**PySpark Project -2**

**File Content and explanation**

**I.** **connection.prop File:**

This file contains the database connection details in the following format:

[DBCRED]

driver=com.mysql.cj.jdbc.Driver

host=jdbc:mysql://127.0.0.1

port=3306

user=root

pass=Root123$

driver: Specifies the MySQL JDBC driver required to connect to the MySQL database.

host: The JDBC URL to the MySQL database located on localhost (127.0.0.1).

port: Port number 3306, which is the default for MySQL.

user: Username to log in to the database (root).

pass: Password for the root user (Root123$).

This file is important because the application uses it to connect to the database. It's read by the application to set up the database connection and interact with the data.

**II.** **app.properties File:**

This file contains the Spark configuration for your application:

[CONFIGS]

spark.app.name=spark\_retail\_app

spark.master=local[1]

spark.eventLog.enabled=true

spark.eventLog.dir="file:///tmp/spark-events"

spark.history.fs.logDirectory="file:///tmp/spark-events"

spark.sql.extensions="io.delta.sql.DeltaSparkSessionExtension"

spark.sql.catalog.spark\_catalog="org.apache.spark.sql.delta.catalog.DeltaCatalog"

Let's break down each configuration:

* spark.app.name=spark\_retail\_app: Sets the name of your Spark application as spark\_retail\_app.
* spark.master=local[1]: Runs Spark locally using one CPU core. This is typically used for testing and local development.
* spark.eventLog.enabled=true: Enables event logging for the application.
* spark.eventLog.dir="file:///tmp/spark-events": Specifies the directory where the event logs will be saved (/tmp/spark-events in this case).
* spark.history.fs.logDirectory="file:///tmp/spark-events": This sets the log directory for the Spark history server to the same directory as the event logs.
* spark.sql.extensions="io.delta.sql.DeltaSparkSessionExtension": Loads the Delta Lake extension into the Spark session **to enable Delta Lake operations** (like working with **Delta tables and ACID** **properties** ).
* spark.sql.catalog.spark\_catalog="org.apache.spark.sql.delta.catalog.DeltaCatalog": Configures Delta Lake as the catalog for Spark SQL.

This file is used to configure and manage the behavior of your Spark application, especially with respect to Delta Lake (a storage layer optimized for big data).

**III. cust\_navigation.json : Customer Data (JSON Format):**

This is a collection of customer data in JSON format, which contains:

json

[

{

"id": 112,

"comments": "Customer very concerned about the exact color of the models...",

"pagevisit": ["home", "about-us", "profile", "cart", "order", "exit"]

},

{

"id": 114,

"comments": "Can we deliver the new Ford Mustang models by end-of-quarter?",

"pagevisit": ["home", "about-us", "profile", "cart", "order", "exit"]

},

...

]

Each entry represents customer interactions and their browsing behavior:

* id: A unique identifier for the customer.
* comments: A free-text field that contains notes or remarks about the customer's concerns or preferences.
* pagevisit: A list of web pages that the customer visited during their session.

This data might be used in the application for customer behavior analysis, such as tracking patterns in page visits, understanding preferences, or identifying common concerns.

**IV. DriverModule.py**

**1. Setting Up the Spark Session**

* The script begins by creating a Spark session using get\_spark\_session(), which initializes the environment for running Spark tasks with Delta support. It configures the session with necessary extensions for Delta Lake and adds JDBC support via arg2\_conn\_jar.
* The script sets the log level to ERROR to suppress unnecessary logs.

|  |
| --- |
| def get\_spark\_session(typeflg,propfile,jdbc\_lib):  spark\_config = SparkConf()  config = ConfigParser() # used to read configuration files  config.read(propfile) # reads the contents of the configuration file using the file path stored in the variable  for config\_name, config\_value in config.items("CONFIGS"):  # This loop iterates over all the key-value pairs in the [CONFIGS] section of the configuration file  spark\_config.set(config\_name, config\_value)  """  app.properties file  [CONFIGS]  spark.app.name=spark\_retail\_app  spark.master=local[1]  spark.eventLog.enabled=true  spark.eventLog.dir="file:///tmp/spark-events"  spark.history.fs.logDirectory="file:///tmp/spark-events"  spark.sql.extensions="io.delta.sql.DeltaSparkSessionExtension"  spark.sql.catalog.spark\_catalog="org.apache.spark.sql.delta.catalog.DeltaCatalog"  """  # creating Spark Session variable  try:  if typeflg=='delta':  #builder= SparkSession.builder.config(conf=spark\_config).config("spark.jars",jdbc\_lib)  #spark = configure\_spark\_with\_delta\_pip(builder).getOrCreate()  #return spark  builder = SparkSession.builder.appName("MyApp") \  .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \  .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog") \  .config("spark.jars", jdbc\_lib).enableHiveSupport()  spark = configure\_spark\_with\_delta\_pip(builder).getOrCreate()  return spark  else:  spark = SparkSession.builder.config(conf=spark\_config).config("spark.jars", jdbc\_lib).enableHiveSupport().getOrCreate()  return spark  except Exception as spark\_error:  print(spark\_error)  sys.exit(1)   * The function dynamically reads Spark configuration settings from a properties file and sets them in the Spark session. * It differentiates between sessions with Delta Lake support and standard Spark sessions based on the typeflg parameter. * It returns a configured SparkSession object that can be used to interact with Spark, optionally with Delta Lake or Hive support, depending on the configurations and the flag. |

**2. Database Management**

* The script creates (or re-creates) databases to store the processed data:
  + **retail\_curated**: Curated, cleaned, and processed data.
  + **retail\_discovery**: Data for analysis and discovery.
  + **retail\_dim**: Dimension tables for entities like employees and products.

**3. Extracting and Processing Employee Data (SCD Type 2)**

* The script retrieves employee records that were updated or inserted in the last day using an SQL query and fetches them using the **getRdbmsData()** function.

|  |
| --- |
| def getRdbmsData(propfile,sparksess,db,tbl):#configuration driven approach  config = ConfigParser()  config.read(propfile)  driver=config.get("DBCRED", 'driver')  host=config.get("DBCRED", 'host')  port=config.get("DBCRED", 'port')  user=config.get("DBCRED", 'user')  passwd=config.get("DBCRED", 'pass')  url=host+":"+port+"/"+db  #jdbc:mysql://127.0.0.1:3307/empoffice  db\_df=sparksess.read.format("jdbc").option("url",url)\  .option("dbtable",tbl)\  .option("user",user).option("password",passwd)\  .option("driver",driver) \  .load()  # dataframe can be created as like below  # db\_df = sparksess.read.jdbc(url=url, table=tbl, properties={"user": user, "password": passwd, "driver": driver})  return db\_df  """  [DBCRED]  driver=com.mysql.cj.jdbc.Driver  host=jdbc:mysql://127.0.0.1  port=3306  user=root  pass=Root123$  """   * The function connects to an RDBMS (like MySQL) using the JDBC protocol, with connection details (such as the host, port, user, and password) provided in a configuration file. * It retrieves data from the specified database table (tbl) and returns it as a Spark DataFrame (db\_df), allowing for further processing and analysis in Spark. * The configuration-driven approach allows for flexible and reusable code, as the database credentials and connection details can easily be changed by modifying the configuration file. |

* **Data Munging**: The raw employee data is cleaned and transformed using munge\_data() to make it tidy and usable for downstream tasks.

|  |
| --- |
| def munge\_data(df,dedupflag,naallflag,naanyflag,naallsubsetflag,naanysubsetflag,sub):  print("Number of partitions in the given DF {}".format(df.rdd.getNumPartitions()))  if dedupflag:  print("Raw df count is {} ".format(df.count()))  df = df.dropDuplicates()  print("deduplicated count is {} ".format(df.count()))  if naallflag:  df=df.na.drop("all")  if naanyflag:  df = df.na.drop("any")  if naallsubsetflag:  df = df.na.drop("all",subset=sub)  if naanysubsetflag:  df = df.na.drop("any",subset=(sub))  print("count of df is {}".format(df.count()))  return df   * **Deduplication**: Removes duplicate rows if dedupflag is set. * **Null Value Handling**: Drops rows based on various conditions of null values (for all columns or specific subsets of columns) controlled by the respective flags. * **Count Printing**: It prints the row counts at various stages to track the effect of the transformations. * **Final Output**: Returns a cleaned DataFrame after performing the deduplication and missing value removal operations. |

* **SCD Type 2 (Slowly Changing Dimension)**: It checks whether the Hive table hive\_employees exists using checkForHiveTable(). If it exists, it updates records based on SCD Type 2 principles; otherwise, it inserts new records using writeHiveTableSCD2()

|  |
| --- |
| def checkForHiveTable(sparksess, dbname, tblname):  from pyspark.sql.functions import col  if (sparksess.sql(f"show tables in {dbname}").filter(col("tableName") == tblname).count() > 0):  return True  else:  return False   * **Purpose**: To check if a Hive table exists in a specific database. * **How**: It lists all tables in the database, filters by the target table name, and checks if there are any matches. * **Return**: True if the table exists, False if it doesn't. |

|  |
| --- |
| def writeHiveTableSCD2(tableexistflag,sparksess,hive\_table,df\_employees\_new\_updated):  if tableexistflag:  print("hive\_employees table exists")  df\_hive\_existing = readHiveTable(sparksess,hive\_table).groupBy("employeeNumber").agg(max("ver").alias("max\_ver"))  if df\_hive\_existing.count() > 0:  joined\_df = df\_employees\_new\_updated.alias("new").join(df\_hive\_existing.alias("exist"), on="employeeNumber",  how="left")#lookup and enrichment (Data wrangling)  joined\_df\_exist = joined\_df \  .select("new.employeeNumber", "new.lastName", "new.firstName", "new.extension",  "new.email", "new.officeCode", "new.reportsTo", "new.jobTitle", "new.upddt", "new.leaveflag",  "exist.max\_ver") \  .withColumn("ver", expr(  "row\_number() over(partition by employeeNumber order by upddt) + coalesce(max\_ver,0)")).drop("max\_ver")  # .where("new.employeeNumber is not null")\  joined\_df\_exist.show(10)  #writeToHiveTable(joined\_df\_exist,"Append","hive",hive\_table)  joined\_df\_exist.createOrReplaceTempView("empnewexist")  print("employee exist and the old+new data is..")  sparksess.sql("select \* from empnewexist").show(5, False)  sparksess.sql("insert into retail\_dim.hive\_employees select \* from empnewexist")  else:  df\_employees\_new\_updated=df\_employees\_new\_updated.withColumn("ver", lit(1))#Data Enrichment  df\_employees\_new\_updated.createOrReplaceTempView("empnew")  print("employee table not exist and the new data is..")  sparksess.sql("select \* from empnew").show(5,False)  sparksess.sql("""CREATE external TABLE retail\_dim.hive\_employees(  `employeenumber` int,`lastname` string,`firstname` string,`extension` string,`email` string,  `officecode` string, `reportsto` int,`jobtitle` string, `upddt` date, `leaveflag` string,`ver` int)  row format delimited fields terminated by ','  location 'hdfs:///user/hduser/empdata/'""")  sparksess.sql("insert into retail\_dim.hive\_employees select \* from empnew")  #writeToHiveTable(df\_employees\_new\_updated, "Overwrite", "hive", hive\_table)  print("table doesn't exists, hence created and loaded the data")  **When the Hive Table Exists (tableexistflag=True):**  a. Reading Existing Hive Data:   * Loads current employee data from hive\_employees. * Groups by employeeNumber to find the maximum version (max\_ver) for each employee, enabling version control using the Windowing function .   b. Joining New and Existing Data:   * Performs a left join between new employee data and existing data on employeeNumber. * Ensures version numbers are incremented for existing employees in Hive.   c. Data Wrangling and Versioning:   * Selects specific columns from both datasets. * Calculates a new version (ver) using row\_number() for each employee, partitioned by employeeNumber. * If the employee does not exist, the version defaults to 1.   d. Writing Data Back to Hive:   * Creates a temporary view (empnewexist) for the updated data. * Uses a SQL INSERT to append the data back into the retail\_dim.hive\_employees table.   **2. When the Hive Table Does Not Exist (tableexistflag=False):**  a. Enriching New Data:   * Adds a ver column with value 1 for all new records.   b. Creating the Hive Table:   * Creates a new external Hive table hive\_employees with the necessary schema.   c. Inserting New Data into Hive:   * Inserts the enriched data into the newly created Hive table using SQL INSERT. |

**4. Processing Office Data**

* The office data is fetched from the RDBMS (using getRdbmsData()), cleaned, and written to Hive in the retail\_dim.offices\_raw table, overwriting the existing data.

|  |
| --- |
| def getRdbmsData(propfile,sparksess,db,tbl):#configuration driven approach  config = ConfigParser()  config.read(propfile)  driver=config.get("DBCRED", 'driver')  host=config.get("DBCRED", 'host')  port=config.get("DBCRED", 'port')  user=config.get("DBCRED", 'user')  passwd=config.get("DBCRED", 'pass')  url=host+":"+port+"/"+db  #jdbc:mysql://127.0.0.1:3307/empoffice  db\_df=sparksess.read.format("jdbc").option("url",url)\  .option("dbtable",tbl)\  .option("user",user).option("password",passwd)\  .option("driver",driver) \  .load()  # dataframe can be created as like below  # db\_df = sparksess.read.jdbc(url=url, table=tbl, properties={"user": user, "password": passwd, "driver": driver})  return db\_df  """  [DBCRED]  driver=com.mysql.cj.jdbc.Driver  host=jdbc:mysql://127.0.0.1  port=3306  user=root  pass=Root123$  """   * The function connects to an RDBMS (like MySQL) using the JDBC protocol, with connection details (such as the host, port, user, and password) provided in a configuration file. * It retrieves data from the specified database table (tbl) and returns it as a Spark DataFrame (db\_df), allowing for further processing and analysis in Spark. * The configuration-driven approach allows for flexible and reusable code, as the database credentials and connection details can easily be changed by modifying the configuration file. |

**5. Customer & Payments Data with Partition and Pushdown Optimization**

* A SQL query joins customers and payments tables, retrieving customer data for payments made after July 2022. Partition and predicate pushdown optimizations are applied to this data using getRdbmsPartData() and optimize\_performance().

|  |
| --- |
| def getRdbmsPartData(propfile,sparksess,db,tbl,partcol,lowerbound,upperbound,numpart):  config = ConfigParser()  config.read(propfile)  driver=config.get("DBCRED", 'driver')  host=config.get("DBCRED", 'host')  port=config.get("DBCRED", 'port')  user=config.get("DBCRED", 'user')  passwd=config.get("DBCRED", 'pass')  url=host+":"+port+"/"+db  db\_df=sparksess.read.format("jdbc").option("url",url)\  .option("dbtable",tbl)\  .option("user",user).option("password",passwd)\  .option("driver",driver) \  .option("lowerBound", lowerbound)\  .option("upperBound", upperbound)\  .option("numPartitions", numpart)\  .option("partitionColumn", partcol)\  .load()  return db\_df   * **Connects to an RDBMS**: Uses JDBC to connect to the specified database using credentials from the configuration file. * **Loads data into a Spark DataFrame**: Retrieves the data from the specified table and loads it into a Spark DataFrame. * **Partitions the data**: The data is partitioned based on a given column (partcol) into a specified number of partitions (numpart) for parallel processing. * **Defines partition range**: Uses the lowerBound and upperBound options to define the range of values for the partitioning column, optimizing the data loading process. * **Returns the DataFrame**: The function returns the partitioned DataFrame for further processing. |

|  |
| --- |
| def optimize\_performance(sparksess,df,numpart,partflag,cacheflag,numshufflepart=200):  print("Number of partitions in the given DF {}".format(df.rdd.getNumPartitions()))  if partflag:  df = df.repartition(numpart)  print("repartitioned to {}".format(df.rdd.getNumPartitions()))  else:  df = df.coalesce(numpart)  print("coalesced to {}".format(df.rdd.getNumPartitions()))  if cacheflag:  df.cache()  print("cached ")  if numshufflepart!=200:  # default partions to 200 after shuffle happens because of some wide transformation spark sql uses in the background  sparksess.conf.set("spark.sql.shuffle.partitions", numshufflepart)  print("Shuffle part to {}".format(numshufflepart))  return df   * Repartitioning or coalescing the DataFrame based on the partflag (to control partition size and improve performance). * Caching the DataFrame if the cacheflag is set to True (to store data in memory for faster access). * Optionally adjusting the number of shuffle partitions to control shuffle behavior during wide transformations. |

* The processed data is written to the Hive table retail\_curated.custpayments with an overwrite option.

**6. Orders & Order Details Data**

* The script joins orders and orderdetails tables to fetch details about recent orders. It applies optimization techniques and writes the data to the Hive table.

**7. Product Data Processing with Delta Lake Merge**

* The product data is fetched from the RDBMS and passed through a UDF (udfProfPromo) to compute profitability metrics, such as profit, profit percentage, promotion indicator, and demand indicator.

|  |
| --- |
| def udfProfPromo(cp, sp, qty):  profit = sp - cp  profitpercent = (profit / cp) \* 100  if profit > 0:  profind = 1  else:  profind = 0  if qty > 0 and qty < 1500 and profitpercent > 50.0:  demandind = 1  else:  demandind = 0  if qty > 1500 and profitpercent > 20.0:  promoind = 1  else:  promoind = 0  return profit, profitpercent, promoind, demandind   * The udfProfPromo function calculates: * The **profit** and **profit percentage** of a product. * A **promotion indicator** (promoind), which is set to 1 if the product is promotional based on quantity and profit percentage. * A **demand indicator** (demandind), which is set to 1 if the product is considered in demand based on quantity and profit percentage. |

* If the Hive table hive\_products exists, a Delta Lake merge operation is performed to update or insert product records.

**8. Data Wrangling: Joining Orders with Products**

* The script performs data wrangling by joining the product and order data, preparing it for further processing. The resulting data is written to the Hive table retail\_curated.ordersproducts.

**9. Parsing Customer Navigation JSON Data**

* Customer navigation data stored in a JSON format is read into a DataFrame with a custom schema (strtype).
* The navigation data is then transformed using the posexplode() function to break down array elements (page visits) into individual rows and written to Hive as retail\_curated.cust\_navigation
* The script generates reports on the first and last pages visited by customers and saves this report as a JSON file in HDFS.

**10. Loading External Order Rate Data**

* An external Hive table (retail\_discovery.dim\_order\_rate) is created and populated by loading data from a local CSV file (orders\_rate.csv).

**11. Employee Rewards Discovery**

* A join operation between offices, employees, and customer payments is performed to find high-performing employees based on the total amount of payments processed.

|  |
| --- |
| high\_performers\_df = high\_performers\_df.groupBy("employee\_id", "employee\_name") \ .agg(sum("payment\_amount").alias("total\_payments")) # Filter employees who have processed  high payments high\_performers\_df = high\_performers\_df.filter(col("total\_payments") > 10000) |

* The top-ranked employee per state (based on payment amounts) is determined, and the results are written to the Hive table retail\_discovery.employeerewards.

**12. Customer Frustration Discovery**

* The script attempts to detect frustrated customers by joining cust\_navigation and dim\_order\_rate tables. Frustration levels are computed based on severity scores.

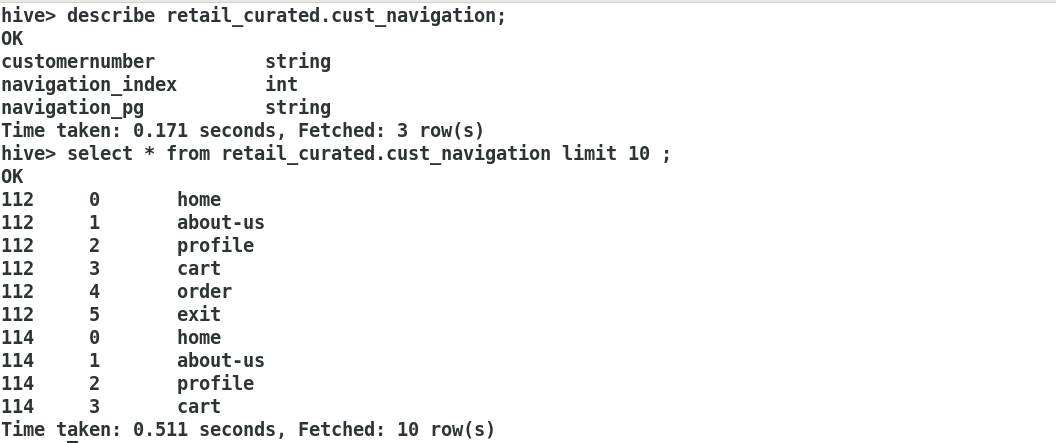
|  |
| --- |
| **Customer Frustration Calculation**  select customernumber,  total\_siverity,  case  when total\_siverity between -10 and -3 then 'highly frustrated'  when total\_siverity between -2 and -1 then 'low frustrated'  when total\_siverity = 0 then 'neutral'  when total\_siverity between 1 and 2 then 'happy'  when total\_siverity between 3 and 10 then 'overwhelming'  else 'unknown'  end as customer\_frustration\_level  **Calculating the Total Severity**  select customernumber, sum(siverity) as total\_siverity  from  (select o.id as customernumber, o.comments, r.orddesc, siverity  from cust\_navigation o  left outer join retail\_discovery.dim\_order\_rate r  where o.comments like concat('%', r.orddesc, '%')) temp1  group by customernumber  Note :  **Example Process:**  Let’s break it down with an example:   * **comments** = "The order was delayed" (from cust\_navigation) * **orddesc** = "delayed" (from dim\_order\_rate) * A match occurs, and the corresponding severity for "delayed" might be **-5** (you would define this in the dim\_order\_rate table or some logic in your data).   For the customer with **customernumber** = 123, if there are multiple similar comments like "The order was delayed," their total severity (total\_siverity) would be the sum of all these severity scores (e.g., -5 + -5 + -3 = -13).  The resulting **frustration level** will then be classified as **"highly frustrated"** based on the total\_siverity falling between -10 and -3.  where o.comments like concat('%', r.orddesc, '%') is performing a **text-based matching** operation in SQL (o.comments is a col in table) |

|  |
| --- |
| frustration\_df = frustration\_df.withColumn("frustration\_level", when(col("severity\_score") > 80, "High") .when(col("severity\_score") > 50, "Medium") .otherwise("Low")) |

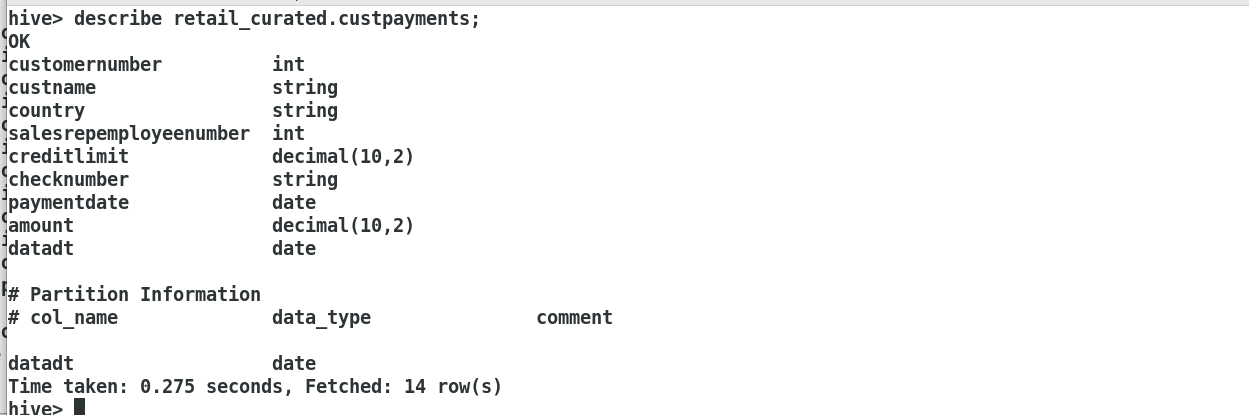
**Databases and tables in it :**

**retail\_curated Database :**

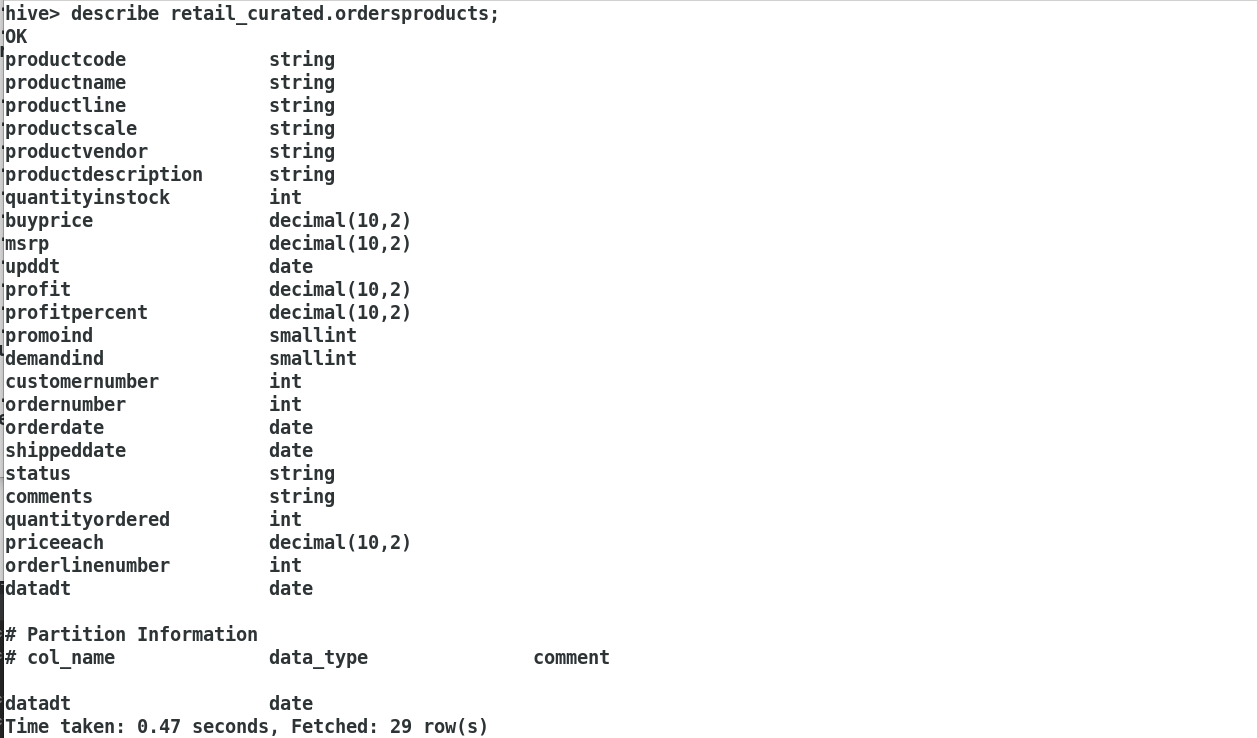
* 1. **cust\_navigation table**



* 1. **custpayments table**

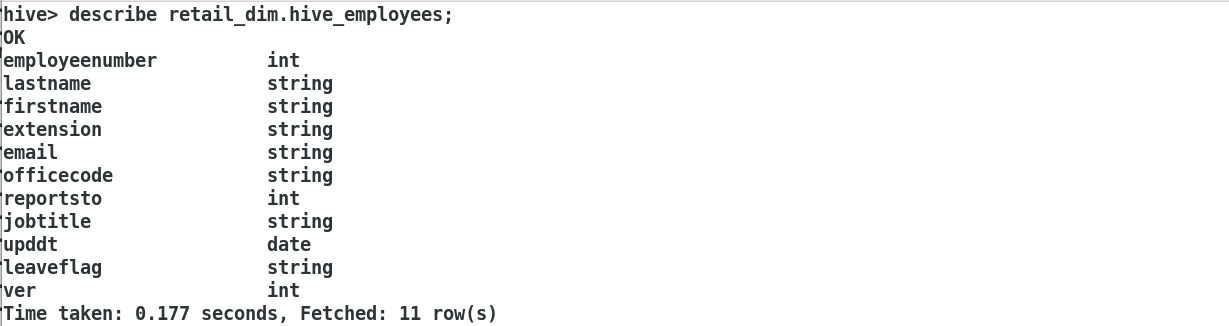


* 1. ordersproduct table

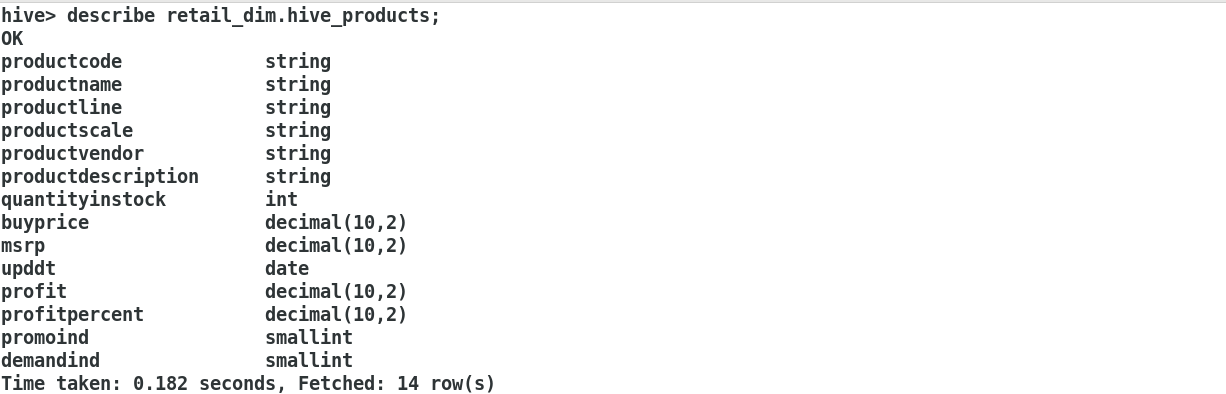


Retail\_dim database

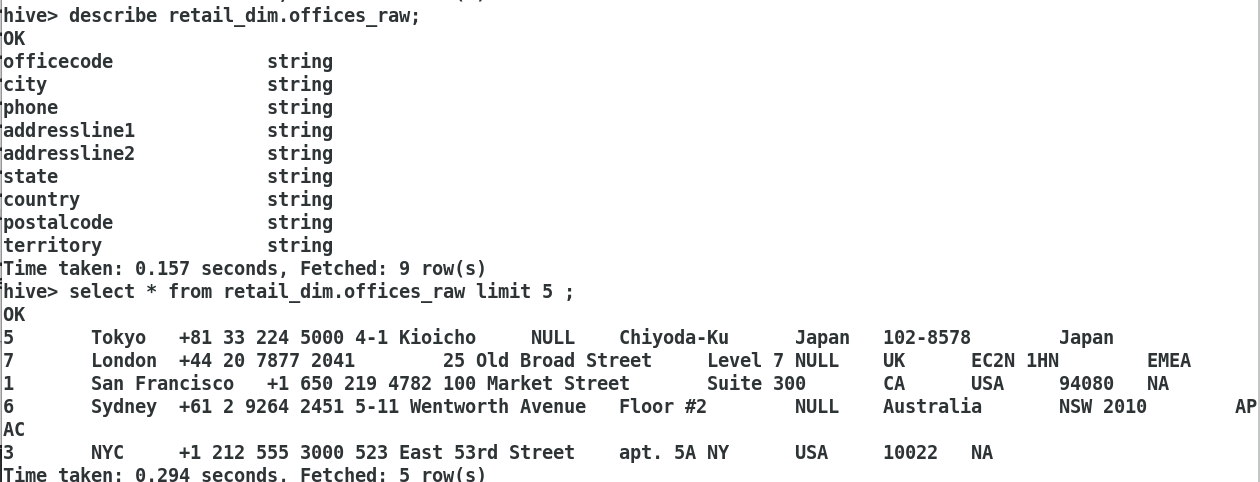
* 1. hive\_employees table



* 1. hive\_products table

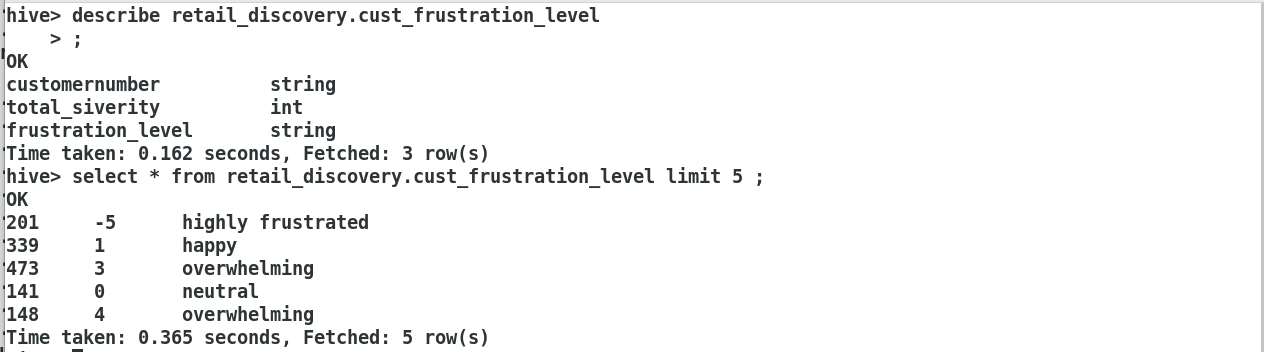


* 1. offices\_raw table

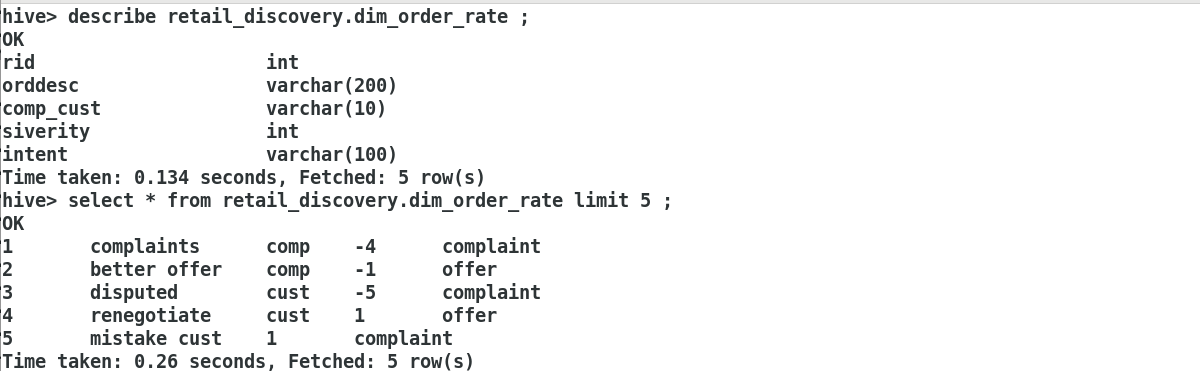


Retail\_discovery Database :

1. cust\_frustration\_level table



1. dim\_order\_rate table



1. employeerewards table

