELECT *
FROM My_Log_Tbl
ORDER BY 1 ;
```

Cntr	TheTime
1	09:48:43
2	09:48:43
3	09:48:43
4	09:48:43
5	09:48:43

This loops 5 times! We didn't have to label Looper because it's the first Begin and End. The LEAVE statement is how the LOOP is told it is done looping.

Using Keywords LEAVE vs. UNTIL for LEAVE vs. REPEAT

```
--Procedure One
CREATE PROCEDURE Ins5 ( )
LOOPER: BEGIN
DECLARE Cntr INTEGER
         DEFAULT 0;
 Loopit:LOOP
  SET Cntr = Cntr + 1;
   IF Cntr > 5 THEN LEAVE Loopit;
   END IF;
   INSERT INTO My_Log_Tbl
      VALUES (:Cntr, TIME);
 END LOOP Loopit ;
END LOOPER;
```

```
--Procedure Two
CREATE PROCEDURE Ins5A ( )
LOOPER:BEGIN
   DECLARE Cntr INTEGER
                DEFAULT 0;
  Loopit:REPEAT
    SET Cntr = Cntr + 1;
     INSERT INTO My_Log_Tbl
     VALUES (:Cntr, TIME);
    UNTIL Cntr > 4
   END REPEAT Loopit ;
END LOOPER;
```

Both Procedures above do the same thing. The UNTIL keyword in Procedure Two jumps it out of the REPEAT Loop once it reaches the Cntr, and the procedure moves on. There are some differences in the above. The first example (Procedure One), tests Cntr before the INSERT. But Procedure two does not, so Procedure two will always do at least one INSERT, no matter what Cntr is set at.

Stored Procedure Basic Assignment

Stored Procedure Basic Assignment

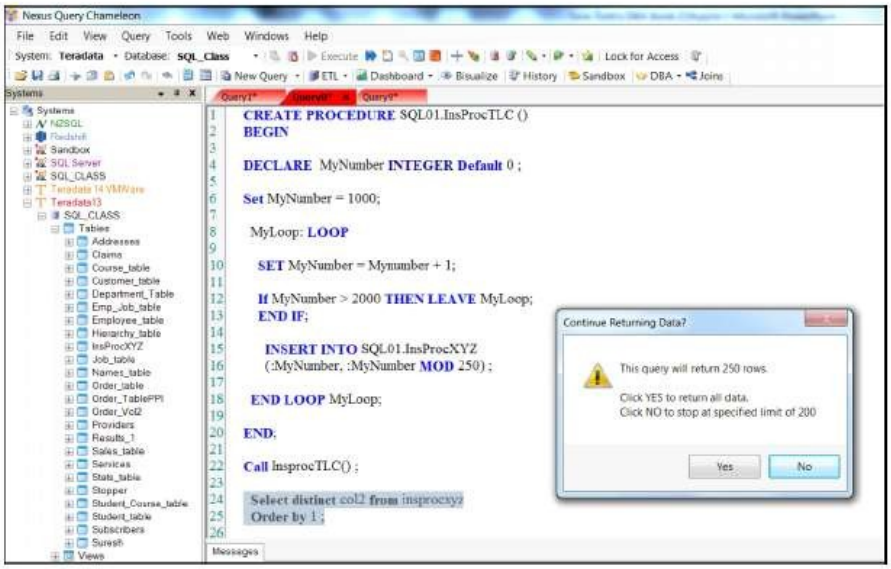
Create the table below and substitute the XYZ with your initials.

```
CREATE MULTISET Table SQL01.InsProcXYZ
( Col1 INTEGER
,Col2 INTEGER
) Primary Index (Col1) ;
```

Now, create a stored procedure called InsertXYZ that places 1,000 rows inside the table. Col1 should have 1000 unique values, and Col2 should have 250 different values.

Turn the page if you need some help.

Answer - Stored Procedure Basic Assignment



Stored Procedure Advanced Assignment

Create both tables below and substitute the XYZ with your initials.

```
CREATE MULTISET Table
SQL01.InsProc2XYZ
( Col1 INTEGER
 ,Col2 INTEGER
) Primary Index (Col1) ;
```

```
CREATE MULTISET Table
SQL01.InsProc3XYZ
( Col1 INTEGER
,Col2 INTEGER
) Primary Index (Col1) ;
```

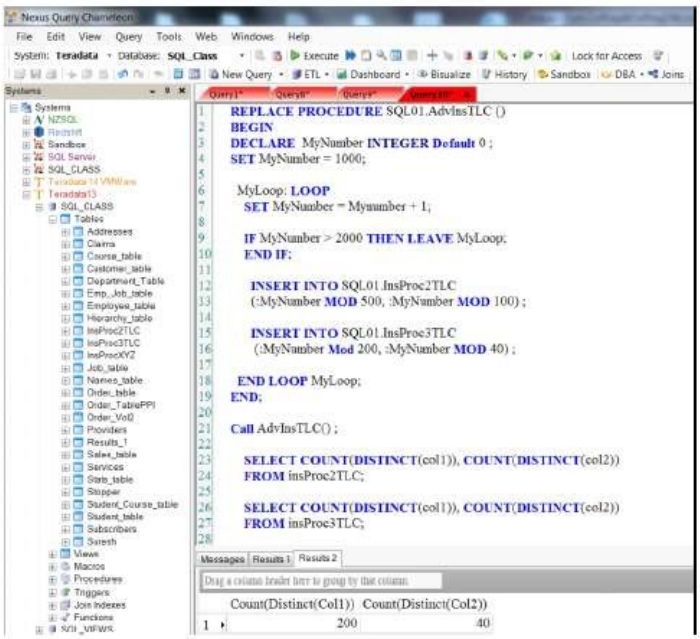
Then, create a stored procedure called AdvInsXYZ that places 1,000 rows inside both tables.

In InsProc2XYZ, the column Col1 should have 500 different values, and Col2 should have 100 different values.

In InsProc3XYZ, the column Col1 should have 200 different values, and Col2 should have 40 different values.

Turn the page if you need some help.

Answer - Stored Advanced Assignment


Nexus Query Chameleon
File Edit View Query Tools Web Windows Help
System: Teradata Database: SQL_Class
Execute
Lock for Access
New Query
ETL
Dashboard
Bisualize
History
Sandbox
DBA
Joins
Systems
Query1*
Query8*
Query9*
Query10*
1 REPLACE PROCEDURE SQL01.AdvInsTLC ()
2 BEGIN
3 DECLARE MyNumber INTEGER Default 0 ;
4 SET MyNumber = 1000;
5
6 MyLoop: LOOP
7 SET MyNumber = Mynumber + 1;
8
9 IF MyNumber > 2000 THEN LEAVE MyLoop;
10 END IF;
11
12 INSERT INTO SQL01.InsProc2TLC
13 (:MyNumber MOD 500, :MyNumber MOD 100) ;
14
15 INSERT INTO SQL01.InsProc3TLC
16 (:MyNumber Mod 200, :MyNumber MOD 40) ;
17
18 END LOOP MyLoop;
19 END;
20
21 Call AdvInsTLC() ;
22
23 SELECT COUNT(DISTINCT(col1)), COUNT(DISTINCT(col2))
24 FROM insProc2TLC;
25
26 SELECT COUNT(DISTINCT(col1)), COUNT(DISTINCT(col2))
27 FROM insProc3TLC;
28
Messages
Results 1
Results 2
Drag a column header here to group by that column.
Count(Distinct(Col1)) Count(Distinct(Col2))
1 200 40

Sub-query Functions

Sub-query Functions

Overview

"A little man often casts a long shadow."
- Italian Proverb

An IN List is much like a Subquery

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

```
SELECT *
FROM Employee_Table
WHERE Dept_No IN(100, 200) ;
```

Employee_No	Dept_No	Last_Name	First_Name	Salary
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00

Answer Set

This query is very simple and easy to understand. It uses an IN List to find all Employees who are in Dept_No 100 or Dept_No 200.

An IN List Never has Duplicates - Just like a Subquery

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

```
SELECT *
FROM Employee_Table
WHERE Dept_No IN (100, 100,200, 200) ;
```

What is going on with this IN List? Why in the world are their duplicates in there? Will this query even work? What will the result set look like? Turn the page!

An IN List Ignores Duplicates

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

```
SELECT *
FROM Employee_Table
WHERE Dept_No IN (100, 100,200, 200) ;
```

Employee_No	Dept_No	Last_Name	First_Name	Salary
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00

Answer Set

Duplicate values are ignored here. We get the same rows back as before, and it is as if the system ignored the duplicate values in the IN List. That is exactly what happened.

The Subquery

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary
1232578	100	Chambers	Mandee	48850.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
2312225	300	Larkins	Loraine	40200.00
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1121334	400	Strickling	Cletus	54500.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00

Department_Table

Dept_No	Department_Name
100	Marketing
200	Research and Dev
300	Sales
400	Customer Support
500	Human Resources

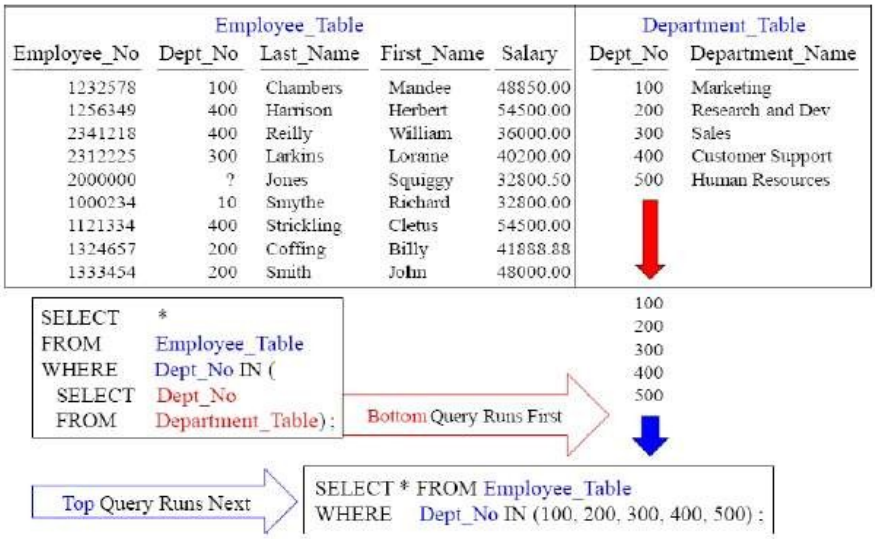
There is a Top Query and a Bottom Query!

```
SELECT *
FROM Employee_Table
WHERE    Dept_No IN (
  SELECT   Dept_No
  FROM     Department_Table) ;
```

Which Query Runs First?

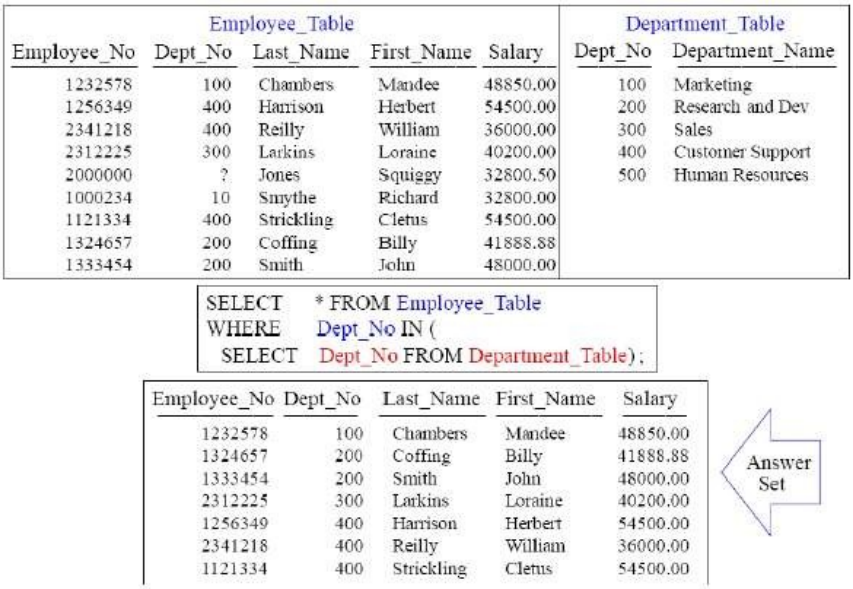
The query above is a Subquery, which means there are multiple queries in the same SQL. The bottom query runs first, and its purpose in life is to build a distinct list of values that it passes to the top query. The top query then returns the result set. This query solves the problem: Show all Employees in Valid Departments!

How a Basic Subquery Works

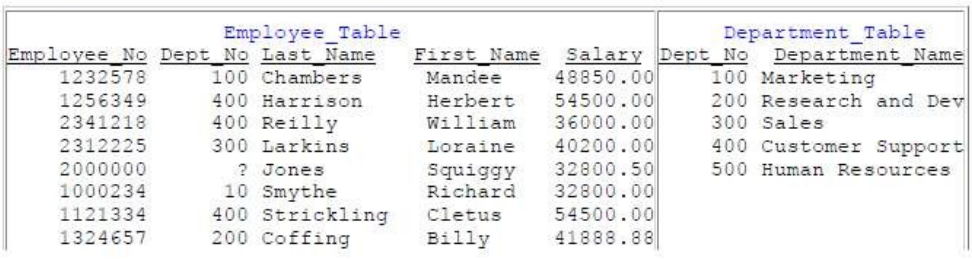


The bottom query runs first, and builds a distinct IN list. Then, the top query runs using the list.

The Final Answer Set from the Subquery



Quiz- Answer the Difficult Question



```
1333454     200 Smith          John        48000.00
```

How are Subqueries similar to Joins between two tables?

A great question was asked above. Do you know the key to answering? Turn the page!

Answer to Quiz- Answer the Difficult Question

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary
1232578	100	Chambers	Mandee	48850.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
2312225	300	Larkins	Loraine	40200.00
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1121334	400	Strickling	Cletus	54500.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00

Department_Table

Dept_No	Department_Name
100	Marketing
200	Research and Dev
300	Sales
400	Customer Support
500	Human Resources

Primary Key

How are Subqueries similar to Joins between two tables?

A Subquery between two tables or a Join between two tables will each need a common key that represents the relationship.
This is called a Primary Key/Foreign Key relationship.
Just like Dept_No and Dept_No!

A Subquery will use a common key linking the two tables together; very similar to a join! When subquerying between two tables, look for the common link between the two tables. They will commonly both have a column with the same name, but not always.

Should you use a Subquery of a Join?

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary
1232578	100	Chambers	Mandee	48850.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
2312225	300	Larkins	Loraine	40200.00
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1121334	400	Strickling	Cletus	54500.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00

Department_Table

Dept_No	Department_Name
100	Marketing
200	Research and Dev
300	Sales
400	Customer Support
500	Human Resources

When do I Subquery?

```
SELECT *
FROM Employee_Table
WHERE     Dept_No IN (
 SELECT   Dept_No
 FROM     Department_Table) ;
```

When do I perform a Join?

```
SELECT E.*
FROM Employee_Table as E
Inner Join
        Department_Table as D
ON E.Dept_No= D.Dept_No;
```

Both queries above return the same data. If you only want to see a report where the final result set has only columns from one table, try a Subquery. Obviously, if you need columns on the report where the final result set has columns from both tables, you have to do a Join.

Quiz- Write the Subquery

Customer_Table	Order_Table

Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91
31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

Write the Subquery

Select all columns in the Customer_Table if the customer has placed an order!

Here is your opportunity to show how smart you are. Write a Subquery that will bring back everything from the Customer_Table if the customer has placed an order in the Order_Table. Good luck! Advice: Look for the common key among both tables!

Answer to Quiz- Write the Subquery

Customer_Table		Order_Table		
Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91
31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

Write the Subquery

Select all columns in the Customer_Table if the customer has placed an order!

```
SELECT *
FROM Customer_Table
WHERE      Customer_Number IN(
 SELECT    Customer_Number
 FROM      Order_Table) ;
```

Customer_Number	Customer_Name
31323134	ACE Consulting
57896883	XYZ Plumbing
11111111	Billy's Best Choice
87323456	Databases N-U

The common key among both tables is Customer_Number. The bottom query runs first and delivers a distinct list of Customer_Numbers, which the top query uses in the IN List!

Quiz- Write the More Difficult Subquery

Customer_Table		Order_Table		
Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91
31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

Write the Subquery

Select all columns in the Customer_Table if the customer has placed an order over $10,000.00 Dollars!

Here is your opportunity to show how smart you are. Write a Subquery that will bring back everything from the Customer_Table if the customer has placed an order in the Order_Table that is greater than $10,000.00.

Answer to Quiz- Write the More Difficult Subquery

Customer_Table		Order_Table		
Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91

31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

Write the Subquery

Select all columns in the Customer_Table if the customer has placed an order over $10,000.00 Dollars!

```
SELECT *
FROM Customer_Table
WHERE      Customer_Number IN (
 SELECT    Customer_Number
 FROM      Order_Table
 WHERE Order_Total > 10000.00) ;
```

Customer_Number	Customer_Name
11111111	Billy's Best Choice
57896883	XYZ Plumbing
87323456	Databases N-U

Here is your answer!

Quiz- Write the Subquery with an Aggregate

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Write the Subquery

Select all columns in the Employee_Table if the employee makes a greater Salary than the AVERAGE Salary.

Another opportunity knocking! Would someone please answer the query door!?

Answer to Quiz- Write the Subquery with an Aggregate

Employee_Table

Employee_No	Dep_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Select all columns in the Employee_Table if the employee makes a greater Salary than the AVERAGE Salary.

```
SELECT * FROM Employee_Table
WHERE      Salary >(
 SELECT     AVG(Salary)
 FROM       Employee_Table) ;
```

Quiz- Write the Correlated Subquery

Employee_Table

Employee_No	Dept_No	Last_Name	First_Name	Salary

2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Write the Correlated Subquery

Select all columns in the Employee_Table if the employee makes a greater Salary than the AVERAGE Salary (within their own Department).

Another opportunity knocking! This is a tough one and only the best get this written correct.

Answer to Quiz- Write the Correlated Subquery

Employee_Table

Employee_No	Dep_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Select all columns in the Employee_Table if the employee makes a greater Salary than the AVERAGE Salary (within their own Department).

```
SELECT * FROM Employee_Table as EE
WHERE     Salary > (
 SELECT   AVG(Salary)
 FROM     Employee_Table as EEEE
  WHERE   EE.Dept_No = EEEE.Dept_No) ;
```

The Basics of a Correlated Subquery

The Top Query is Co-Related (Correlated) with the Bottom Query.

The same table is used twice, but given a different alias both times.

The bottom WHERE clause co-relates Dept_No from Top and Bottom.

```
SELECT *
FROM Employee_Table as EE
WHERE     Salary > (
 SELECT   AVG(Salary)
 FROM     Employee_Table as EEEE
 WHERE    EE.Dept_No = EEEE.Dept_No) ;
```

Does the Top or Bottom Query run first?

The Top Query always runs first in a Correlated Subquery

```
SELECT *
FROM Employee_Table as EE
WHERE     Salary > (
```

```
SELECT  AVG(Salary)
FROM    Employee_Table as EEEE
 WHERE  EE.Dept_No = EEEE.Dept_No) ;
```

The Top Query always runs first in a Correlated Subquery?

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Results only after the TOP Query has run.

This is NOT the Final ANSWER Set.

The Bottom Query runs last in a Correlated Subquery

```
SELECT * FROM Employee_Table as EE
WHERE     Salary > (
 SELECT   AVG(Salary)
 FROM     Employee_Table as EEEE
  WHERE   EE.Dept_No = EEEE.Dept_No) ;
```

The Top Query always runs 1st in a Correlated Subquery?

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Dept_No	AVG(Salary)
?	32800.50
10	32800.00
100	48850.00
200	44944.44
300	40200.00
400	48333.33

The Bottom Query run 2nd to get the Average Salary once for each distinct Dept_No!

Quiz- Who is coming back in the Final Answer Set?

```
SELECT * FROM Employee_Table as EE
WHERE      Salary > (
 SELECT    AVG(Salary)
 FROM      Employee_Table as EEEE
  WHERE    EE.Dept_No = EEEE.Dept_No) ;
```

The Top Query always runs 1st in a Correlated Subquery?

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Dept_No	AVG(Salary)
?	32800.50
10	32800.00
100	48850.00
200	44944.44
300	40200.00
400	48333.33

Which Employees will be in the Final Answer Set?

Look at the results from the TOP Query on Left and then look at how the Bottom Query is run once per Dept_No. Figure out (in your head) what is coming back.

Answer- Who is coming back in the Final Answer Set?

```
SELECT * FROM Employee_Table as EE
WHERE      Salary > (
 SELECT    AVG(Salary)
 FROM      Employee_Table as EEEE
  WHERE EE.Dept_No = EEEE.Dept_No) ;
```

Employee_No	Dept_No	Last_Name	First_Name	Salary
2000000	?	Jones	Squiggy	32800.50
1000234	10	Smythe	Richard	32800.00
1232578	100	Chambers	Mandee	48850.00
1324657	200	Coffing	Billy	41888.88
1333454	200	Smith	John	48000.00
2312225	300	Larkins	Loraine	40200.00
1256349	400	Harrison	Herbert	54500.00
2341218	400	Reilly	William	36000.00
1121334	400	Strickling	Cletus	54500.00

Dept_No	AVG(Salary)
?	32800.50
10	32800.00
100	48850.00
200	44944.44
300	40200.00
400	48333.33

Employee_No	Dept_No	Last_Name	First_Name	Salary
1333454	200	Smith	John	48000.00
1256349	400	Harrison	Herbert	54500.00
1121334	400	Strickling	Cletus	54500.00

Answer Set

Correlated Subquery Example vs. a Join with a Derived Table

```
SELECT Last_Name, Dept_No, Salary
FROM Employee_Table as EE
WHERE    Salary > (
 SELECT  AVG(Salary)
 FROM    Employee_Table as EEEE
 WHERE EE.Dept_No = EEEE.Dept_No) ;
```

Correlated Subquery

Last_Name	Dept_No	Salary
Smith	200	48000.00
Harrison	400	54500.00
Strickling	400	54500.00

```
SELECT E.*, AVGSAL
FROM Employee_Table as E
INNER JOIN
```

Join with a Derived Table

```
  (SELECT Dept_No, AVG(Salary)
    FROM   Employee_Table
     GROUP BY Dept_No)
    as TeraTom (Depty, AVGSAL)
ON   Dept_No = Depty
AND Salary > AVGSAL ;
```

Last_Name	Dept_No	Salary	AVGSAL
Smith	200	48000.00	44944.44
Harrison	400	54500.00	48333.33
Strickling	400	54500.00	48333.33

Both queries above will bring back all employees making a salary that is greater than the average salary in their department. The biggest difference is that the Join with the Derived Table also shows the Average Salary in the result set.

Quiz- A Second Chance to Write a Correlated Subquery

All Rows are NOT Displayed

Product_ID	Sale_Date	Daily_Sales
1000	10/02/2000	32800.50
1000	09/30/2000	36000.07
1000	10/01/2000	40200.43
2000	10/04/2000	32800.50
2000	10/02/2000	36021.93
2000	09/28/2000	41888.88
3000	10/04/2000	15675.33
3000	10/02/2000	19678.94
3000	10/03/2000	21553.79

Write the Correlated Subquery

Select all columns in the Sales_Table if the Daily_Sales column is greater than the Average Daily_Sales within its own Product_ID.

Another opportunity knocking! This is your second chance. I will even give you a third chance.

Answer - A Second Chance to Write a Correlated Subquery

Select all columns in the Sales_Table if the Daily_Sales column is greater
than the Average Daily_Sales within its own Product_ID.

```
SELECT * FROM Sales_Table as TopS
WHERE      Daily_Salary > (
 SELECT    AVG(Daily_Sales)
 FROM      Sales_Table as BotS
 WHERE     TopS.Product_ID= BotS.Product_ID)
ORDER BY   Product_ID, Sale_Date ;
```

Answer Set

Product_ID	Sale_Date	Daily_Sales
1000	09/28/2000	48850.40
1000	09/29/2000	54500.22
1000	10/03/2000	64300.00
1000	10/04/2000	54553.10
2000	09/29/2000	48000.00
2000	09 30 2000	49850.03
2000	10/01/2000	54850.29
3000	09/28/2000	61301.77
3000	09/29/2000	34509.13
3000	09/30/2000	43868.86

Quiz- A Third Chance to Write a Correlated Subquery

Product_ID	Sale_Date	Daily_Sales
1000	10/02/2000	32800.50
1000	09/30/2000	36000.07

All Rows are NOT Displayed

1000	10/01/2000	40200.43
2000	10/04/2000	32800.50
2000	10/02/2000	36021.93
2000	09/28/2000	41888.88
3000	10/04/2000	15675.33
3000	10/02/2000	19678.94
3000	10/03/2000	21553.79

Write the Correlated Subquery

Select all columns in the Sales_Table if the Daily_Sales column is greater than the Average Daily_sales within its own Sale_Date.

Another opportunity knocking! Just one minor adjustment and you are home free.

Answer - A Third Chance to Write a Correlated Subquery

Select all columns in the Sales_Table if the Daily_Sales column is greater than the Average Daily_Sales within its own Product_ID.

```
SELECT * FROM Sales_Table as TopS
WHERE     Daily_Salary > (
 SELECT AVG(Daily_Sales)
 FROM Sales_Table as BotS
  WHERE TopS.Sale_Date = BotS.Sale_Date)
ORDER BY Sale_Date ;
```

Answer Set

Product_ID	Sale_Date	Daily_Sales
3000	09/28/2000	61301.77
2000	09/29/2000	48000.00
1000	09/29/2000	54500.22
3000	09/30/2000	43868.86
2000	09/30/2000	49850.03
2000	10/01/2000	54850.29
2000	10/02/2000	36021.93
1000	10/02/2000	32800.50
2000	10/03/2000	43200.18
1000	10/03/2000	64300.00
1000	10/04/2000	54553.10

Quiz- Last Chance to Write a Correlated Subquery

Quiz-Last Chance To Write a Correlated Subquery

Student_Table

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
423400	Larkins	Michael	FR	0.00
125634	Hanson	Henry	FR	2.88
280023	McRoberts	Richard	JR	1.90
260000	Johnson	Stanley	?	?
231222	Wilson	Susie	SO	3.80
234121	Thomas	Wendy	FR	4.00
324652	Delaney	Danny	SR	3.35
123250	Phillips	Martin	SR	3.00
322133	Bond	Jimmy	JR	3.95
333450	Smith	Andy	SO	2.00

Write the Correlated Subquery

Select all columns in the Student_Table if the Grade_Pt column is greater than the Average Grade_Pt within its own Class_Code.

Another opportunity knocking! Just one minor adjustment and you are home free.

Answer - Last Chance to Write a Correlated Subquery

Select all columns in the Sales_Table if the Daily_Sales column is greater than the Average Daily_Sales within its own Product_ID.

```
SELECT* FROM Student_Table as TopS
WHERE    Grade_Pt > (
 SELECT   AVG(Grade_Pt)
  FROM     Student_Table as BotS
   WHERE   TopS. Class_Code = BotS.Class_Code )
ORDER BY Class_Code;
```

Answer Set

Student_ID	Last_Name	First_Name	Class_Code	Grade_Pt
234121	Thomas	Wendy	FR	4.00
125634	Hanson	Henry	FR	2.88
322133	Bond	Jimmy	JR	3.95
231222	Wilson	Susie	SO	3.80
324652	Delaney	Danny	SR	3.35

Correlated Subquery that Finds Duplicates

This is just a general example, because the Claim_Pay_Table does not exist in our database.

```
SELECT    Provider_Name, Claim_Id, Subscriber_Name
          ,Member_Name, Claim_Payment
FROM      Claims_Table clm
INNER JOIN  Subscriber_Table sub
ON          clm.Subscriber_Id = sub.Subscriber_Id
AND         clm.Member_No = sub.Member_No
INNER JOIN   Claim_Pay_Table  cpt
ON            cpt.Claim_Id=clm.Claim_Id
WHERE 1 < (select count(*) from Claim_Table
           where clm.Claim_Id=Claim_Id
            and clm.subscriber_no=Subscriber_No
             and clm.Member_No=Member_No
              and clm.Provider_No=Provider_No);
```

There is another challenge faced by many people today, and it relates to improper controls being used in the front-end systems and human error. Specifically, duplicate records can be a problem in the data, as well as in the functioning of the organization. For instance, a physician may bill the healthcare insurance company twice and they might even pay it twice. Either way, this activity should exist in the data warehouse. The following query can be used to find this erroneous type of data or occurrence.

Quiz- Write the NOT Subquery

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Write the Subquery

Select all columns in the Customer_Table if the Customer has NOT placed an order.

Another opportunity knocking! Write the above query!

Answer to Quiz- Write the NOT Subquery

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Select all columns in the Customer_Table if the Customer has NOT placed an order.

```
SELECT *
FROM  Customer_Table
WHERE Customer_Number
                  NOT IN
(SELECT Customer_Number
 FROM Order_Table
 WHERE Customer_Number
        IS NOT NULL) ;
```

```
SELECT *
FROM  Customer_Table
WHERE Customer_Number
                  NOT = ALL
(SELECT Customer_Number
 FROM Order_Table
 WHERE Customer_Number
        IS NOT NULL) ;
```

Wow! You can see that both queries are the same with just a few different techniques.

Quiz- Write the Subquery using a WHERE Clause

Quiz-Write the Subquery using a WHERE Clause

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Write the Subquery

Select all columns in the Order_Table that were placed by a customer with 'Bill' anywhere in their name.

Another opportunity to show your brilliance is ready for you to make it happen.

Answer - Write the Subquery using a WHERE Clause

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Write the Subquery

Select all columns in the Order_Table that were placed by a customer with 'Bill' anywhere in their name.

```
SELECT * FROM     Order_Table
```

```
WHERE Customer_Number IN
(SELECT Customer_Number FROM Customer_Table
   WHERE Customer_Name LIKE '%Bill%') ;
```

Great job on writing your query! This is a difficult task, but a rewarding one once you understand how powerful correlated Subqueries can be.

Quiz- Write the Subquery with Two Parameters

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Write the Subquery

What is the highest dollar order for each Customer?
This Subquery will involve two parameters!

Get ready to be amazed at either yourself or the answer on the next page!

Answer to Quiz- Write the Subquery with Two Parameters

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Write the Subquery

What is the highest dollar order for each Customer?
This Subquery will involve two parameters!

```
SELECT Customer_Number, Order_Number, Order_Total
FROM Order_Table
WHERE   (Customer_Number,      Order_Total) IN
(SELECT Customer_Number, MAX(Order_Total)
 FROM   Order_Table GROUP BY 1) ;
```

This is how you utilize multiple parameters in a Subquery! Turn the page for more.

How the Double Parameter Subquery Works

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

```
SELECT Customer_Number, Order_Number, Order_Total
FROM Order_Table
WHERE   (Customer_Number,      Order_Total) IN
```

```
(SELECT Customer_Number, MAX(Order_Total)
 FROM   Order_Table GROUP BY 1) ;
```

Customer_Number	Max(Order_Total)
11111111	12347.53
31323134	5111.47
87323456	15231.62
57896883	23454.84

These 4 rows are sent to the top query

The bottom query runs first, returning two columns. Next page for more info!

More on how the Double Parameter Subquery Works

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

```
SELECT Customer_Number, Order_Number, Order_Total
FROM   Order_Table
WHERE (Customer_Number, Order_Total ) IN
      ( 11111111   ,12347.53
        31323134   , 5111.47
        87323456   ,15231.62
        57896883   ,23454.84 );
```

The top query now uses the In-list

The IN list is built. The top query can now process for the final Answer Set.

Quiz - Write the Triple Subquery

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Write the Subquery

What is the Customer_Name who has the highest dollar order among all customers? This query will have multiple Subqueries!

Good luck in writing this…Remember that this will involve multiple Subqueries.

Answer to Quiz - Write the Triple Subquery

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Write the Subquery

What is the Customer_Name who has the highest dollar order among all customers? This query will have multiple Subqueries!

```
SELECT Customer_Name
FROM   Customer_Table
WHERE    Customer_Number IN
(SELECT  Customer_Number FROM Order_Table
 WHERE    Order_Total IN
 (SELECT  Max(Order_Total) FROM Order_Table)) ;
```

The query is above. Of course, the answer is XYZ Plumbing.

Quiz - How many rows return on a NOT IN with a NULL?

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84
999999	?	99999.99

We added a Null Value to the Order_Table NULL

```
SELECT      Customer_Name
FROM        Customer_Table
WHERE       Customer_Number NOT IN
     (SELECT Customer_Number FROM Order_Table ) ;
```

How many rows return from the query conceptually?

We really didn't place a new row inside the Order_Table with a NULL value for the Customer_Number column. In theory, if we had, how many rows would return?

Answer - How many rows return on a NOT IN with a NULL?

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84
999999	?	99999.99

We added a Null Value to the Order_Table NULL

```
SELECT      Customer_Name
FROM        Customer_Table
WHERE       Customer_Number NOT IN
     (SELECT Customer_Number FROM Order_Table ) ;
```

How many rows return from the query conceptually?
ZERO

The answer is no rows. This is because when you have a NULL value in a NOT IN list the system doesn't know the value of NULL so it returns nothing.

How to handle a NOT IN with Potential NULL Values

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84
999999	?	99999.99

```
SELECT      Customer_Name
FROM        Customer_Table
WHERE       Customer_Number NOT IN
     (SELECT Customer_Number FROM Order_Table
      WHERE Customer_Number IS NOT NULL) ;
```

How many rows return NOW from the query? 1 Acme Products

You can utilize a WHERE clause that tests to make sure Customer_Number IS NOT NULL. This should be used when a NOT IN could encounter a NULL.

IN is equivalent to =ANY

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84

Instead of an IN you can use the = ANY

```
SELECT Customer_Number
      ,Customer_Name
FROM   Customer_Table
WHERE  Customer_Number IN
  (SELECT Customer_Number
   FROM Order_Table ) ;
```

```
SELECT Customer_Number
      ,Customer_Name
FROM   Customer_Table
WHERE  Customer_Number = ANY
  (SELECT Customer_Number
   FROM Order_Table ) ;
```

Instead of using the IN, you can use the = ANY command. These queries work the SAME. The above queries will produce the same result set.

Using a Correlated Exists

Customer_Table		Order_Table		
Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91
31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

Use EXISTS to find which Customers have placed an Order?

```
SELECT    Customer_Number, Customer_Name
FROM      Customer_Table as Top1
WHERE     EXISTS
  (SELECT * FROM Order_Table as Bot1
    Where Top1.Customer_Number= Bot1.Customer_Number) ;
```

The EXISTS command will determine, via a Boolean, if something is True or False. If a customer placed and order it EXISTS and using the Correlated Exists statement only customers who have placed an order will return in the answer set. EXISTS is different than IN as it is less restrictive as you will soon understand.

How a Correlated Exists matches up

Customer_Table		Order_Table		
Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91
31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

```
SELECT    Customer_Number, Customer_Name
FROM      Customer_Table as Top1
WHERE     EXISTS
  (SELECT * FROM Order_Table as Bot1
    Where Top1.Customer_Number= Bot1.Customer_Number) ;
```

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Only customers who placed an order return with the above Correlated EXISTS.

The Correlated NOT Exists

Customer_Table		Order_Table		
Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91
31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

Use NOT EXISTS to find which Customers have NOT placed an Order?

```
SELECT     Customer_Number, Customer_Name
FROM       Customer_Table as Top1
WHERE      NOT EXISTS
  (SELECT * FROM Order_Table as Bot1
    Where Top1.Customer_Number= Bot1.Customer_Number) ;
```

The EXISTS command will determine, via a Boolean, if something is True or False. If a customer placed and order it EXISTS and using the Correlated Exists statement only customers who have placed an order will return in the answer set. EXISTS is different than IN as it is less restrictive as you will soon understand.

The Correlated NOT Exists Answer Set

Customer_Table		Order_Table		
Customer_Number	Customer_Name	Order_Number	Customer_Number	Order_Total
11111111	Billy's Best Choice	123456	11111111	12347.53
31313131	Acme Products	123512	11111111	8005.91
31323134	ACE Consulting	123552	31323134	5111.47
57896883	XYZ Plumbing	123585	87323456	15231.62
87323456	Databases N-U	123777	57896883	23454.84

Use NOT EXISTS to find which Customers have NOT placed an Order?

```
SELECT     Customer_Number, Customer_Name
FROM       Customer_Table as Top1
WHERE      NOT EXISTS
    (SELECT *  FROM Order_Table as Bot1
      Where Top1.Customer_Number = Bot1.Customer_Number ) ;
```

Customer_Number	Customer_Name
31313131	Acme Products

Answer Set

The only customer who did NOT place an order was Acme Products.

Quiz - How many rows come back from this NOT Exists?

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84
999999	?	99999.99

We added a Null Value to the Order_Table NULL

```
SELECT      Customer_Number, Customer_Name
FROM        Customer_Table as Top1
WHERE       NOT EXISTS
     (SELECT *  FROM Order_Table as Bot1
       Where Top1.Customer_Number = Bot1.Customer_Number ) ;
```

How many rows return from the query conceptually?

A NULL value in a list for queries with NOT IN returned nothing, but you must now decide if that is also true for the NOT EXISTS. How many rows will return?

Answer - How many rows come back from this NOT Exists?

Customer_Table

Customer_Number	Customer_Name
11111111	Billy's Best Choice
31313131	Acme Products
31323134	ACE Consulting
57896883	XYZ Plumbing
87323456	Databases N-U

Order_Table

Order_Number	Customer_Number	Order_Total
123456	11111111	12347.53
123512	11111111	8005.91
123552	31323134	5111.47
123585	87323456	15231.62
123777	57896883	23454.84
999999	?	99999.99

We added a Null Value to the Order_Table NULL

```
SELECT      Customer_Number, Customer_Name
FROM        Customer_Table as Top1
WHERE       NOT EXISTS
     (SELECT *  FROM Order_Table as Bot1
       Where Top1.Customer_Number = Bot1.Customer_Number ) ;
```

How many rows return from the query conceptually? One row
Acme Products

NOT EXISTS is unaffected by a NULL in the list. That's why it is more flexible!