**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session3-Module3.sql**

**\*\* Desc: Summarizing Data (DML)**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/21/2018**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

########################## COUNT ##########################

# COUNT function used to count rows that do not contain a NULL value.

# number of employees in the employees table

SELECT

COUNT(addressLine2) AS numberOfEmployees

FROM

customers;

# count on single column - employeeNumber

SELECT

COUNT(employeeNumber) AS numberOfEmployees

FROM

employees;

# count on column 1

SELECT

COUNT(1) AS numberOfEmployees

FROM

employees;

# use keyword ALL

SELECT

COUNT(ALL addressLine2) AS numberOfCustomers

FROM

customers;

SELECT

COUNT(ALL status) AS diffStatus

FROM

orders;

# add columns ahead of the COUNT function to showcase varied results

# incorrect use - the results do not make sense

# the first row is returned with the count results

SELECT

\*,

COUNT(employeeNumber) AS numberOfEmployees

FROM

employees;

# find all total number of unique offices from employees table

SELECT

COUNT(DISTINCT officeCode) AS numberOfOffices

FROM

employees;

# find all total number of offices from employees table

SELECT

COUNT(officeCode) AS numberOfOffices

FROM

employees;

# Count ignores null values

# number of sales representatives in the customer table

SELECT

COUNT(salesRepEmployeeNumber) AS numberOfSalesReps

FROM

customers;

# total number of rows in the customer table

SELECT

COUNT(\*) AS numberOfCustomers

FROM

customers;

################## COUNT with WHERE clause #################

# number of orders that were placed by customers with id 103 and 114.

SELECT

COUNT(\*) AS ordersPlaced

FROM

orders

WHERE

customerNumber = 103

OR customerNumber = 114;

# count of customers living in Auckland or Brickhaven

SELECT

COUNT(customerNumber)

FROM

customers

WHERE

city = 'Auckland' OR city = 'Brickhaven';

# number of customers who made payments in 2015

SELECT

DISTINCT(COUNT(customerNumber)) AS totalCustomers

FROM

payments

WHERE

paymentDate >= '2015-01-01'

AND paymentDate < '2016-01-01';

SELECT

COUNT(DISTINCT customerNumber) AS totalCustomers

FROM

payments

WHERE

paymentDate >= '2015-01-01'

AND paymentDate < '2016-01-01';

SELECT

COUNT(customerNumber) AS totalCustomers

FROM

payments

WHERE

paymentDate >= '2015-01-01'

AND paymentDate < '2016-01-01';

# number of employees working at the Boston office (without joining employee and office tables - not recommended, join is perferred)

#step1: find out the office code of the Boston office

SELECT

officeCode

FROM

offices

WHERE

city = 'Boston';

#step2: find the employees at the Boston location

SELECT

COUNT(employeeNumber) AS numberOfEmployees

FROM

employees

WHERE

officeCode = 2;

############## AGGREGATES - SUM, MIN, MAX ####################

# SUM - returns a total on the values of a column for a group of rows. NULL values are ignored

# total quantity ordered by customers

SELECT

SUM(quantityOrdered) AS totalQuantityOrdered

FROM

orderdetails;

# MIN - returns the minimum value of a column for a group of rows. NULL values are ignored

# minimum units for any order

SELECT

MIN(quantityOrdered) AS minQuantityOrdered

FROM

orderdetails;

# MAX - returns the maximum value of a column for a group of rows. NULL values are ignored.

# maximum units for any order

SELECT

MAX(quantityOrdered) AS maxQuantityOrdered

FROM

orderdetails;

############## AGGREGATES - STATISTICAL ##################

# Arithmetic mean - the average of a set of numerical values, calculated by adding them together and dividing by the number of terms in the set.

# AVG - return a total on the values of a column for a group of rows

# arithmetic mean of the buyPrice for all products

SELECT

SUM(priceEach) / COUNT(priceEach) AS avgPrice,

AVG(priceEach) AS avgPrice

FROM

orderdetails;

# STD - population standard deviation

# standard deviation of units across all orders

SELECT

STD(quantityOrdered) AS stdQuantityOrdered

FROM

orderdetails;

# VARIANCE - population standard variance

# variance of units across all orders

SELECT

VARIANCE(quantityOrdered) AS varQuantityOrdered

FROM

orderdetails;

# Weighted average - an average resulting from the multiplication of each component by a factor reflecting its importance.

# weighted average of the buyPrice ( number of units \* price ) for all products

SELECT

SUM(quantityOrdered \* priceEach) / SUM(priceEach) AS weightedAverage

FROM

orderdetails;

# Geometric mean - indicates the central tendency of a set of numbers by using the product of their values

# geometric mean of the buyPrice for all products

SELECT

EXP(SUM(LOG(priceEach)) / COUNT(priceEach)) AS geometricMean

FROM

orderdetails;

# midrange - arithmetic mean of the maximum and minimum values in a data set

# midrange of the buyPrice for all products

SELECT

(MAX(priceEach) + MIN(priceEach)) / 2 AS midRange

FROM

orderdetails;

# combining aggregates in a single row

# some statistics on the ordersdetails table

SELECT

COUNT(DISTINCT orderNumber),

MIN(quantityOrdered) AS minQuantityOrdered,

MAX(quantityOrdered),

AVG(quantityOrdered),

SUM(quantityOrdered),

MAX(priceEach),

MIN(priceEach),

STD(priceEach)

FROM

orderdetails;

############## AGGREGATES with WHERE clause ##################

# sum of inventory in stock with buy price between 50$ and 100$

SELECT

SUM(quantityInStock) AS totalQuantityInStock

FROM

products

WHERE

buyPrice BETWEEN 50 AND 100;

# sum of inventory ( other than motorcycles ) in stock with buy price between 50$ and 100$

SELECT

SUM(quantityInStock) AS totalQuantityInStock

FROM

Products

WHERE

(buyPrice BETWEEN 50 AND 100)

AND NOT productLine IN ('Motorcycles');

# Validate

SELECT

SUM(quantityInStock) AS totalQuantityInStock

FROM

Products

WHERE

productLine='Motorcycles' AND (buyPrice BETWEEN 50 AND 100);

# average credit limit of customers living in USA or FRANCE and with

# creditLimit > 100000

SELECT

AVG(creditLimit) AS avgCreditLimit

FROM

customers

WHERE

(country = 'USA' OR country = 'France')

AND creditlimit > 100000;

# minimum MSRP for the list of products that are not Harley Davidson

SELECT

MIN(MSRP) AS minMSRP

FROM

products

WHERE

productName NOT LIKE '%Harley%';

# total amount spent by customers for the year 2016

SELECT

COUNT(DISTINCT(customerNumber)) AS totalCustomers,

SUM(amount) AS totalAmountSpent

FROM

payments

WHERE

paymentDate BETWEEN CAST('2016-01-01' AS DATE) AND CAST('2016-12-31' AS DATE);

# total orders that have been shipped or cancelled

SELECT

SUM(IF(status = 'Shipped', 1, 0)) AS Shipped,

SUM(IF(status = 'Cancelled', 1, 0)) AS Cancelled

FROM

orders;

# total orders by each status using IF condition

SELECT

COUNT(IF(status = 'Shipped', 1, NULL)) 'Shipped',

COUNT(IF(status = 'On Hold', 1, NULL)) 'On Hold',

COUNT(IF(status = 'In Process', 1, NULL)) 'In Process',

COUNT(IF(status = 'Resolved', 1, NULL)) 'Resolved',

COUNT(IF(status = 'Cancelled', 1, NULL)) 'Cancelled',

COUNT(IF(status = 'Disputed', 1, NULL)) 'Disputed',

COUNT(\*) AS Total

FROM

orders;

# total orders by each status using CASE statements

SELECT

SUM(CASE

WHEN status = 'Shipped' THEN 1

ELSE 0

END) AS 'Shipped',

SUM(CASE

WHEN status = 'On Hold' THEN 1

ELSE 0

END) AS 'On Hold',

SUM(CASE

WHEN status = 'In Process' THEN 1

ELSE 0

END) AS 'In Process',

SUM(CASE

WHEN status = 'Resolved' THEN 1

ELSE 0

END) AS 'Resolved',

SUM(CASE

WHEN status = 'Cancelled' THEN 1

ELSE 0

END) AS 'Cancelled',

SUM(CASE

WHEN status = 'Disputed' THEN 1

ELSE 0

END) AS 'Disputed',

COUNT(\*) AS Total

FROM

orders;

#ref : <http://www.mysqltutorial.org/>

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session3-Module4.sql**

**\*\* Desc: Sorting and Grouping Data (DML)**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/21/2018**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

######################## ORDER BY ########################

# ORDER BY - used to sort the result set

# customers sorted ( single column ) by last name in ascending order

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city

FROM

customers

ORDER BY contactLastname;

# use ASC keyword and compare results from previous query. results should be the same

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city

FROM

customers

ORDER BY contactLastname ASC;

# customers sorted by last name in descending order

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city

FROM

customers

ORDER BY contactLastname DESC;

# same as above but ordered using column number (integer)

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city

FROM

customers

ORDER BY 1 DESC;

# customers sorted by state and city ( multiple columns )

# sort the state column first in descending order and then sort the

# city within each state in ascending order.

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city

FROM

customers

ORDER BY state DESC, city ASC;

# remove null state values

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city

FROM

customers

WHERE

state IS NOT NULL

ORDER BY state DESC , city ASC;

######################## GROUP BY ########################

# GROUP BY - arrange identical data into groups

# generally not necessary unless using aggregate functions

##########################################################

# count of order status ( single column ) that were either cancelled/Disputed/Resolved/Shipped

SELECT

status, COUNT(\*) AS numberOfOrders

FROM

orders

GROUP BY status; #group by is used for aggregation

# use ASC keyword and compare results from previous query. results should be the same

SELECT

status, COUNT(\*) AS numberOfOrders

FROM

orders

GROUP BY status; # GROUP BY status ASC; will give an error. ascending doesn't work anymore. New version doesn't support ascending/Descending

# order status in descending order

SELECT

status, COUNT(\*) AS numberOfOrders

FROM

orders

GROUP BY status; #GROUP BY status DESC;

# same example as above but ordered by using an integer to represent the column name

SELECT

status, COUNT(\*) AS numberOfOrders

FROM

orders

GROUP BY 1; #GROUP BY 1 DESC;

# customers grouped by customer number and status ( multiple columns )

# group the first in descending order and then sort the city within each state

# in ascending order. (note customer 328 or 119 )

SELECT

customerNumber, status, COUNT(\*) AS numberOfOrders

FROM

orders

GROUP BY customerNumber , status; #GROUP BY customerNumber ASC , status DESC;

# Basket value of each order

SELECT

orderNumber,

SUM(quantityOrdered \* priceEach) AS total

FROM

orderdetails

GROUP BY orderNumber ; #GROUP BY orderNumber DESC ; Combine by end all together. Group by first

######################## GROUP BY and ORDER BY ########################

# GROUP BY clause is always placed before ORDER BY clause

########################################################################

# number of orders and status of order per each customer

SELECT

customerNumber,

status,

COUNT(\*) AS numberOfOrders

FROM

orders

GROUP BY customerNumber , status

ORDER BY numberOfOrders DESC;

# number of customers in each city

SELECT

city,

COUNT(\*) AS numberOfCustomers

FROM

CUSTOMERS

GROUP BY city

ORDER BY 2 DESC, 1;

######################## HAVING ########################

# HAVING - filter condition for groups of rows or aggregates

# HAVING is to GROUP BY as WHERE is to SELECT

#########################################################

# find orders where quantity ordered >100

SELECT

ordernumber,

SUM(quantityOrdered) AS totalQuantity

FROM

orderdetails

GROUP BY ordernumber

HAVING totalQuantity > 100; # give the

# find the products where total price sold is greater than 10000

# order by total price sold

SELECT

productCode,

SUM(quantityOrdered) as totalQuantity,

SUM(quantityOrdered\*priceEach) as totalPrice

FROM

orderdetails

GROUP BY productCode

HAVING totalPrice > 10000

ORDER BY totalPrice DESC;

/\*

The difference between the WHERE & HAVING is in the relationship to the GROUP BY clause:

- WHERE comes before GROUP BY; SQL evaluates the WHERE clause before it groups records.

- HAVING comes after GROUP BY; SQL evaluates HAVING after it groups records.

- HAVING clause applies the filter condition to each group of rows, while the WHERE

clause applies the filter condition to each individual row.

\*/

# List all products that matches a pattern ( string )

SELECT

productCode,

SUM(quantityOrdered) as totalQuantity,

SUM(quantityOrdered\*priceEach) as totalPrice

FROM

orderdetails

WHERE productCode like 'S18%'

GROUP BY productCode

HAVING totalPrice > 10000

ORDER BY totalPrice DESC;

SELECT

productCode,

SUM(quantityOrdered) as totalQuantity,

SUM(quantityOrdered\*priceEach) as totalPrice

FROM

orderdetails

GROUP BY productCode

HAVING productCode like 'S18%' AND totalPrice > 10000

ORDER BY totalPrice DESC;

######################## LIMIT ########################

# Top 5 customers with highest credit limit

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city,

creditLimit

FROM

customers

ORDER BY creditlimit DESC

LIMIT 5;

# 5 customers who have the lowest credit limit

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city,

creditLimit

FROM

customers

WHERE creditLimit > 0

ORDER BY creditlimit ASC

LIMIT 5;

# Return only the customer with 2nd highest credit limit

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city,

creditLimit

FROM

customers

ORDER BY creditlimit DESC

LIMIT 1 , 1; # After the first one, gives the second one.

# Staring from the 5th highest credit limit, get the next 7 customers

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city,

creditLimit

FROM

customers

ORDER BY creditlimit DESC

LIMIT 5 ,7;

# Staring from the 3rd highest credit limit, get the next 5 customers

SELECT

contactLastname,

contactFirstname,

customerNumber,

state,

city,

creditLimit

FROM

customers

ORDER BY creditlimit DESC

LIMIT 5 OFFSET 3;

#LIMIT 3, 5;

# Combine all the clauses - WHERE + GROUPBY + HAVING + ORDER BY + LIMIT

# Find the top 2 cities in california with the highest Average Credit Limit for customers

SELECT

city, AVG(creditLimit) as AvgCreditLimit

FROM

customers

WHERE

state like "CA"

GROUP BY city DESC

HAVING city like "San%"

ORDER BY AvgCreditLimit DESC

LIMIT 2;

#################### STATISTICAL FUNCTION ####################

# find the most frequently occuring (mode) buyPrice across products

SELECT

buyPrice AS buyPriceMode,

productName

FROM

products

GROUP BY 1,2

ORDER BY COUNT(1) DESC

LIMIT 1;

#ref : <http://www.mysqltutorial.org/>

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module1.sql**

**\*\* Desc: Combining Data (DML)**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2018**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

######################## ALIAS ########################

# allows SQL queries and the result set to be more readable

# column alias - customerName column & postalCode column

# table alias - customer table

SELECT

customerNumber,

customerName AS businessName,

contactFirstName,

contactLastName,

postalCode AS postCode

FROM

customers AS cust;

################################## JOINS ###############################

# A join combines two or more tables to retrieve data from multiple tables

# INNER JOIN: Returns all rows when there is at least one match in BOTH tables

# Outer JOIN :

# LEFT JOIN: Return all rows from the left table, and the matched rows from the right table

# RIGHT JOIN: Return all rows from the right table, and the matched rows from the left table

# FULL JOIN: Return all rows when there is a match in ONE of the tables

# LEFT JOIN and RIGHT JOIN are shorthand for LEFT OUTER JOIN and RIGHT OUTER JOIN

######################## INNER JOIN OR EQUI JOINS ########################

# Returns all rows when there is at least one match in BOTH tables ( intersection )

# by joining two tables with a common column

# old Syntax for inner joins

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

o.orderNumber,

o.orderDate,

o.status

FROM

customers AS C,

orders AS o

WHERE

C.customerNumber = o.customerNumber;

# returns customers that have placed an order ( 2 tables )

# with join - 326 rows

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

o.orderNumber,

o.orderDate,

o.status

FROM

customers c

INNER JOIN

orders o ON c.customerNumber = o.customerNumber;

#################### CROSS JOIN OR The Cartesian Product ##################

# without the where clause - 122 consumer records \* 326 order records = 39772

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

o.orderNumber,

o.orderDate,

o.status

FROM

customers c, orders o;

# Exercise : join customers and order tables using the CROSS JOIN keyword

# products customerNumber 119 brought and the status of each ( 3 tables )

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

o.customerNumber,

o.orderNumber,

o.orderDate,

o.status,

d.productCode

FROM

customers c

INNER JOIN

orders o ON c.customerNumber = o.customerNumber

INNER JOIN

orderDetails d ON o.orderNumber = d.orderNumber

WHERE

c.customerNumber = 119;

######################## NATURAL JOIN ########################

# A Natural Join is where 2 tables are joined on the basis of all common columns.

# same as inner join but eliminates duplicate columns in the joining columns

# customerNumber duplicated on the first query - first column and the last column

SELECT

\*

FROM

customers c

INNER JOIN

orders o ON c.customerNumber = o.customerNumber;

# with natural join - No ON criteria match

SELECT

\*

FROM

customers

NATURAL JOIN

orders;

######################## LEFT JOIN ########################

# select rows from the both left and right tables that are matched,

# plus all rows from the left table ( T1 ) even there is no match

# found for them in the right table ( T2 ).

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

orderNumber,

o.orderDate,

o.STATUS

FROM

customers c

LEFT JOIN

orders o ON c.customerNumber = o.customerNumber

# rows in the left table that do not match with the rows in the right table

# customers who have not placed orders

WHERE

orderNumber IS NULL;

# Exercise : products customerNumber 119 brought and the status of each ( 3 tables )

# HINT : USE LEFT JOINS on customers, orders and orderDetails tables

# list the offices where each employee works along with the office address

SELECT

e.employeeNumber,

e.firstName,

e.lastName,

o.officeCode,

o.city

FROM

employees e

LEFT OUTER JOIN

offices o ON e.officecode = o.officecode;

# insert a record into the employees table

INSERT INTO `employees`

(`employeeNumber`,`lastName`,`firstName`,`extension`,`email`,`officeCode`,`reportsTo`, `jobTitle`)

VALUES

(2045,'Doe','John','x5801','jdoe@classicmodelcars.com',7,NULL,'founder');

SELECT

e.employeeNumber,

e.firstName,

e.lastName,

o.officeCode,

o.city

FROM

employees e

LEFT OUTER JOIN

offices o ON e.officecode = o.officecode;

# Exercise : delete employeeNumber 2045 record from the employee table

######################## RIGHT JOIN ########################

# The RIGHT JOIN keyword returns all the rows from the right table

# even if there are no matches in the left table.

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

orderNumber,

o.orderDate,

o.STATUS

FROM

customers c

RIGHT outer JOIN

orders o ON c.customerNumber = o.customerNumber;

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

o.orderNumber,

o.orderDate,

productCode,

o.STATUS

FROM

customers c

RIGHT OUTER JOIN

orders o ON c.customerNumber = o.customerNumber

RIGHT OUTER JOIN

orderDetails d ON o.orderNumber = d.orderNumber

WHERE

c.customerNumber = 119;

# rows in the left table that do not match with the rows in the right table

# WHERE

# c.customerNumber = 125; / \* OR 168 \*/

######################## SELF JOIN ########################

# list of employees along with their managers and designation ( organization structure )

SELECT

e1.firstName AS managerFirstName,

e1.lastName AS managerLastName,

e1.jobTitle AS managerJobTitle,

e2.firstName AS empFirstName,

e2.lastName AS empLastName,

e2.jobTitle AS empJobTitle

FROM

employees e1,

employees e2

WHERE

e1.employeeNumber = e2.reportsTo;

# same results as above using INNER JOIN

SELECT

e1.firstName AS managerFirstName,

e1.lastName AS managerLastName,

e1.jobTitle AS managerJobTitle,

e2.firstName AS empFirstName,

e2.lastName AS empLastName,

e2.jobTitle AS empJobTitle

FROM

employees e1

INNER JOIN

employees e2

ON

e1.employeeNumber = e2.reportsTo;

# Additional Queries with GroupBy and Having

# Total revenue for each year for the products that were shipped ?

SELECT

YEAR(orderDate) AS year,

SUM(quantityOrdered \* priceEach) AS total

FROM

orders

INNER JOIN

orderdetails USING (orderNumber)

WHERE

status = 'Shipped'

GROUP BY YEAR(orderDate);

# Exercise : Find all the orders that were shipped with total sales greater than $1500

#### SET OPERATORS - UNION is the only compound operator supported in MySQL ####

# union on customers and offices based on location

(SELECT

city,state,country

FROM

customers) UNION (SELECT

city,state,country

FROM

offices)

ORDER BY 2 , 1;

# Remove states with NULL value

(SELECT

city,state,country

FROM

customers

WHERE state is NOT NULL ) UNION (SELECT

city,state,country

FROM

offices

WHERE state is NOT NULL)

ORDER BY 2 , 1;

# UNION ALL operator does not remove duplicates - check city San Francisco

(SELECT

city,state,country

FROM

customers

WHERE state is NOT NULL ) UNION ALL (SELECT

city,state,country

FROM

offices

WHERE state is NOT NULL)

ORDER BY 2 , 1;

# Full Outer Join - My SQL does not support Full Outer Joins

SELECT

c.customerNumber,

c.customerName,

c.contactFirstName,

c.contactLastName,

c.postalCode,

o.orderNumber,

o.orderDate,

o.status

FROM

customers c

FULL OUTER JOIN

orders o ON c.customerNumber = o.customerNumber;

# Below is the syntax to get to full join. The first query relating to Left Join returns 350 rows and second query relating to the Right Join returns 326 rows totalling 676 rows. Check for customer number 206 that exists in the customer table but not in orders table

SELECT \* FROM customers

LEFT JOIN orders ON customers.customerNumber = orders.customerNumber

UNION ALL

SELECT \* FROM customers

RIGHT JOIN orders ON customers.customerNumber = orders.customerNumber;

ref : <http://www.mysqltutorial.org/>

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module2.sql**

**\*\* Desc: Nested Queries (DML)**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2019**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

################### Subquery ###################

# query that is nested inside another query such as SELECT, INSERT, UPDATE or DELETE.

# what is the maximum payment

SELECT MAX(amount) FROM payments;

# customer who made the maximum payment

SELECT

customerNumber,

checkNumber,

amount

FROM

payments

WHERE

amount = (SELECT

MAX(amount)

FROM

payments);

# Exercise : Find all payments greater than average payment using sub query. The outer query filters payments that are greater than the average payment returned by the subquery

# HINT : use AVG key word.

SELECT

customerNumber,

checkNumber,

amount

FROM

payments

WHERE

amount > (SELECT

AVG(amount)

FROM

payments);

# customers who have ordered atleast 1 product (IN operator)

SELECT

customerName,

contactFirstName,

contactLastName,

phone

FROM

customers

WHERE

customerNumber IN (SELECT

customerNumber

FROM

orders);

# Exercise : customers who have not ordered any products (NOT IN operator)

################### Subquery with EXIST ###################

# EXISTS or NOT EXISTS operator - returns a Boolean value of TRUE or FALSE.

# customers who have ordered atleast 1 product (EXISTS)

SELECT

customerName,

contactFirstName,

contactLastName,

phone

FROM

customers

WHERE

EXISTS( SELECT

\*

FROM

orders

WHERE

customers.customerNumber = orders.customerNumber);

# Exercise : customers who have not ordered any products (NOT EXISTS)

# customers who have placed at least one order greater than 10K

SELECT

customerName,

contactFirstName,

contactLastName,

phone

FROM

customers

WHERE

EXISTS( SELECT

\*

FROM

orderdetails

INNER JOIN

orders

WHERE

customers.customerNumber = orders.customerNumber

AND orderdetails.orderNumber = orders.orderNumber

AND (priceEach \* quantityOrdered) > 10000);

################ Subquery in FROM clause ###############

# maximum, minimum and average number of items in sale orders

SELECT

MAX(items) AS maxItems,

MIN(items) AS minItems,

AVG(items) AS avgItems

FROM

(SELECT

orderNumber, COUNT(orderNumber) AS items

FROM

orderdetails

GROUP BY orderNumber) AS lineitems;

# list all products whose buy price is greater than all the Harley products - ALL

SELECT

productCode,

productName,

buyPrice

FROM

products

WHERE

buyPrice > ALL (SELECT

buyPrice

FROM

products

WHERE

productName LIKE '%Harley%');

# Exercise : what is the average buy price of a product whose buy price is greater than all the Harley products - ALL

# list all products whose buy price is greater than any of the Harley products - ANY

SELECT

productCode,

productName,

buyPrice

FROM

products

WHERE

buyPrice > ANY (SELECT

buyPrice

FROM

products

WHERE

productName LIKE '%Harley%');

# products whose buy prices are greater than the average buy price of products within their product line

SELECT

productName, productLine, buyPrice

FROM

products AS p

WHERE

buyPrice > (SELECT

AVG(buyPrice)

FROM

products p);

# WHERE

# productline = p.productline);

################ Subquery in SELECT clause ###############

# Used when you wish to compute an aggregate value, but you do not want the aggregate function to apply to the main query.

SELECT

productName,

buyPrice,

(SELECT

MAX(buyPrice)

FROM

products

WHERE

productline = p.productline) AS maxBuyPriceInLine,

(SELECT

AVG(buyPrice)

FROM

products

WHERE

productline = p.productline) AS avgBuyPriceInLine,

(SELECT

MIN(buyPrice)

FROM

products

WHERE

productline = p.productline) AS minBuyPriceInLine

FROM

products p;

#ref : <http://www.mysqltutorial.org/>

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module3.sql**

**\*\* Desc: Views and Indexes**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2019**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

################### CREATE VIEWS ###################

# view containing only customer name and country

CREATE VIEW customerCountry AS

SELECT

customerName, country

FROM

customers;

# describe the structure of view

DESCRIBE customerCountry;

# lists all the tables and views

SHOW FULL TABLES;

# select data from the view

SELECT

\*

FROM

customerCountry;

# view for basket values using orderdetails table

CREATE VIEW basketValue AS

SELECT

orderNumber, SUM(quantityOrdered \* priceEach) AS total

FROM

orderDetails

GROUP BY orderNumber

ORDER BY total DESC;

# select data from the view

SELECT

orderNumber, total

FROM

basketValue;

### create view highestBasketValue from another view basketValue

# Exercise : Show the top 50 orders by basketValue

CREATE VIEW highestBasketValue AS

SELECT

orderNumber, ROUND(total, 2) AS total

FROM

basketValue

ORDER BY total DESC

LIMIT 50;

# select data from the view

SELECT

orderNumber, total

FROM

highestBasketValue;

# create views with joins - order number, customer name, and total sales per order

CREATE VIEW customerOrders AS

SELECT

d.orderNumber,

c.customerName,

c.phone,

SUM(quantityOrdered \* priceEach) total

FROM

orderDetails d

INNER JOIN

orders o ON o.orderNumber = d.orderNumber

INNER JOIN

customers c ON c.customerNumber = o.customerNumber

GROUP BY d.orderNumber

ORDER BY total DESC;

# select data from the view

SELECT

orderNumber, customerName, phone, total

FROM

customerOrders;

# views with subqueries - products whose buy prices are higher than the average price of all products

CREATE VIEW productsAboveAvgPrice AS

SELECT

productCode, productName, buyPrice

FROM

products

WHERE

buyPrice > (SELECT

AVG(buyPrice)

FROM

products)

ORDER BY buyPrice DESC;

# select data from the view

SELECT

productCode, productName, buyPrice

FROM

productsAboveAvgPrice;

################### UPDATE VIEWS ###################

# create a officeInfo view

CREATE VIEW officeInfo AS

SELECT

officeCode, phone, city

FROM

offices;

# Exercise : Select data from the view

# find views that can be updated

SELECT

table\_name, is\_updatable

FROM

information\_schema.views

WHERE

table\_schema = 'classicmodels';

#set the update mode

SET SQL\_SAFE\_UPDATES = 0;

# update phone for the paris office from '+33 14 723 4404' to '+33 14 723 5555'

UPDATE officeInfo

SET

phone = '+33 14 723 5555'

WHERE

officeCode = 4;

# select data from the view

SELECT

officeCode, phone, city

FROM

officeInfo

WHERE

officeCode = 4;

# Exercise : select data from underlying table

SELECT

officeCode, phone, city

FROM

\*

WHERE

# Update view with CHECK OPTION

# ensuring data consistency so that only valid data will be written to the database

# employees whose job title is VP or higher

CREATE OR REPLACE VIEW vps AS

SELECT

\*

FROM

employees

WHERE

jobTitle LIKE '%VP%';

# select data from the view

SELECT

\*

FROM

vps;

# new employee was recruited for IT Manager position. INSERT data using the vps view

INSERT INTO vps(employeeNumber,firstname,lastname,jobtitle,extension,email,officeCode,reportsTo)

values(1703,'Lily','Bush','IT Manager','x9111','lilybush@classicmodelcars.com',1,1002);

# select data from employees table - employee 1703 added

SELECT

\*

FROM

employees

WHERE

employeeNumber = 1703;

# select data from the view - employee 1703 data not seen

SELECT

\*

FROM

vps;

# this time use CHECK OPTION

CREATE OR REPLACE VIEW vps AS

SELECT

\*

FROM

employees

WHERE

jobTitle LIKE '%VP%' WITH CHECK OPTION;

# delete the recently added employee

DELETE FROM employees

where employeeNumber = 1703;

# re-insert the employee - this throws an error

INSERT INTO vps(employeeNumber,firstname,lastname,jobtitle,extension,email,officeCode,reportsTo)

values(1703,'Lily','Bush','IT Manager','x9111','lilybush@classicmodelcars.com',1,1002);

# insert new employee as VP - this works

INSERT INTO vps(employeeNumber,firstname,lastname,jobtitle,extension,email,officeCode,reportsTo)

VALUES(1704,'John','Smith','VP Finance','x9112','johnsmith@classicmodelcars.com',1,1076);

# select data from the view - employee 1704 data seen

SELECT

\*

FROM

vps;

# check on 1704 employee number

SELECT

\*

FROM

employees

where employeeNumber = 1704;

# Managing Views

# Reporting structure

CREATE OR REPLACE VIEW organization AS

SELECT

CONCAT(e.lastname, ' ', e.firstname) AS employee,

CONCAT(m.lastname, ' ', m.firstname) AS manager

FROM

employees AS e

INNER JOIN

employees AS m ON m.employeeNumber = e.reportsTo

ORDER BY Manager;

# select data from the view

SELECT

\*

FROM

organization;

# adding emails to the organization view

ALTER VIEW organization AS

SELECT

CONCAT(E.lastname, ' ', E.firstname) AS employee,

CONCAT(M.lastname, ' ', M.firstname) AS manager,

e.email as employeeEmail,

m.email as managerEmail

FROM

employees AS e

INNER JOIN

employees AS m ON m.employeeNumber = e.reportsTo

ORDER BY manager;

# check to see if email is added for each employee

SELECT

\*

FROM

organization;

# clean up rows added in this module

DELETE FROM employees

WHERE

employeeNumber = 1703 or employeeNumber = 1704;

# Drop views if it already exists

DROP VIEW IF EXISTS customerCountry;

DROP VIEW IF EXISTS basketValue;

DROP VIEW IF EXISTS highestBasketValue;

DROP VIEW IF EXISTS customerOrders;

DROP VIEW IF EXISTS productsAboveAvgPrice;

DROP VIEW IF EXISTS officeInfo;

DROP VIEW IF EXISTS vps;

DROP VIEW IF EXISTS organization;

##################### INDEXES ######################

# performance-tuning method for faster retrieval of records.

# show all indexes in the classicmodels schema

SELECT DISTINCT

TABLE\_NAME, INDEX\_NAME

FROM

INFORMATION\_SCHEMA.STATISTICS

WHERE

TABLE\_SCHEMA = 'classicmodels';

# show all indexes on an existing table

SHOW INDEX FROM customers;

# create a new index on the customers table based on city

CREATE INDEX cityIndex

ON customers (city);

# show all indexes on an existing table

SHOW INDEX FROM customers;

# The index is used when the below query (WHERE clause with city) is executed

SELECT

customerNumber,

contactFirstName,

contactLastName,

city,

state

FROM

CUSTOMERS

WHERE

city = 'Auckland';

# create a new index on the customers table based on city,state

CREATE INDEX cityState\_index

ON customers (city, state);

# check to see indexes were created

SHOW INDEX FROM customers;

ALTER TABLE customers

RENAME INDEX cityState\_index TO cityStateIndex;

# drop the index created

ALTER TABLE customers DROP INDEX cityIndex;

ALTER TABLE customers DROP INDEX cityState\_Index;

# check to see if the index is dropped

SHOW INDEX FROM customers;

#ref : <http://www.mysqltutorial.org/>

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module4.sql**

**\*\* Desc: String, Numeric, Date functions**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2019**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

################### String functions ###################

# TRIM - remove leading and trailing white spaces

SELECT TRIM(' MySQL TRIM function ') AS trimText;

# LTRIM - remove leading spaces on left

SELECT LTRIM(' MySQL TRIM function ') AS ltrimText;

# RTRIM - remove trailing spaces on right

SELECT RTRIM(' MySQL TRIM function ') AS rtrimText;

# CONCAT - combine 1 or many columns

# CONCAT\_WS - concatenate with separator

# Space - add space between the first name and the last name

SELECT

customerNumber,

CONCAT (contactFirstName, space(1), contactLastName ) AS 'contactFullName',

CONCAT\_WS(', ', addressLine1, addressLine2, state, city) AS 'contactAddress'

FROM

customers;

# CONCAT - with TRIM

SELECT

customerNumber,

CONCAT (TRIM(contactFirstName), space(1), TRIM(contactLastName) ) AS 'contactFullName',

CONCAT\_WS(', ', addressLine1, addressLine2, state, city) AS 'contactAddress'

FROM

customers;

# UPPER - converts to uppercase / LOWER - converts to lowercase

SELECT

customerNumber,

country,

UPPER(country) AS countryUpper,

city,

LOWER(city) AS cityLower

FROM

customers;

# INSTR - search the set of characters ( case sensitive ) and indicate the position

SELECT

customerNumber,

addressLine1,

INSTR(addressLine1, 'St.') AS positionOfMatch

FROM

customers

WHERE

INSTR(addressLine1, 'St.') > 0;

# LPAD - pads leading spaces / RPAD - pads trailing spaces

SELECT

customerNumber,

addressLine1, RPAD(addressLine1, 20, '\*') AS rapdAddressLine1,

addressLine2, LPAD(addressLine2, 20, '\*') AS lapdAddressLine2

FROM

customers;

# REPLACE

# translate "St." to "Street" in the addressLine1 column

SELECT

customerNumber,

addressLine1,

REPLACE(addressLine1, 'St.', 'Street') AS newAddressLine1

FROM

customers

WHERE

country = 'USA';

######### SUBSTRING - extracts part of the string ##########

SELECT

customerNumber,

addressLine1,

# extract the street number

substring(addressLine1,1,4) AS streetNumber,

# extract all characters from position 5 till end of string

substring(addressLine1,5) AS streetName1,

substring(addressLine1 FROM 5) AS streetName2,

# extract all characters from position -10 for length 6

substring(addressLine1, -10, 6) AS streetName3,

substring(addressLine1 FROM -10 for 6) AS streetName4

FROM

customers

WHERE

country = 'USA';

# CHAR\_LENGTH - length of string (number of characters including spaces )

# street adress - removing last 3 characters

SELECT

customerNumber,

addressLine1,

SUBSTRING(addressLine1,

1,

CHAR\_LENGTH(addressLine1) - 3) AS streetName

FROM

customers

WHERE

country = 'USA';

# COALESCE - returns the first non-NULL value of a list

# returns the first non-NULL value from the list ( City, state ) or NULL

SELECT

customerNumber,

state,

country,

COALESCE(state, country) AS stateOrCountry

FROM

customers;

# RegEx - special string that describes a search pattern

# find out products whose last name starts with character A, B or C.

SELECT

productname

FROM

products

WHERE

productname REGEXP '^(A|B|C)'

ORDER BY productname;

# find the product whose name ends with f

SELECT

productname

FROM

products

WHERE

productname REGEXP 'f$';

# find products whose name contains exactly 10 characters

SELECT

productname

FROM

products

WHERE

productname REGEXP '^.{10}$';

################### Numeric functions ###################

# apply FLOOR, CEIL, ROUND, TRUNCATE to buyPrice column

SELECT

productName,

buyPrice,

FLOOR(buyPrice) AS floorBuyPrice,

CEIL(buyPrice) AS ceilBuyPrice,

ROUND(buyPrice) AS roundBuyPrice,

ROUND(buyPrice,3) AS roundBuyPrice,

TRUNCATE(buyPrice,1) AS truncateBuyPrice

FROM

products

WHERE

ProductName LIKE '%Harley%';

# Apply POWER, SQRT to MSRP column

SELECT

productName,

buyPrice,

MSRP,

POWER(MSRP,2) AS power2MSRP,

POWER(MSRP,-2) AS powerNeg2MSRP,

POWER(MSRP,1/2) AS powerReciprocal2MSRP,

SQRT(MSRP) AS squareRootMSRP

FROM

products

WHERE

ProductName LIKE '%Harley%';

# apply ABS and SIGN to the difference between MSRP and buyPrice

SELECT

productName,

buyPrice,

MSRP,

(buyPrice - MSRP) AS buyPriceMSRPDiff,

ABS(buyPrice - MSRP) AS absBuyPriceMSRPDiff,

SIGN(buyPrice - MSRP) AS signBuyPriceMSRPDiff,

SIGN(MSRP - buyPrice ) AS signMSRPBuyPriceDiff

FROM

products

WHERE

ProductName LIKE '%Harley%';

################### Date functions ###################

# current system time

SELECT NOW() AS currentTime;

# date part of the DATETIME and CURDATE

SELECT DATE(NOW()), CURDATE();

# format the current date using DATE\_FORMAT function

SELECT

CURDATE() AS 'default',

DATE\_FORMAT(CURDATE(), '%m-%d-%y') AS '%m-%d-%y',

DATE\_FORMAT(CURDATE(), '%d/%m/%Y') AS '%d/%m/%Y',

DATE\_FORMAT(CURDATE(), '%M-%d-%Y') AS '%M-%d-%Y',

DATE\_FORMAT(CURDATE(), '%D-%M-%y') AS '%D-%M-%y';

SELECT

orderDate,

shippedDate,

DATEDIFF(shippedDate, orderDate) AS numberOfDays

# DATEDIFF ('06-03-2016' AS DATE , '06-03-2014' AS DATE) AS numberOfDays

FROM

orders

ORDER BY numberOfDays DESC;

select \* from orders;

# Add a value from a date value using DATE\_ADD function

SELECT

orderDate,

requiredDate,

DATE\_ADD(orderDate, INTERVAL 1 DAY) AS dayAfterOrderDate,

DATE\_ADD(orderDate, INTERVAL 1 WEEK) AS weekAfterOrderDate,

DATE\_ADD(orderDate, INTERVAL 1 MONTH) AS monthAfterOrderDate,

DATE\_ADD(orderDate, INTERVAL 1 YEAR) AS yearAfterOrderDate

FROM

orders;

# subtract a value from a date value using DATE\_SUB function

SELECT

orderDate,

shippedDate,

DATE\_SUB(shippedDate, INTERVAL 1 DAY) AS dayBeforeShippedDate,

DATE\_SUB(shippedDate, INTERVAL 1 WEEK) AS weekBeforeShippedDate,

DATE\_SUB(shippedDate, INTERVAL 1 MONTH) AS monthBeforeShippedDate,

DATE\_SUB(shippedDate, INTERVAL 1 YEAR) AS yearBeforeShippedDate

FROM

orders;

################### Conversions functions ###################

# Conversion functions are used to convert a data type into another data type

# MySQL converts a string into an integer implicitly before calculation:

SELECT (1 + '1')/2;

# Explicit cast

SELECT (1 + CAST('1' AS UNSIGNED))/2 ;

# phone is varchar(50) and when cast to unsigned integer we have erroneous results

DESCRIBE offices;

SELECT

phone,

CAST(phone AS UNSIGNED) AS phoneAsInteger,

CAST(phone AS CHAR) AS phoneAsChar

FROM

offices;

# orders whose required dates are in January 2015.

# The data type of the requireDate column is DATE, therefore, MySQL has to convert the literal strings: '2015-01-01' and '2015-01-31' into TIMESTAMP values before evaluating the WHERE condition.

SELECT

orderNumber, requiredDate

FROM

orders

WHERE

requiredDate BETWEEN '2015-01-01' AND '2015-01-31';

# To be safe, you can use CAST() function to explicitly convert a string into a TIMESTAMP value as follows:

SELECT

orderNumber, requiredDate

FROM

orders

WHERE

requiredDate BETWEEN CAST('2015-01-01' AS DATETIME) AND CAST('2015-01-31' AS DATETIME);

# converts DOUBLE values into CHAR values and uses the results as the arguments to CONCAT function

SELECT

productName,

CONCAT('BuyPrice, MSRP : (',

CAST(buyprice AS CHAR),

',',

CAST(msrp AS CHAR),

')') AS prices

FROM

products;

#ref : <http://www.mysqltutorial.org/>

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module7.sql**

**\*\* Desc: MySQL Partitioning**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2018**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

################### Partitions ###################

# Partitioning (a database design technique) improves performance, manageability and reduces the cost of storing large datasets

################### RANGE Partitioning ###################

# specify various ranges for which data is assigned. Ranges should be contiguous but not overlapping, and are defined using the VALUES LESS THAN operator

CREATE TABLE customer\_rangepartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL,

PRIMARY KEY (`customerNumber`)

) PARTITION BY RANGE (customerNumber)

(PARTITION p0 VALUES LESS THAN (200) ,

PARTITION p1 VALUES LESS THAN (300) ,

PARTITION p2 VALUES LESS THAN (400) ,

PARTITION p3 VALUES LESS THAN (500));

# Check to see if all partitions are created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# Insert data into the partitioned table from the original customers table

INSERT INTO customer\_rangepartition (

SELECT \* FROM customers);

# Check to see if all partitions have data

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# validate if the partitioning is working ( P0 partition has 32 customers )

Select count(\*) from customers where customerNumber < 200;

# truncate data from partition 0

ALTER TABLE customer\_rangepartition TRUNCATE PARTITION p0;

# Check to see if data from partition 0 is truncated

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# Drop Partition 0

ALTER TABLE customer\_rangepartition DROP PARTITION p0;

# Check to see if partition 0 is dropped

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# Alter table to data refresh by adding Partitions

ALTER TABLE customer\_rangepartition ADD PARTITION (PARTITION p4 VALUES LESS THAN (600));

# Add Partition on the fly

SELECT

PARTITION\_NAME, TABLE\_ROWS

FROM

INFORMATION\_SCHEMA.PARTITIONS

WHERE

TABLE\_NAME = 'customer\_rangepartition';

SELECT

\*

FROM

customer\_rangepartition

WHERE

contactLastName LIKE 'S%';

SELECT

\*

FROM

customer\_rangepartition PARTITION (p0 , p2)

WHERE

contactLastName LIKE 'S%';

# Drop customer\_partitioned table

drop table customer\_rangepartition;

################### List Partitioning ###################

# each partition is defined and selected based on the membership of a column value in one of a set of value lists, rather than in one of a set of contiguous ranges of values

CREATE TABLE customers\_listpartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL

) PARTITION BY LIST COLUMNS (country)

(PARTITION pcountry\_a VALUES IN ('Australia' , 'Austria' , 'Belgium', 'Canada', 'Denmark' ) ,

PARTITION pcountry\_b VALUES IN ('Finland' , 'France', 'Germany', 'Hong Kong', 'Ireland','Israel', 'Italy' ) ,

PARTITION pcountry\_c VALUES IN ('Japan' , 'Netherlands', 'New Zealand', 'Norway', 'Philippines', 'Poland'),

PARTITION pcountry\_d VALUES IN ('Portugal' , 'Russia', 'Singapore', 'South Africa', 'Spain', 'Sweden', 'Switzerland','UK','USA'));

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customers\_listpartition';

# insert data from customers table

INSERT INTO customers\_listpartition (

SELECT \* FROM customers);

# check to see the data into partitions

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customers\_listpartition';

# validate results

SELECT

COUNT(\*)

FROM

customers

WHERE

country IN ('Australia' , 'Austria',

'Belgium',

'Canada',

'Denmark');

# Drop customer\_partitioned table

DROP TABLE customers\_listpartition;

################### Hash Partitioning ###################

# Distribute data among a predefined number of partitions on a column value or expression based on a column value.

CREATE TABLE customer\_hashpartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL,

PRIMARY KEY (`customerNumber`)

) PARTITION BY HASH (customerNumber) PARTITIONS 8;

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_hashpartition';

# insert data from customers table

INSERT INTO customer\_hashpartition (

SELECT \* FROM customers);

# check to see the data into partitions

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_hashpartition';

SELECT

\*

FROM

customer\_hashpartition PARTITION (p0);

# Drop customer\_partitioned table

DROP TABLE customer\_hashpartition;

################### Key Partitioning ###################

# special form of HASH partition, where the hashing function for key partitioning is supplied by the MySQL server

CREATE TABLE customer\_keypartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL,

PRIMARY KEY (`customerNumber`)

) PARTITION BY KEY(customerNumber)

PARTITIONS 8;

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_keypartition';

# insert data from customers table

INSERT INTO customer\_keypartition (

SELECT \* FROM customers);

# check to see the data into partitions

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_keypartition';

SELECT

\*

FROM

CUSTOMER\_KEYPARTITION PARTITION (p0);

# Drop customer\_partitioned table

DROP TABLE customer\_keypartition;

################### Sub-Partitioning ###################

# method to divide each partition further in a partitioned table

CREATE TABLE orders\_subpartition (

`orderNumber` int(11) NOT NULL,

`orderDate` date NOT NULL,

`requiredDate` date NOT NULL,

`shippedDate` date DEFAULT NULL,

`status` varchar(15) NOT NULL,

`comments` text,

`customerNumber` int(11) NOT NULL

) PARTITION BY RANGE (YEAR(orderDate))

SUBPARTITION BY HASH (TO\_DAYS(orderDate))

SUBPARTITIONS 3 (

PARTITION p0 VALUES LESS THAN (2015) ,

PARTITION p1 VALUES LESS THAN (2016),

PARTITION p2 VALUES LESS THAN MAXVALUE);

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition';

# insert data from customers table

INSERT INTO orders\_subpartition (

SELECT \* FROM orders);

# check to see the data into partitions

SELECT PARTITION\_NAME, SUBPARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition';

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition';

SELECT PARTITION\_NAME FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition' GROUP BY PARTITION\_NAME;

# Drop customer\_partitioned table

DROP TABLE orders\_subpartition;

---- END ----

# References

# http://www.w3resource.com/mysql/mysql-partition.php

# http://www.mysqltutorial.org/

# Use case for partitioning

# data is collected on a daily basis from a set of 124 grocery stores. Each days data was completely distinct from every other days. We partitioned the data on the date. This allowed us to have faster searches because oracle can use partitioned indexes and quickly eliminate all of the non-relevant days. This also allows for much easier backup operations because you can work in just the new partitions. Also after 5 years of data we needed to get rid of an entire days data. You can "drop" or eliminate an entire partition at a time instead of deleting rows. So getting rid of old data was a snap.

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module6.sql**

**\*\* Desc: SQL Stored Procedures and Triggers**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2019**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

################### Stored procedures ###################

# Segment of declarative SQL statements stored inside the database catalog. A stored procedure can be invoked by triggers, other stored procedures, and applications such as Java, Python, PHP, etc.

# create a stored procedure that will pull all products from the products table

DELIMITER //

CREATE PROCEDURE GetAllProducts()

BEGIN

SELECT \* FROM products;

END //

DELIMITER ;

# check to see if the stored procedure was created

SHOW PROCEDURE STATUS WHERE db = 'classicmodels';

SHOW CREATE PROCEDURE GetAllProducts;

# execute ( call ) the stored procedure and make sure we it aligns with the 110 records from the products table.

CALL GetAllProducts();

# validate

SELECT

\*

FROM

products;

# Use of parameters within stored procedures

# Create a stored procedure that will take customer number as input and will provide details based on status

DELIMITER //

CREATE PROCEDURE get\_order\_by\_cust(

IN cust\_no INT,

OUT shipped INT,

OUT canceled INT,

OUT resolved INT,

OUT disputed INT)

BEGIN

-- shipped

SELECT

COUNT(\*)

INTO shipped FROM

orders

WHERE

customerNumber = cust\_no

AND status = 'Shipped';

-- canceled

SELECT

COUNT(\*)

INTO canceled FROM

orders

WHERE

customerNumber = cust\_no

AND status = 'Canceled';

-- resolved

SELECT

COUNT(\*)

INTO resolved FROM

orders

WHERE

customerNumber = cust\_no

AND status = 'Resolved';

-- disputed

SELECT

COUNT(\*)

INTO disputed FROM

orders

WHERE

customerNumber = cust\_no

AND status = 'Disputed';

END //

DELIMITER ;

# find the status of all orders for customer 141

CALL get\_order\_by\_cust(141,@shipped,@canceled,@resolved,@disputed);

SELECT @shipped,@canceled,@resolved,@disputed;

SELECT

\*

FROM

customers

WHERE

customerNumber = 141;

# Drop stored procedure

DROP PROCEDURE GetAllProducts;

DROP PROCEDURE get\_order\_by\_cust;

################### Triggers ###################

# stored program executed automatically to respond to a specific event e.g., insert, update or delete occurred in a table. The main difference between a trigger and a stored procedure is that a trigger is called automatically when a data modification event is made against a table whereas a stored procedure must be called explicitly

# Log old price in a separate table named price\_logs when there is a change in the price of a product (column MSRP )

# create a new price\_logs table

CREATE TABLE price\_logs (

id INT(11) NOT NULL AUTO\_INCREMENT,

product\_code VARCHAR(15) NOT NULL,

price DOUBLE NOT NULL,

updated\_at TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (id),

KEY product\_code (product\_code),

CONSTRAINT price\_logs\_ibfk\_1 FOREIGN KEY (product\_code)

REFERENCES products (productCode)

ON DELETE CASCADE ON UPDATE CASCADE

);

# check the price logs table to make sure there are no records

SELECT \* FROM price\_logs;

# create a trigger that activates when the BEFORE UPDATE event of the products table occurs

DELIMITER $$

CREATE TRIGGER before\_products\_update

BEFORE UPDATE ON products

FOR EACH ROW

BEGIN

INSERT INTO price\_logs(product\_code,price)

VALUES(old.productCode,old.msrp);

END $$

DELIMITER ;

# Check to make sure the trigger is created

SHOW TRIGGERS FROM classicmodels

WHERE `table` = 'products';

# change the price of a product and query the price\_logs table using the following UPDATE statement:

UPDATE products

SET

msrp = 95.1

WHERE

productCode = 'S10\_1678';

# check the price logs table to make sure the old value is saved

SELECT

\*

FROM

price\_logs;

# We want to see not only the old price and when it was changed but also who changed it.

# Add additional columns to the price\_logs table. However, for the purpose of multiple triggers demonstration, we will create a new table user\_change\_logs to store the data of users who made the changes.

# create a user\_change\_logs table

CREATE TABLE user\_change\_logs (

id INT(11) NOT NULL AUTO\_INCREMENT,

product\_code VARCHAR(15) DEFAULT NULL,

updated\_at TIMESTAMP NOT NULL DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

updated\_by VARCHAR(30) NOT NULL,

PRIMARY KEY (id),

KEY product\_code (product\_code),

CONSTRAINT user\_change\_logs\_ibfk\_1 FOREIGN KEY (product\_code)

REFERENCES products (productCode)

ON DELETE CASCADE ON UPDATE CASCADE

);

# Now, we create a second trigger that activates on the BEFORE UPDATE event of the products table. This trigger will update the user\_change\_logs table with the data of the user who made the changes. It is activated after the before\_products\_update trigger.

DELIMITER $$

CREATE TRIGGER before\_products\_update\_2

BEFORE UPDATE ON products

FOR EACH ROW FOLLOWS before\_products\_update

BEGIN

INSERT INTO user\_change\_logs(product\_code,updated\_by)

VALUES(old.productCode,user());

END$$

DELIMITER ;

# First, we update the prices of the product using the UPDATE statement as follows:

UPDATE products

SET

msrp = 95.3

WHERE

productCode = 'S10\_1678';

# Second, we query the data from both price\_logs and user\_change\_logs tables:

SELECT \* FROM price\_logs;

SELECT \* FROM user\_change\_logs;

# Information On Triggers Order. The first one does not show order while the second query shows ordering

SHOW TRIGGERS FROM classicmodels;

SELECT

# trigger\_name, action\_order

\*

FROM

information\_schema.triggers

WHERE

trigger\_schema = 'classicmodels'

ORDER BY event\_object\_table ,

action\_timing ,

event\_manipulation;

# Drop all databse objects ( triggers & tables )

DROP TABLE IF EXISTS `price\_logs`;

DROP TABLE IF EXISTS `user\_change\_logs`;

# Note: Triggers don't exist at the table level. They are database level objects that are just associated with tables. There is no means to delete them by related table.

DROP TRIGGER IF EXISTS `before\_products\_update`;

DROP TRIGGER IF EXISTS `before\_products\_update\_2`;

#ref : <http://www.mysqltutorial.org/>

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module7.sql**

**\*\* Desc: MySQL Partitioning**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2018**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

################### Partitions ###################

# Partitioning (a database design technique) improves performance, manageability and reduces the cost of storing large datasets

################### RANGE Partitioning ###################

# specify various ranges for which data is assigned. Ranges should be contiguous but not overlapping, and are defined using the VALUES LESS THAN operator

CREATE TABLE customer\_rangepartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL,

PRIMARY KEY (`customerNumber`)

) PARTITION BY RANGE (customerNumber)

(PARTITION p0 VALUES LESS THAN (200) ,

PARTITION p1 VALUES LESS THAN (300) ,

PARTITION p2 VALUES LESS THAN (400) ,

PARTITION p3 VALUES LESS THAN (500));

# Check to see if all partitions are created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# Insert data into the partitioned table from the original customers table

INSERT INTO customer\_rangepartition (

SELECT \* FROM customers);

# Check to see if all partitions have data

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# validate if the partitioning is working ( P0 partition has 32 customers )

Select count(\*) from customers where customerNumber < 200;

# truncate data from partition 0

ALTER TABLE customer\_rangepartition TRUNCATE PARTITION p0;

# Check to see if data from partition 0 is truncated

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# Drop Partition 0

ALTER TABLE customer\_rangepartition DROP PARTITION p0;

# Check to see if partition 0 is dropped

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_rangepartition';

# Alter table to data refresh by adding Partitions

ALTER TABLE customer\_rangepartition ADD PARTITION (PARTITION p4 VALUES LESS THAN (600));

# Add Partition on the fly

SELECT

PARTITION\_NAME, TABLE\_ROWS

FROM

INFORMATION\_SCHEMA.PARTITIONS

WHERE

TABLE\_NAME = 'customer\_rangepartition';

SELECT

\*

FROM

customer\_rangepartition

WHERE

contactLastName LIKE 'S%';

SELECT

\*

FROM

customer\_rangepartition PARTITION (p0 , p2)

WHERE

contactLastName LIKE 'S%';

# Drop customer\_partitioned table

drop table customer\_rangepartition;

################### List Partitioning ###################

# each partition is defined and selected based on the membership of a column value in one of a set of value lists, rather than in one of a set of contiguous ranges of values

CREATE TABLE customers\_listpartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL

) PARTITION BY LIST COLUMNS (country)

(PARTITION pcountry\_a VALUES IN ('Australia' , 'Austria' , 'Belgium', 'Canada', 'Denmark' ) ,

PARTITION pcountry\_b VALUES IN ('Finland' , 'France', 'Germany', 'Hong Kong', 'Ireland','Israel', 'Italy' ) ,

PARTITION pcountry\_c VALUES IN ('Japan' , 'Netherlands', 'New Zealand', 'Norway', 'Philippines', 'Poland'),

PARTITION pcountry\_d VALUES IN ('Portugal' , 'Russia', 'Singapore', 'South Africa', 'Spain', 'Sweden', 'Switzerland','UK','USA'));

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customers\_listpartition';

# insert data from customers table

INSERT INTO customers\_listpartition (

SELECT \* FROM customers);

# check to see the data into partitions

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customers\_listpartition';

# validate results

SELECT

COUNT(\*)

FROM

customers

WHERE

country IN ('Australia' , 'Austria',

'Belgium',

'Canada',

'Denmark');

# Drop customer\_partitioned table

DROP TABLE customers\_listpartition;

################### Hash Partitioning ###################

# Distribute data among a predefined number of partitions on a column value or expression based on a column value.

CREATE TABLE customer\_hashpartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL,

PRIMARY KEY (`customerNumber`)

) PARTITION BY HASH (customerNumber) PARTITIONS 8;

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_hashpartition';

# insert data from customers table

INSERT INTO customer\_hashpartition (

SELECT \* FROM customers);

# check to see the data into partitions

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_hashpartition';

SELECT

\*

FROM

customer\_hashpartition PARTITION (p0);

# Drop customer\_partitioned table

DROP TABLE customer\_hashpartition;

################### Key Partitioning ###################

# special form of HASH partition, where the hashing function for key partitioning is supplied by the MySQL server

CREATE TABLE customer\_keypartition (

`customerNumber` int(11) NOT NULL,

`customerName` varchar(50) NOT NULL,

`contactLastName` varchar(50) NOT NULL,

`contactFirstName` varchar(50) NOT NULL,

`phone` varchar(50) NOT NULL,

`addressLine1` varchar(50) NOT NULL,

`addressLine2` varchar(50) DEFAULT NULL,

`city` varchar(50) NOT NULL,

`state` varchar(50) DEFAULT NULL,

`postalCode` varchar(15) DEFAULT NULL,

`country` varchar(50) NOT NULL,

`salesRepEmployeeNumber` int(11) DEFAULT NULL,

`creditLimit` decimal(10,2) DEFAULT NULL,

PRIMARY KEY (`customerNumber`)

) PARTITION BY KEY(customerNumber)

PARTITIONS 8;

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_keypartition';

# insert data from customers table

INSERT INTO customer\_keypartition (

SELECT \* FROM customers);

# check to see the data into partitions

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='customer\_keypartition';

SELECT

\*

FROM

CUSTOMER\_KEYPARTITION PARTITION (p0);

# Drop customer\_partitioned table

DROP TABLE customer\_keypartition;

################### Sub-Partitioning ###################

# method to divide each partition further in a partitioned table

CREATE TABLE orders\_subpartition (

`orderNumber` int(11) NOT NULL,

`orderDate` date NOT NULL,

`requiredDate` date NOT NULL,

`shippedDate` date DEFAULT NULL,

`status` varchar(15) NOT NULL,

`comments` text,

`customerNumber` int(11) NOT NULL

) PARTITION BY RANGE (YEAR(orderDate))

SUBPARTITION BY HASH (TO\_DAYS(orderDate))

SUBPARTITIONS 3 (

PARTITION p0 VALUES LESS THAN (2015) ,

PARTITION p1 VALUES LESS THAN (2016),

PARTITION p2 VALUES LESS THAN MAXVALUE);

# Check to see if partitions were created

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition';

# insert data from customers table

INSERT INTO orders\_subpartition (

SELECT \* FROM orders);

# check to see the data into partitions

SELECT PARTITION\_NAME, SUBPARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition';

SELECT PARTITION\_NAME, TABLE\_ROWS FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition';

SELECT PARTITION\_NAME FROM INFORMATION\_SCHEMA.PARTITIONS WHERE TABLE\_NAME='orders\_subpartition' GROUP BY PARTITION\_NAME;

# Drop customer\_partitioned table

DROP TABLE orders\_subpartition;

---- END ----

# References

# http://www.w3resource.com/mysql/mysql-partition.php

# http://www.mysqltutorial.org/

# Use case for partitioning

# data is collected on a daily basis from a set of 124 grocery stores. Each days data was completely distinct from every other days. We partitioned the data on the date. This allowed us to have faster searches because oracle can use partitioned indexes and quickly eliminate all of the non-relevant days. This also allows for much easier backup operations because you can work in just the new partitions. Also after 5 years of data we needed to get rid of an entire days data. You can "drop" or eliminate an entire partition at a time instead of deleting rows. So getting rid of old data was a snap.

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\* DATA ENGINEERING PLATFORMS (MSCA 31012)**

**\*\* File: Session4-Module8.sql**

**\*\* Desc: Full-Text Search & JSON Support**

**\*\* Auth: Shreenidhi Bharadwaj**

**\*\* Date: 1/27/2018**

**\*\* Ref : http://www.mysqltutorial.org/**

**\*\* ALL RIGHTS RESERVED | DO NOT DISTRIBUTE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

# select database

USE classicmodels;

################### Introduction to MySQL JSON data type ###################

# Track visitors & actions on website. Some visitors may just view the pages and other may view the pages and buy the products.

# Insert data onto the events table

# id - uniquely identifies the event.

# event\_name - An event also has a name e.g., pageview, purchase, etc.,

# visitor - used to store the visitor information.

# properties - store properties of an event

# browser - specification of the browser that visitors use to browse the website.

CREATE TABLE events(

id int auto\_increment primary key,

event\_name varchar(255),

visitor varchar(255),

properties json,

browser json

);

# Insert sample event data

INSERT INTO events(event\_name, visitor,properties, browser)

VALUES (

'pageview',

'1',

'{ "page": "/" }',

'{ "name": "Safari", "os": "Mac", "resolution": { "x": 1920, "y": 1080 } }'

),

('pageview',

'2',

'{ "page": "/contact" }',

'{ "name": "Firefox", "os": "Windows", "resolution": { "x": 2560, "y": 1600 } }'

),

(

'pageview',

'1',

'{ "page": "/products" }',

'{ "name": "Safari", "os": "Mac", "resolution": { "x": 1920, "y": 1080 } }'

),

(

'purchase',

'3',

'{ "amount": 200 }',

'{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1600, "y": 900 } }'

),

(

'purchase',

'4',

'{ "amount": 150 }',

'{ "name": "Firefox", "os": "Windows", "resolution": { "x": 1280, "y": 800 } }'

),

(

'purchase',

'4',

'{ "amount": 500 }',

'{ "name": "Chrome", "os": "Windows", "resolution": { "x": 1680, "y": 1050 } }'

);

# validate

SELECT

\*

FROM

events;

# Which browser was used the most ? pull the data from the JSON blob

SELECT

id,

browser->'$.name' browser

FROM

events;

# to remove the quotes from the string use the inline path operator (->>)

SELECT

id,

browser->>'$.name' browser

FROM

events;

# Find the browser usage

SELECT

browser->>'$.name' browser,

count(browser)

FROM

events

GROUP BY

browser->>'$.name';

# calculate the total revenue by the visitor

SELECT

visitor,

SUM(properties->>'$.amount') revenue

FROM

events

WHERE

properties->>'$.amount' > 0

GROUP BY

visitor;

################### Introduction to MySQL Full-Text Search ###################

# Natural Language Full-Text Searches

# Rows or documents that are relevant to the free-text natural human language query

# For natural-language full-text searches, it is a requirement that the columns named in the MATCH() function be the same columns included in FULLTEXT index in your table.

# https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html

ALTER TABLE products ADD FULLTEXT(productline);

ALTER TABLE products ADD FULLTEXT index\_name(productName);

# Search for products whose product lines contain the term Classic

SELECT

productName, productline

FROM

products

WHERE

MATCH (productline) AGAINST ('Classic' );

# search for product whose product line contains Classic or Vintage term

SELECT

productName, productline

FROM

products

WHERE

MATCH (productline) AGAINST ('Classic,Vintage' IN NATURAL LANGUAGE MODE);

# Search for products whose names contain Ford and/or 1932 using the following query:

SELECT

productName, productline

FROM

products

WHERE

MATCH (productName) AGAINST ('1932,Ford' );

# Boolean Full-Text Searches : Perform a full-text search based on very complex queries in the Boolean mode along with Boolean operators.

# Search for a product whose product name contains the Truck word.

SELECT

productName, productline

FROM

products

WHERE

MATCH (productName) AGAINST ('Truck' IN BOOLEAN MODE);

# find the product whose product names contain the Truck word but not any rows that contain Pickup

# use the exclude Boolean operator ( - )

SELECT

productName, productline

FROM

products

WHERE

MATCH (productName) AGAINST ('Truck -Pickup' IN BOOLEAN MODE)