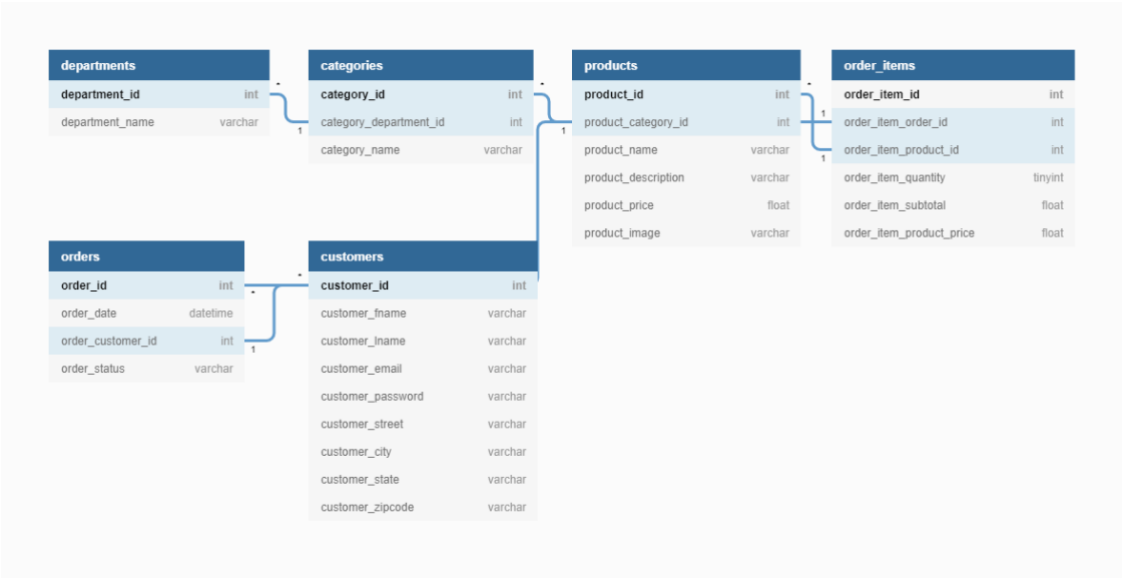
/Users/hencilpeter/hencil-data/certification/current

1. create an account in google cloud platform
2. create a new project in google cloud : CCA-175-cluster
3. create a spark cluster using Dataproc
  Left navicatin -> data proc -> clusters
  Create Cluster -> enter cluster name "cca-175-cluster"
  cluster mode : single node (1 master 0 worker)
  machine type -> n1 standard 8
  click compressed gateway
  image -> 1.5 (Debian 10, Hadoop 2.19, Spark 2.4)
 click "Add Initialization action"
 click "Create"

1. creating bucket in google cloud
   -> go to : https://console.cloud.google.com/
   -> on the left side, select storage -> Browser
   > click on "create bucket" and enter name of the bucket as "cca-175-practice-bucket-hencil" (cca-175-practice-hencil) and accept all the default properties and click "Create "
2. upload the local folder into bucket
     -> click on upload folder and select folder : retial_db
3. mount the bucket as a file system (follow instructions from doc)

some changes
  [[[[use correct bucket name]]]]
gcsfuse --implicit-dirs cca-175-practice-hencil ~/retail_db_dataset
  --> go to VM instances
 Create VM Instance in google cloud
Compute -> Compute Engines -> VM instances



```
hencilpeter@cca-175-cluster-m:~$ spark-shell
scala> spark
```
Clear screen : Ctrl + L

hdfs dfs -ls /user/spark/dataset/retail_db | sed '1d;s/ */ /g'| cut -d\ -f8;

#	Operation	Command	Remark/Description
	CSV FILE OPERATIONS		
1	Read/load csv file and create DF	val catDF = spark.read.format("csv").load("/user/spark/dataset/retail_db/categories")	
2	Show first 5 records	catDF.show(5)	
3	Read/load csv file and create DF - another approach	val catDF = spark.read.csv("/user/spark/dataset/retail_db/categories")	
4	Read and give column names	val catDF = spark.read.csv("/user/spark/dataset/retail_db/categories"). | toDF("category_id", "category_department_id", "category_name")	
5	Selecting columns	catDF.select("category_id", "category_name").show(5)	
6	Loading csv file with header	val catDF = spark.read.option("header", true). | csv("/user/spark/dataset/retail_db/categories-header")	
7	Import spark sql	import org.apache.spark.sql.types._ import org.apache.spark.sql._	
8	Creating schema	val mySchema = StructType(Array(| StructField("cat_id", IntegerType, true), | StructField("cat_dep_id", IntegerType, true), | StructField("cat_name", StringType, true)))	
9	Load data with Schema details	val catDF = spark.read.option("header", true). | schema(mySchema). | csv("/user/spark/dataset/retail_db/categories-header")	
10	Print schema	catDF.printSchema	

11	Write CSV content in hdfs	catDF.write.option("header", true).csv("/user/spark/dataset/output/cat-header")	
12	Verify the written content	spark.read.option("header", true).csv("/user/spark/dataset/output/cat-header").show(3)	
13	Compress and Write	catDF.write.option("header", true).mode(SaveMode.Overwrite).option("compression","snappy").csv("/user/spark/dataset/output/cat-header") --verify the file after compressed write hdfs dfs -ls /user/spark/dataset/output/cat-header hdfs dfs -cat /user/spark/dataset/output/cat-header/*	
	JSON FILE OPERATIONS		
	Read/load JSON (single lines file)	val custData = spark.read.format("json").load("/user/spark/dataset/retail_db/customers-json") val custData = spark.read.json("/user/spark/dataset/retail_db/customers-json") custData.select("customer_city","customer_email","customer_id").show(3) --check the file content in hdfs hdfs dfs -tail /user/spark/dataset/retail_db/customers-json/part-m-00000.json	
	Read/load JSON (multilines file)	val multilineJson = spark.read.option(**"multiline",true**).json("/user/spark/dataset/retail_db/customers-multiline-json") --check the file in hdfs hdfs dfs -tail /user/spark/dataset/retail_db/customers-multiline-json/part-m-00000.json	
	Write multiline json	multilineJson.write.json("/user/spark/dataset/output/multiline-json-op") --check in hdfs hdfs dfs -tail /user/spark/dataset/output/multiline-json-op/part-00000-87ae1ec1-168e-4f7a-8a0d-fe2fb9a17c0e-c000.json	
	Write custData with compression	val custJson = spark.read.json("/user/spark/dataset/retail_db/customers-json") custJson.write.option("compression", "gzip").json("/user/spark/dataset/output/cust-data-gzip")	

		--check the file in hdfs hdfs dfs -ls /user/spark/dataset/output/cust-data-gzip result : /user/spark/dataset/output/cust-data-gzip/part-00000-2e7bd925-c4dd-4ea7-b3b0-a3602bfe4e2d-c000.json.**gz** hdfs dfs -tail /user/spark/dataset/output/cust-data-gzip/part-00000-2e7bd925-c4dd-4ea7-b3b0-a3602bfe4e2d-c000.json.gz	
	PARQUET FILE OPERATIONS		
	Read/load parquet format file	val orders = spark.read.parquet("/user/spark/dataset/retail_db/orders_parquet")	
	Filter order status = "PENDING_PAYMENT"	val orderFiltered = orders.filter($"order_status" === "PENDING_PAYMENT")	
	Select two columns	val filteredOrderIdAndStatus = orderFiltered.select("order_id", "order_status")	
	Write pending orders	pendingPaymentOrders.write.parquet("/user/spark/dataset/output/order-pending-payment") -- Default compression is snappy hdfs dfs -ls /user/spark/dataset/output/order-pending-payment /user/spark/dataset/output/order-pending-payment/part-00000-6e6107cb-9d56-49b4-b1a7-0b3ab37600e8-c000.snappy.parquet	
	View the parquet file content	--switch to the folder where parquet jar is copied and execute the below hadoop jar parquet-tools-1.9.0.jar cat --json /user/spark/dataset/output/order-pending-payment/part-00000-6e6107cb-9d56-49b4-b1a7-0b3ab37600e8-c000.snappy.parquet	
	Write parquet file with different compression (as default is snappy)	pendingPaymentOrders.write.option("compression", "gzip").parquet("/user/spark/dataset/output/order-pending-payment-gzip") -verify the new file hdfs dfs -ls /user/spark/dataset/output/order-pending-payment-gzip Result: /user/spark/dataset/output/order-pending-payment-gzip/part-00000-16d156e8-bd16-46ee-966d-d435f94d89f0-c000.gz.parquet --view the new content	

		hadoop jar parquet-tools-1.9.0.jar cat --json /user/spark/dataset/output/order-pending-payment-gzip/part-00000-16d156e8-bd16-46ee-966d-d435f94d89f0-c000.gz.parquet	
	AVRO FILE OPERATIONS		
		spark-shell --packages org.apache.spark:spark-avro_2.12:2.4.5	
	Load avro format file	val prodDF = spark.read.format("avro").load("/user/spark/dataset/retail_db/products_avro")	
	Write avro file	filteredProdDF.write.format("avro").option("compression","snappy").save("/user/spark/dataset/output/products-avro-snappy")	
	Extract schema	hadoop jar avro-tools-1.8.2.jar getschema /user/spark/dataset/output/products-avro-snappy/part-00001-634cee1c-4ffe-42fb-89b5-0a6336628fd0-c000.avro > output.avsc --check the result cat output.avsc	
	Get meta data	hadoop jar avro-tools-1.8.2.jar getmeta /user/spark/dataset/output/products-avro-snappy/part-00001-634cee1c-4ffe-42 fb-89b5-0a6336628fd0-c000.avro	
	ORC FILE OPERATIONS		
	Read the orc file	val prodDF =spark.read.orc("/user/spark/dataset/retail_db/products_orc")	
	Write orc file in hdfs	prodDF.write.orc("/user/spark/dataset/output/prodORC") --check the fiels in hdfs hdfs dfs -ls /user/spark/dataset/output/prodORC	
	Orcfiledump	hive --orcfiledump -d /user/spark/dataset/output/prodORC/part-00001-83da14bb-bdbe-44be-bcba-4513266f5d00-c000.snappy.orc > prod_data_dump.txt tail -f prod_data_dump.txt	

#	Operation	
	DATAFRAME	
1	Drop columns	val custDF = spark.read.option("header", true).csv("/user/spark/dataset/retail_db/customers")

		val filteredCustDF = custDF.drop("customer_email", "customer_password", "customer_street")
2	Add columns	val custDFCountry = custNewDF.withColumn("country", lit("USA"))
3	Combine two columns	val custDFCombineColumns = custDFCountry.withColumn("customer_name", concat_ws(" ", $"customer_fname", $"customer_lname")) --view new result custDFCombineColumns.show(2)
4	Drop the duplicate columns	val custDF = custDFCombineColumns.drop("customer_fname", "customer_lname")
5	Print schema	custDF.printSchema
6	Casting column to Integer	val custDFCast = custDF.withColumn("customer_zipcode", col("customer_zipcode").cast("Integer")) --view changes custDFCast.printSchema
7	Substring on column	val custDFSubstring = custDFCast.withColumn("Country", col("Country").substr(0,2)) --view custDFSubstring.show(2)
8	withColumnRenamed	val custDFCountryRenamed = custDFSubstring.withColumnRenamed("Country", "customer_country")
	TIME CONVERSION	
	Load data	var orderDF = spark.read.parquet("/user/spark/dataset/retail_db/orders_parquet") var orderDFDate = orderDF.select("order_date")
	Convert using "from_unixtime"	var orderDFDateTimestamp = orderDFDate.withColumn("order_ts", from_unixtime($"order_date"/1000)) --verify result orderDFDateTimestamp.show(3) orderDFDateTimestamp.printSchema
	To_date function	var orderDFDatedt = orderDFDateTimestamp.withColumn("order_dt", to_date(from_unixtime($"order_date"/1000))) --view result orderDFDatedt.show(3)

		orderDFDatedt.printSchema
	Day, month and year	var orderDFDayMonthYear = orderDFDatedt.withColumn("year", year($"order_dt")).withColumn("month", month($"order_dt")).withColumn("day", dayofmonth($"order_dt"))
	Hour, minute, second	orderDFDayMonthYear = orderDFDayMonthYear.withColumn("Hour", hour($"order_dt")).withColumn("Min", minute($"order_dt")).withColumn("sec", second($"order_dt"))
	current_date function	var orderDFDayMonthYearCurrentDate = orderDFDayMonthYear.withColumn("cur_date", current_date()) --verify orderDFDayMonthYearCurrentDate.show(3)
	current_timestamp function	var orderDFDayMonthYearCurrentDateTS = orderDFDayMonthYear.withColumn("cur_ts", current_timestamp()) --verify orderDFDayMonthYearCurrentDateTS.show(3, false)
	STRING FUNCTIONS	
	Concat_ws	val cust= spark.read.json("/user/spark/dataset/retail_db/customers-json") --combine columns cust.select(concat_ws(" ", $"customer_fname", $"customer_lname")).show(3) --specify the column name using alias function cust.select(concat_ws(" ", $"customer_fname", $"customer_lname").alias("customer_name")).show(3)
	Lower/lower/upper	cust.select(lower($"customer_fname"), upper($"customer_fname"), $"customer_fname").show(3)
	regexp_replace	cust.select($"customer_zipcode", regexp_replace($"customer_zipcode", "00", "99").alias("zip code")).show(10)
	split	cust.select($"customer_street", split($"customer_street", " ").getItem(0).alias("house_no")).show(5)
	substring	cust.select($"customer_fname", substring($"customer_fname", 0,3)).show(3)
	HIVE METASTORE	
	Read hive table	val orders = spark.sql("select * from default.orders")
	Create new table in HMS using SaveAsTable API	orders.write.format("hive").saveAsTable("orders_replica") --we can check in scala prompt or hive hive> select * from default.orders_replica limit 10;
	Describe table	*hive> describe formated orders_replica;*
	Set configuration	*scala> spark.sqlContext.setConf("hive.exec.dynamic.partition", "true")*

		scala> spark.sqlContext.setConf("hive.exec.dynamic.partition.mode", "nonstrict")
	Create partitioned table in HMS using SaveAsTable API	scala> orders.write.format("hive").partitionBy("order_date").saveAsTable("default.orders_partitioned") --view the result in hive hive> select * from default.orders_partitioned limit 10; hive> describe formatted orders_partitioned; Result ===== # Partition Information # col_name data_type comment order_date string
	Create table using Create table command	scala> spark.sql(""" | create table order_parquet(| order_id int, | order_date string, | order_customer_id int, | order_status string) STORED AS PARQUET | """) --check the created table hive> describe formatted order_parquet;
	Write data in overwrite mode	scala> orders.write.format("hive").mode("overwrite").saveAsTable("default.order_parquet") scala> orders.count --verify result hive> select * from order_parquet limit 10; hive> select count(1) from order_parquet;
	Write data in append mode	scala> orders.write.format("hive").mode("append").saveAsTable("default.order_parquet") --verify result hive> select count(1) from order_parquet;
	Group and Aggregate functions	
	Load data	--load data scala> val prod = spark.sql("select * from default.products")

		scala> val cat = spark.sql("select * from default.categories") --create views scala> prod.createOrReplaceTempView("productsView") scala> cat.createOrReplaceTempView("categoriesView") ---merge two views scala> spark.sql("""select * from productsView p join categoriesView c on p.product_category_id = c.category_id """).show(10) --remove unnecessary columns scala> val data = spark.sql("""select * from productsView p join categoriesView c on p.product_category_id = c.category_id """).drop("product_description", "product_image", "category_department_id", "product_category_id") --check result scala> data.show(10) data.createOrReplaceTempView("productsView")
	Group by	scala> spark.sql(""" | select category_name, count(1) from productsView | group by category_name"""). | show(5)
	Order by	scala> spark.sql("select distinct category_name from productsView").show() scala> spark.sql("select category_name, count(1) as count from productsView group by category_name order by count(1) desc").show()
	Aggregation functions: Count, Max, Min, Avg, Sum	spark.sql("select category_name, max(product_price), min(product_price), avg(product_price), sum(product_price) from productsView group by category_name").show(10)
	Ranking Windows Functions	
	RANK()	--skip the rank if the peers' ranks are same spark.sql("select category_name, product_price, rank() over (partition by category_name order by product_price) rank from productsView").show(5) spark.sql("select category_name, product_price, rank() over (partition by category_name order by product_price desc) rank from productsView").show(5)
	DENSE_RANK()	--will not skip the rank even if the peers' ranks are same scala> spark.sql("select category_name, product_price, dense_rank() over (partition by category_name order by product_price) rank from productsView").show(40)

	ROW_NUMBER()	scala> spark.sql("select category_name, product_price, row_number() over (partition by category_name order by product_price) rank from productsView").show(40)
		scala> val top_cat = spark.sql("select category_name, product_name, product_price, dense_rank() over (partition by category_name order by product_price) rank from productsView") scala> top_cat.filter($"rank"===1).select("category_name","product_name","product_price").show()
	Windows Functions	
		Calculate average price for all categories by using AVG() window function
		--below prints average price of all spark.sql("select category_name, product_name, product_price, avg(product_price) over() avg_price from productsView").show(10, false)
		spark.sql("select category_name, product_name, product_price, avg(product_price) over(partition by category_name) avg_price from productsView").show(30, false)
		scala> spark.sql("select category_name, product_name, product_price, avg(product_price) over(partition by category_name order by product_price desc) avg_price from productsView").show(30, false)