Advance SQL: (TRANSACTION IN DATABASE USING ADVANCE SQL)

This sample examples of SQL(Structure Query Language) is one of the inspiration that made me to built 2 applications

1. Content Management System
2. E-Commerce

And all are in my GitHub account.

Note: I set up my table and performed different SQL operations as follows:

1. Union

The purpose of the SQL **UNION** query is to combine the results of two queries together while removing duplicates. In other words, when using **UNION**, only unique values are returned (similar to **SELECT** **DISTINCT**).

Example

The below 2 tables are used for the operations:

1) Table : ***Store_Information***

2) Table : ***Internet_Sales***

The Tables are in the Database called "dsti_sql2019"

Note: The following query are perform on the 2 Tables as follows:

```
SELECT Txn_Date FROM Store_Information
UNION
SELECT Txn_Date FROM Internet_Sales;
```

(SELECT Txn_Date FROM Store_Information UNION SELECT Txn_Date FROM Internet_Sales)

The Out comes as follows:

```
Txn_Date
Jan-07-1999
Jan-10-1999
Jan-11-1999
Jan-12-1999
```

Because its only Txn_Date that are common in both table in the columns.

- Important Note: Please note that if we type "**SELECT DISTINCT Txn_Date**" for either or both of the SQL statement, we will get the same result set.

2. Union All

The purpose of the SQL **UNION ALL** command is to combine the results of two queries together without removing any duplicates

The syntax for **UNION ALL** is as follows:

```
SELECT Txn_Date FROM Store_Information
UNION ALL
SELECT Txn_Date FROM Internet_Sales;
```

(SELECT Txn_Date FROM Store_Information UNION ALL SELECT Txn_Date FROM Internet_Sales)

To find out all the dates where there is a sales transaction at a store as well as all the dates where there is a sale over the internet, we use the following SQL statement:

The Outcome of the query as follows:

```
Txn_Date
Jan-05-199
Jan-07-1999
Jan-08-1999
Jan-08-1999
Jan-07-1999
Jan-10-1999
Jan-11-1999
Jan-12-1999
```

UNION vs UNION ALL

UNION and **UNION ALL** both combine the results of two SQL queries. The difference is that, while **UNION** only returns distinct values, **UNION ALL** selects all values. If we use **UNION** in the above example,

The Result outcome:

SELECT Txn_Date FROM Store_Information

UNION
SELECT Txn_Date FROM Internet_Sales;

(SELECT Txn_Date FROM Store_Information UNION SELECT Txn_Date FROM Internet_Sales)

Txn_Date
Jan-05-199
Jan-07-1999
Jan-08-1999
Jan-10-1999
Jan-11-1999
Jan-12-1999

Notice that while the **UNION ALL** query returns "Jan-07-1999" and "Jan-08-1999" twice, the **UNION** query returns each value only once.

3. Intersect

The **INTERSECT** command in SQL combines the results of two SQL statement and returns only data that are present in both SQL statements.

INTERSECT can be thought of as an **AND** operator (value is selected only if it appears in both statements), while **UNION** and **UNION ALL** can be thought of as an **OR** operator (value is selected if it appears in either the first or the second statement).

The syntax for **INTERSECT** is as follows:

The Following 2 tables show the tables we used for the query- in the database “dust-sql2019”

1. Table ***Store_Information***
2. Table ***Internet_Sales***

To find out all the dates where there are both store sales and internet sales, we use the following SQL statement:

The query and the out come

SELECT Txn_Date
FROM Store_Information
INTERSECT
SELECT Txn_Date

FROM Internet_Sales;

Txn_Date:
Jan-07-1999

- Important Notice: Please note that the **INTERSECT** command will only return distinct values.

4. LIMIT

The **LIMIT** clause restricts the number of results returned from a SQL statement. It is available in MySQL.

The Following 2 tables show the tables we used for the query- in the database "dust-sql2019"

1. Table ***Store_Information***
2. Table ***Internet_Sales***

To find out all the dates where there are both store sales and internet sales, we use the following SQL statement:

The syntax for **LIMIT** is as follows:

SELECT Store_Name, Sales, Txn_Date FROM Store_Information ORDER BY Sales DESC LIMIT 2

Store_Name	**Sales**	**Txn_Date**
Los Angeles	1500	Jan-05-199
Boston	700	Jan-08-1999

- Important Note: The SQL Server equivalent to **LIMIT** is **TOP**.

5. **Subquery**

A **subquery** is a SQL statement that has another SQL query embedded in the **WHERE** or the **HAVING** clause.

The syntax for a subquery when the embedded SQL statement is part of the **WHERE** condition is as follows:

The Following 2 tables show the tables we used for the query- in the database "dust-sql2019"

1. Table ***Store_Information***
2. Table ***Geography***

To find out all the dates where there are both store sales and internet sales, we use the following SQL statement:

The Simple subquery - To use a subquery to find the sales of all stores in the West region, we use the following SQL statement:

```
SELECT SUM(Sales) FROM Store_Information
WHERE Store_Name IN
(SELECT Store_Name FROM Geography
WHERE Region_Name = 'West');
```

SELECT SUM(Sales) FROM Store_Information WHERE Store_Name IN (SELECT Store_Name FROM Geopgraphy WHERE Region_Name = ‘East’)

```
Result:
 SUM(Sales)
   700
```

Example of Correlated subquery

```
  SELECT SUM(a1.Sales) FROM Store_Information a1
WHERE a1.Store_Name IN
(SELECT Store_Name FROM Geopgraphy a2
 WHERE a2.Store_Name = a1.Store_Name);
```

If the inner query is dependent on the outer query, we will have a **correlated subquery**. An example of a **correlated subquery** is shown below:

Result:

SUM(a1.Sales)
700

NOTE: Here, the inner query is used to make sure that SQL only sums up sales amount from stores that appear in both the *Store_Information* and the *Geography* tables.

*Important Note

Notice the **WHERE** clause in the inner query, where the condition involves a table from the outer query.

6. Exits

EXISTS is a Boolean operator used in a subquery to test whether the inner query returns any row. If it does, then the outer query
proceeds. If not, the outer query does not execute, and the entire SQL statement returns nothing.

The Following 2 tables show the tables we used for the query- in the database “dust-sql2019”

1. Table ***Store_Information***
2. Table ***Geography***

To find out all the dates where there are both store sales and internet sales, we use the following SQL

The syntax for **EXISTS** is:

```
SELECT SUM(Sales) FROM Store_Information
WHERE EXISTS
(SELECT * FROM Geopgraphy
WHERE Region_Name= 'West');
```

The Outcome Result is as thus:

```
SUM(Sales)
2750
```

*Important Note:

At first, this may appear confusing, because the subquery includes the [region_name = 'West'] condition, yet the query summed up sales for stores in all regions. Upon closer inspection, we find that since the subquery returns more than zero row, the **EXISTS** condition is true, and the rows returned from the query "SELECT SUM(Sales) FROM Store_Information" become the final result.

7. AUTO_INCREMENT

AUTO_INCREMENT is used in MySQL to create a numerical primary key value for each additional row of data.

The syntax for **AUTO_INCREMENT** is as follows:

Examples on how to use AUTO_INCREMENT:

Assume we want to create a table that consists of a primary key, last name, and first name. We run the following **CREATE TABLE** statement:

The Creation of the Table with auto_increment as follows:

```
CREATE TABLE USER_TABLE
(Userid int NULL AUTO_INCREMENT,
Last_Name varchar(50),
First_Name varchar(50),
PRIMARY KEY(Userid));
```

Here we have to populate the table with the following code:

Upon creation, there is no data in this table.

We insert the first value:

```
INSERT INTO `USER_TABLE`(`Userid`, `Last_Name`, `First_Name`) VALUES
(NULL,’Benya’,’Jamiu’);
INSERT INTO `USER_TABLE`(`Userid`, `Last_Name`, `First_Name`) VALUES
(NULL,’Shaffiyah’,’Hirakawa’);
INSERT INTO `USER_TABLE`(`Userid`, `Last_Name`, `First_Name`) VALUES
(NULL,'Adeyanju','Waheed');
```

Notice when we insert the first row, *Userid* is set to 1. When we insert the second row, *Userid* increases by 1 and becomes 2, etc

By default, **AUTO_INCREMENT** starts with 1 and increases by 1. To change the default starting value, we can use the **ALTER TABLE** command as follows:

The **AUTO INCREMENT** interval value is controlled by the MySQL Server variable auto_increment_increment and applies globally. To change this to a number different from the default of 1, use the following command in MySQL:

8. Rank

```
SELECT a1.Name,a1.Sales, COUNT(a2.Sales) Sales_Rank FROM Total_Sales a1,
Total_Sales a2 WHERE a1.Sales < a2.Sales OR (a1.Sales=a2.Sales AND a1.Name
= a2.Name) GROUP BY a1.Name, a1.Sales
```

9. Update

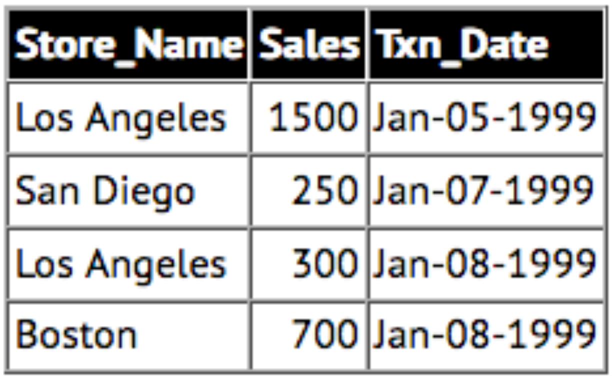
The **UPDATE** statement is used to modify data in a database table.

Syntax

UPDATE can be used to modify one column at a time or multiple columns at a time. The syntax for
updating a single column is as follows:

The following Table has been used for the SQL its in our Database table:

Table ***Store_Information***



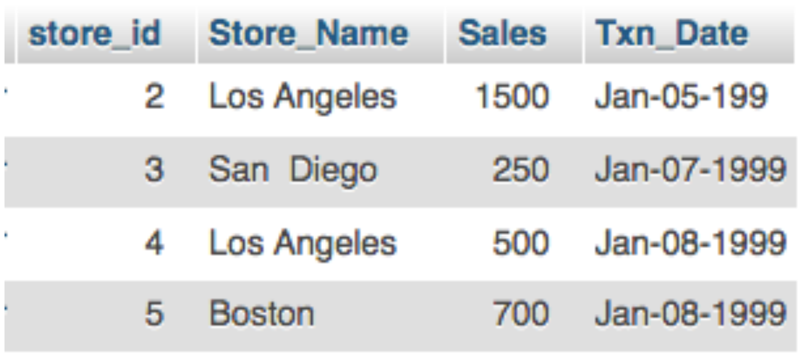
Store_Name	Sales	Txn_Date
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

a) Update a single column

The query goes as thus:

```
UPDATE Store_Information
SET Sales = 500
WHERE Store_Name = 'Los Angeles'
AND Txn_Date = 'Jan-08-1999'
```

We can see that after updating the Los Angeles with particular Txn_Date (Jan-08-1999) has been updated

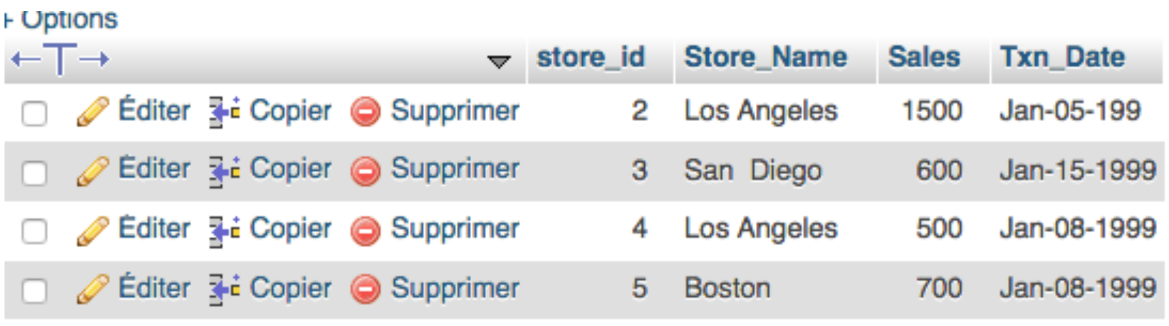


store_id	Store_Name	Sales	Txn_Date
2	Los Angeles	1500	Jan-05-199
3	San Diego	250	Jan-07-1999
4	Los Angeles	500	Jan-08-1999
5	Boston	700	Jan-08-1999

b) Update multiple columns

The following query has been perform on the Database (table) and the out comes as follows:

UPDATE Store_Information
SET Sales = 600, Txn_Date = 'Jan-15-1999'
WHERE Store_Name = 'San Diego';



Options

				store_id	Store_Name	Sales	Txn_Date
☐	Éditer	Copier	Supprimer	2	Los Angeles	1500	Jan-05-199
☐	Éditer	Copier	Supprimer	3	San Diego	600	Jan-15-1999
☐	Éditer	Copier	Supprimer	4	Los Angeles	500	Jan-08-1999
☐	Éditer	Copier	Supprimer	5	Boston	700	Jan-08-1999

IMPORTANT: When using the **UPDATE** statement, pay special attention to make sure that some type of filtering criteria is specified. Otherwise, the value of all rows can be changed all together is “SET” and particular “ID” is not using:

10. DELETE

The **DELETE FROM** statement in SQL is used to remove records from a table.

Please note that the **DELETE FROM** command cannot delete any rows of data that would violate **FOREIGN KEY** or other **constraints**.

The syntax for the **DELETE FROM** statement is as follows:

The **WHERE** clause is important here. Without specifying a condition, all records from the table will be deleted.

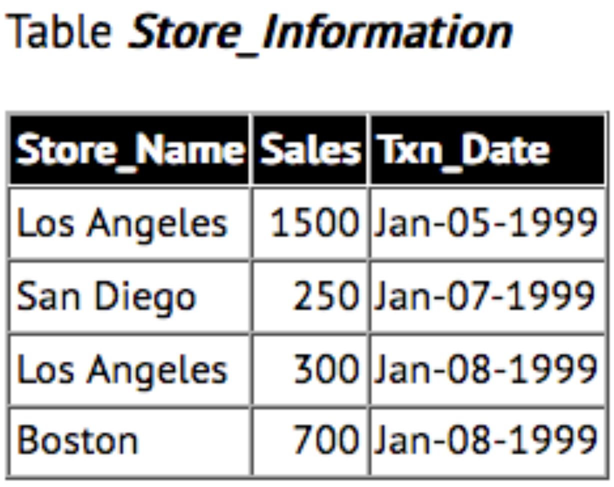


Table *Store_Information*

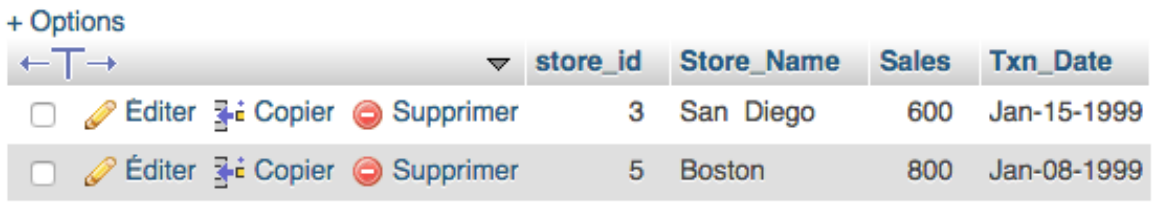
Store_Name	Sales	Txn_Date
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

This is the table from our Database and we are performing SQL on it:

We decide not to keep any information on Los Angeles in this table. To accomplish this, we type the following SQL:

The SQL is as Thus:

DELETE FROM Store_Information
WHERE Store_Name = 'Los Angeles';
and the outcome is as thus:



	store_id	Store_Name	Sales	Txn_Date
Éditer Copier Supprimer	3	San Diego	600	Jan-15-1999
Éditer Copier Supprimer	5	Boston	800	Jan-08-1999

DELETE FROM using the results from a subquery

in above example (Table), the criteria we use to determine which rows to delete is quite simple. We can also use a more complex condition. Below is an example where we use a subquery as the condition. Assume we have the following two tables:

Here we would be using 2 Tables as follows:

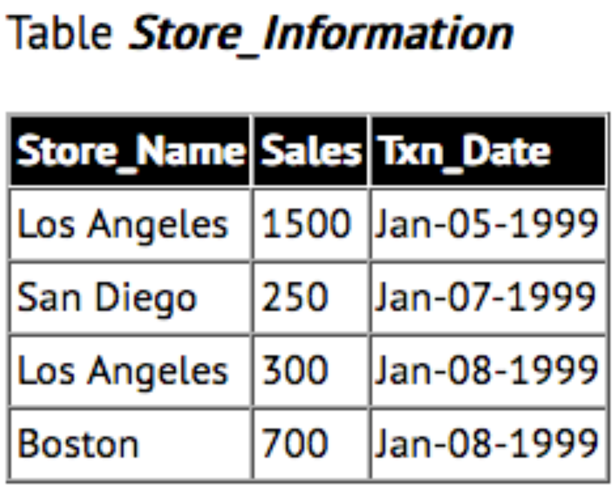


Table ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	Jan-05-1999
San Diego	250	Jan-07-1999
Los Angeles	300	Jan-08-1999
Boston	700	Jan-08-1999

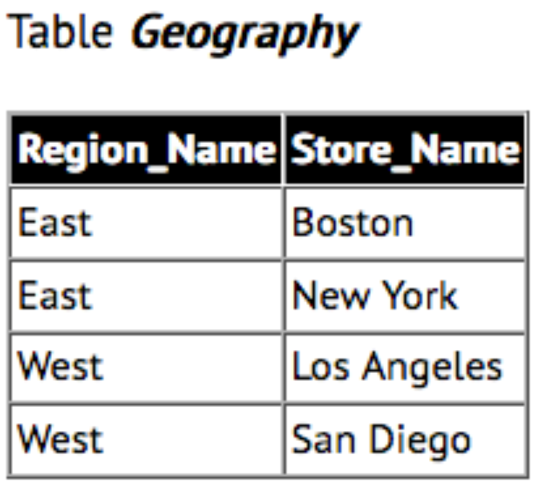


Table ***Geography***

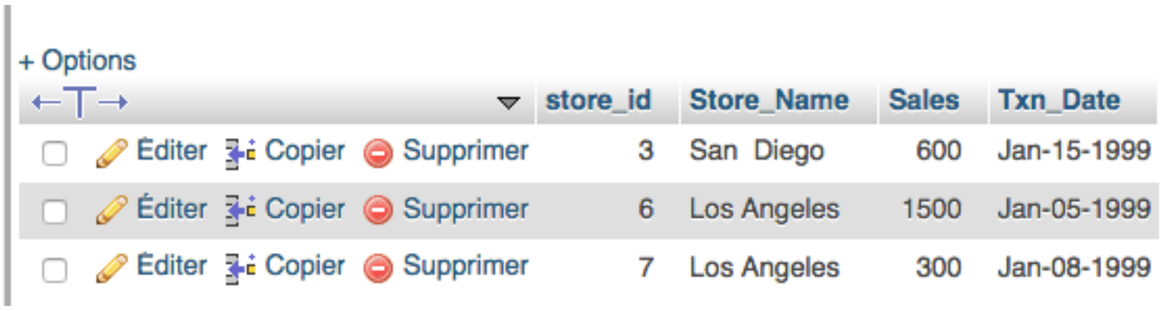
Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

We want to remove data for all stores in the East region from ***Store_Information*** (assuming that a store is either in the East region or the West region—it cannot be in more than one region). We use the following SQL statement to accomplish this:

The query goes as thus:

```
DELETE FROM Store_Information
WHERE Store_Name IN
(SELECT Store_Name FROM Geopgraphy
 WHERE Region_Name = 'East');
```

And the Result is as thus:

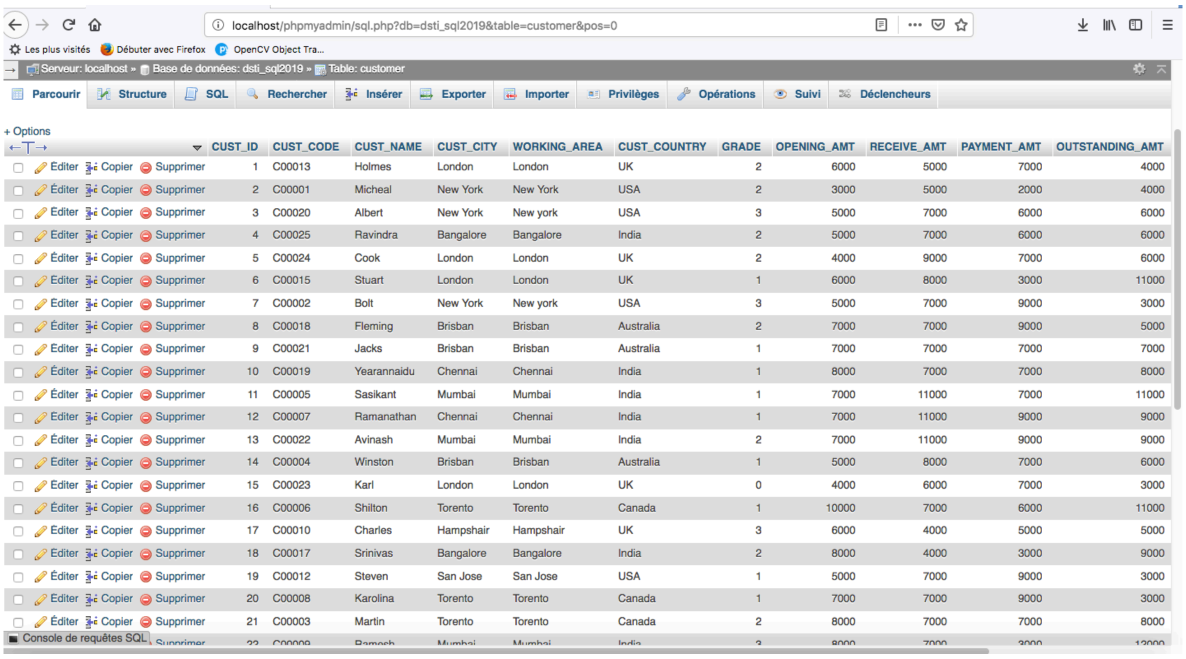


+ Options

	store_id	Store_Name	Sales	Txn_Date
Éditer Copier Supprimer	3	San Diego	600	Jan-15-1999
Éditer Copier Supprimer	6	Los Angeles	1500	Jan-05-1999
Éditer Copier Supprimer	7	Los Angeles	300	Jan-08-1999

Arithmetic Operators in SQL

Table



	CUST_ID	CUST_CODE	CUST_NAME	CUST_CITY	WORKING_AREA	CUST_COUNTRY	GRADE	OPENING_AMT	RECEIVE_AMT	PAYMENT_AMT	OUTSTANDING_AMT
Éditer Copier Supprimer	1	C00013	Holmes	London	London	UK	2	6000	5000	7000	4000
Éditer Copier Supprimer	2	C00001	Micheal	New York	New York	USA	2	3000	5000	2000	4000
Éditer Copier Supprimer	3	C00020	Albert	New York	New york	USA	3	5000	7000	6000	6000
Éditer Copier Supprimer	4	C00025	Ravindra	Bangalore	Bangalore	India	2	5000	7000	6000	6000
Éditer Copier Supprimer	5	C00024	Cook	London	London	UK	2	4000	9000	7000	6000
Éditer Copier Supprimer	6	C00015	Stuart	London	London	UK	1	6000	8000	3000	11000
Éditer Copier Supprimer	7	C00002	Bolt	New York	New york	USA	3	5000	7000	9000	3000
Éditer Copier Supprimer	8	C00018	Fleming	Brisban	Brisban	Australia	2	7000	7000	9000	5000
Éditer Copier Supprimer	9	C00021	Jacks	Brisban	Brisban	Australia	1	7000	7000	7000	7000
Éditer Copier Supprimer	10	C00019	Yearannaidu	Chennai	Chennai	India	1	8000	7000	7000	8000
Éditer Copier Supprimer	11	C00005	Sasikant	Mumbai	Mumbai	India	1	7000	11000	7000	11000
Éditer Copier Supprimer	12	C00007	Ramanathan	Chennai	Chennai	India	1	7000	11000	9000	9000
Éditer Copier Supprimer	13	C00022	Avinash	Mumbai	Mumbai	India	2	7000	11000	9000	9000
Éditer Copier Supprimer	14	C00004	Winston	Brisban	Brisban	Australia	1	5000	8000	7000	6000
Éditer Copier Supprimer	15	C00023	Karl	London	London	UK	0	4000	6000	7000	3000
Éditer Copier Supprimer	16	C00006	Shilton	Torento	Torento	Canada	1	10000	7000	6000	11000
Éditer Copier Supprimer	17	C00010	Charles	Hampshair	Hampshair	UK	3	6000	4000	5000	5000
Éditer Copier Supprimer	18	C00017	Srinivas	Bangalore	Bangalore	India	2	8000	4000	3000	9000
Éditer Copier Supprimer	19	C00012	Steven	San Jose	San Jose	USA	1	5000	7000	9000	3000
Éditer Copier Supprimer	20	C00008	Karolina	Torento	Torento	Canada	1	7000	7000	9000	3000
Éditer Copier Supprimer	21	C00003	Martin	Torento	Torento	Canada	2	8000	7000	7000	8000
Supprimer	22	C00009	Ramesh	Mumbai	Mumbai	India	3	8000	7000	3000	12000

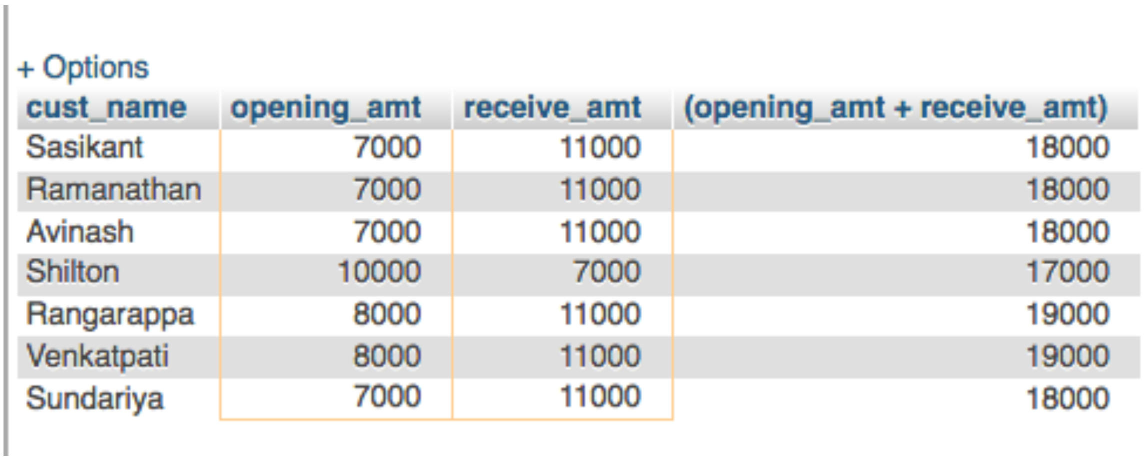
QUERY: (+)
SAMPLE OF QUERY FOR INSERT:

INSERT INTO `customer` (`CUST_ID`, `CUST_CODE`, `CUST_HOME`, `CUST_CITY`, `WORKING_AREA`, `CUST_COUNTRY`, `GRADE`, `OPENING_AMT`, `RECEIVE_AMT`, `PAYMENT_AMT`, `OUTSTANDING_AMT`, `PHONE_NO`, `AGENT_CODE`) VALUES (NULL, ' C00015 ', 'Stuart ', 'London', 'London', 'UK', '1', '6000.00', '8000.00', '3000.00', '11000.00', 'GFSGERS', 'A008');

PERFORMING OPERATION IN THE ABOVE TABLE WITH THE FOLLOWING QUERY:

```
SELECT cust_name, opening_amt,
receive_amt, (opening_amt + receive_amt)
FROM customer
WHERE (opening_amt + receive_amt)>15000;
```

OUTCOME:



+ Options

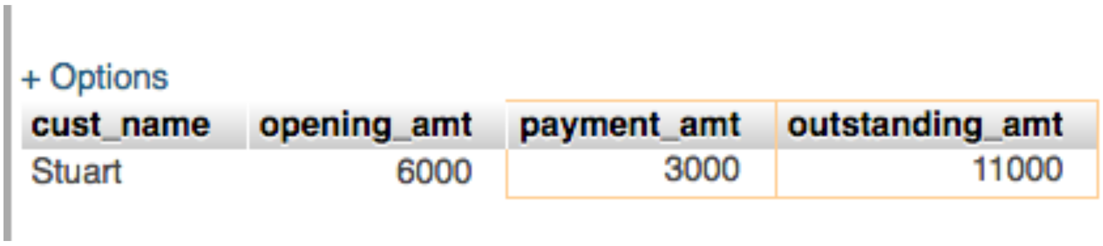
cust_name	opening_amt	receive_amt	(opening_amt + receive_amt)
Sasikant	7000	11000	18000
Ramanathan	7000	11000	18000
Avinash	7000	11000	18000
Shilton	10000	7000	17000
Rangarappa	8000	11000	19000
Venkatpati	8000	11000	19000
Sundariya	7000	11000	18000

SQL minus (-) operator

FROM THE SAME TABLE THE FOLLOWING OPERATION IN QUERY WAS PERFORMER:

```
SELECT cust_name,opening_amt, payment_amt, outstanding_amt
FROM customer
WHERE(outstanding_amt-payment_amt)=receive_amt;
```

THE OUTCOME IS AS FOLLOW:



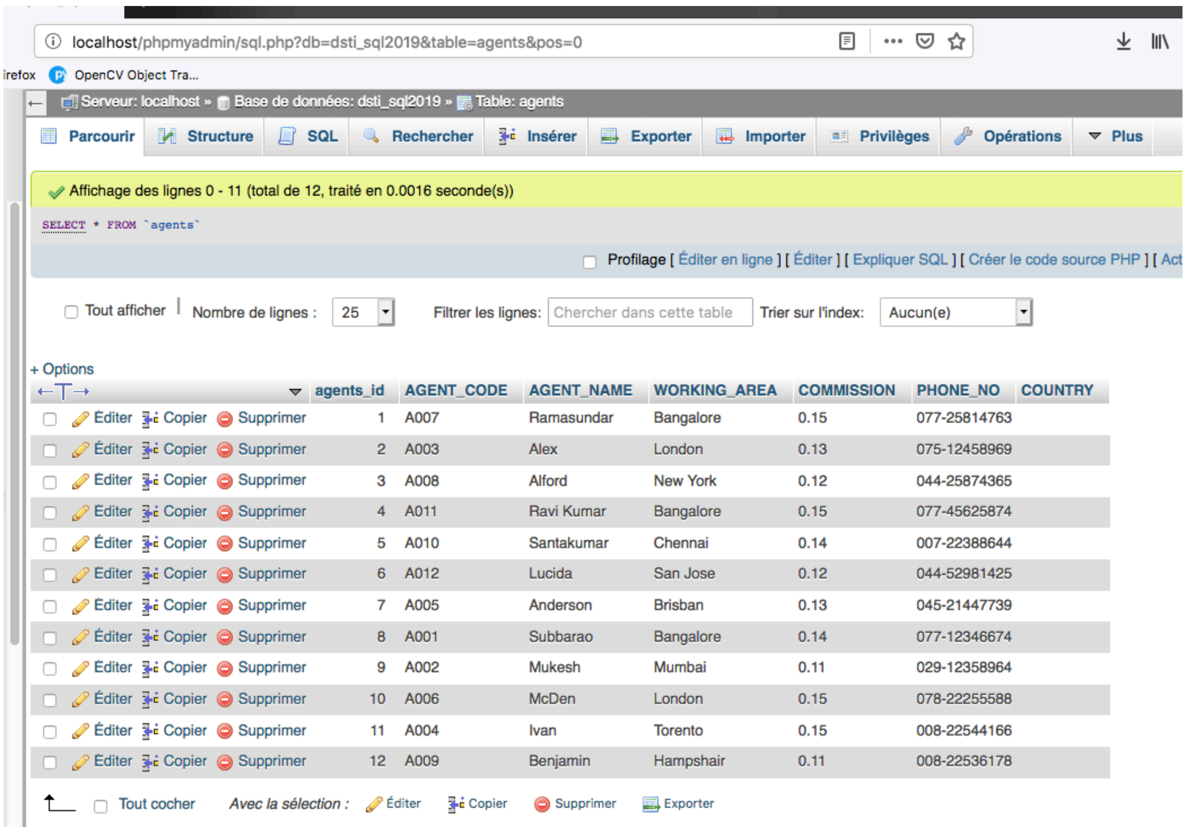
+ Options

cust_name	opening_amt	payment_amt	outstanding_amt
Stuart	6000	3000	11000

SQL multiply (*) operator

The SQL multiply (*) operator is used to multiply two or more expressions or numbers.

The above operation was performed in the following Table:


localhost/phpmyadmin/sql.php?db=dsti_sql2019&table=agents&pos=0
irefox
OpenCV Object Tra...
Serveur: localhost » Base de données: dsti_sql2019 » Table: agents
Parcourir
Structure
SQL
Rechercher
Insérer
Exporter
Importer
Privilèges
Opérations
Plus
Profilage [Éditer en ligne] [Éditer] [Expliquer SQL] [Créer le code source PHP] [Act
Tout afficher
Nombre de lignes : 25
Filtrer les lignes: Chercher dans cette table
Trier sur l'index: Aucun(e)
+ Options
Tout cocher
Avec la sélection : Éditer Copier Supprimer Exporter

Affichage des lignes 0 - 11 (total de 12, traité en 0.0016 seconde(s))

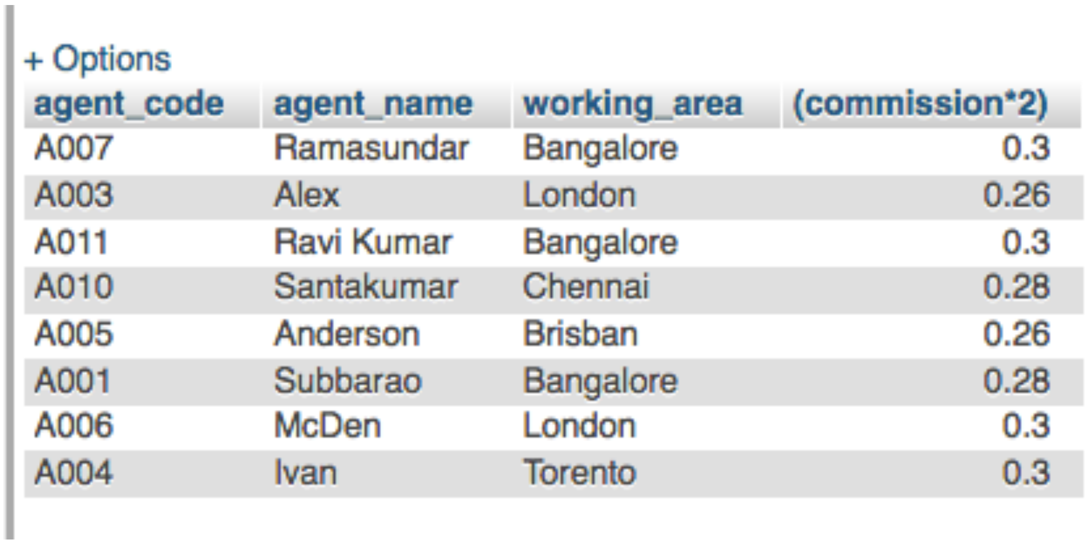
```
SELECT * FROM `agents`
```

agents_id	AGENT_CODE	AGENT_NAME	WORKING_AREA	COMMISSION	PHONE_NO	COUNTRY
1	A007	Ramasundar	Bangalore	0.15	077-25814763	
2	A003	Alex	London	0.13	075-12458969	
3	A008	Alford	New York	0.12	044-25874365	
4	A011	Ravi Kumar	Bangalore	0.15	077-45625874	
5	A010	Santakumar	Chennai	0.14	007-22388644	
6	A012	Lucida	San Jose	0.12	044-52981425	
7	A005	Anderson	Brisban	0.13	045-21447739	
8	A001	Subbarao	Bangalore	0.14	077-12346674	
9	A002	Mukesh	Mumbai	0.11	029-12358964	
10	A006	McDen	London	0.15	078-22255588	
11	A004	Ivan	Torento	0.15	008-22544166	
12	A009	Benjamin	Hampshair	0.11	008-22536178	

The following query was performed ;

SELECT agent_code, agent_name,
working_area, (commission*2)
FROM agents
WHERE (commission*2)>0.25;

and the outcome follows as thus:



+ Options

agent_code	agent_name	working_area	(commission*2)
A007	Ramasundar	Bangalore	0.3
A003	Alex	London	0.26
A011	Ravi Kumar	Bangalore	0.3
A010	Santakumar	Chennai	0.28
A005	Anderson	Brisban	0.26
A001	Subbarao	Bangalore	0.28
A006	McDen	London	0.3
A004	Ivan	Torento	0.3

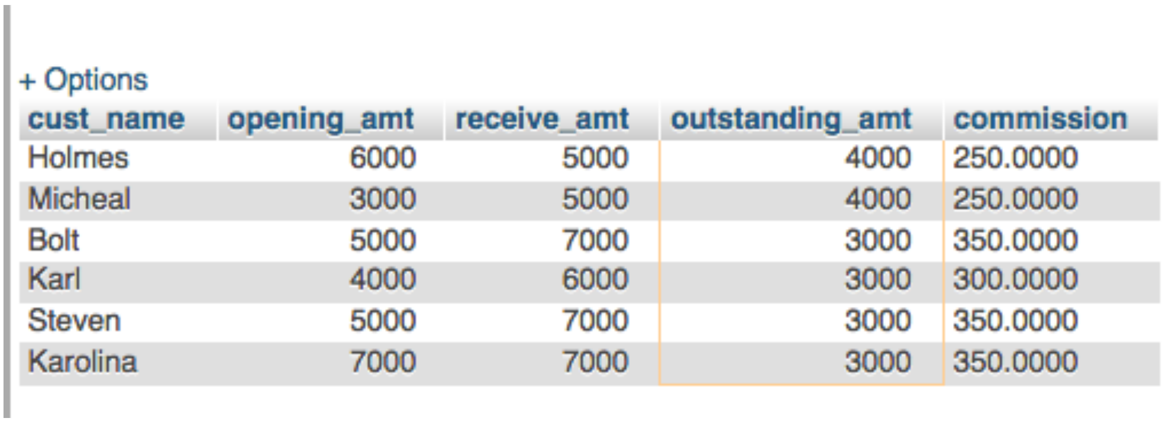
SQL divide (/) operator

Here we performed the operation within the 2 tables
1 Table : customer
2 Table : agents

With the following query:

SELECT cust_name, opening_amt, receive_amt,
outstanding_amt, (receive_amt*5/ 100) commission
FROM customer
WHERE outstanding_amt<=4000;

And the Final Table Combine (COMMISSION) from Agents Table to the new table:



+ Options

cust_name	opening_amt	receive_amt	outstanding_amt	commission
Holmes	6000	5000	4000	250.0000
Micheal	3000	5000	4000	250.0000
Bolt	5000	7000	3000	350.0000
Karl	4000	6000	3000	300.0000
Steven	5000	7000	3000	350.0000
Karolina	7000	7000	3000	350.0000

SQL modulo (%) operator

References:

1 - https://www.w3resource.com/sql/arithmetic-operators/sql-arithmetic-operators.php

2 - https://www.1keydata.com/fr/sql/