DATAMASS

Big Data Academy 2019

Kainos

Lab 4 - Data types/partitions

Introduction

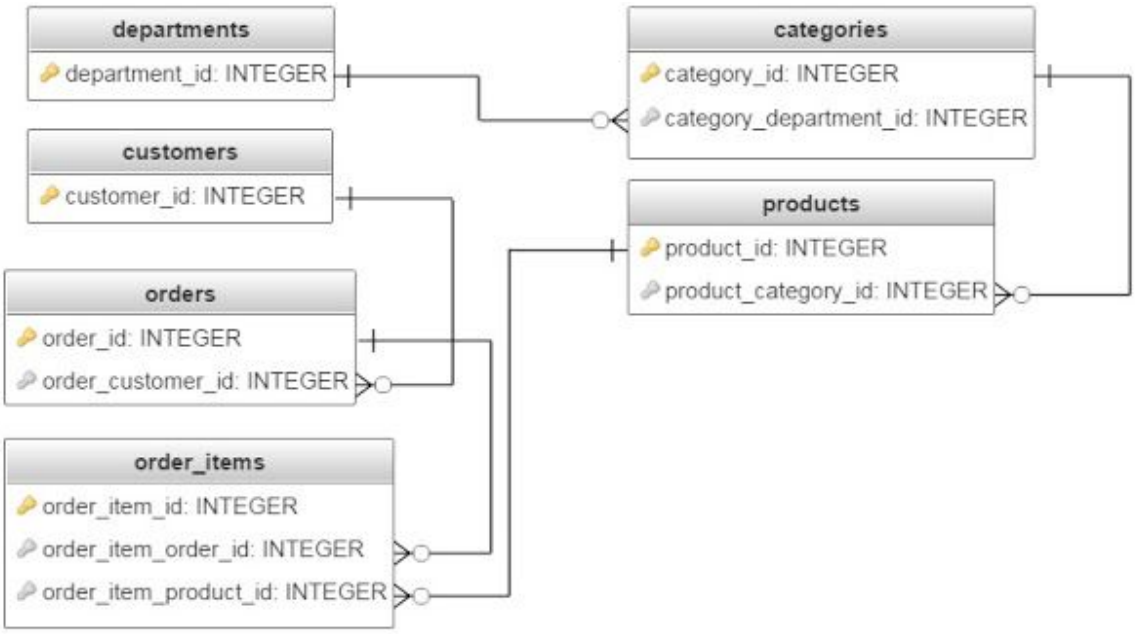
The aim of the laboratory classes is to familiarize the participants of the course with different data formats in order to more efficiently store data on distributed environments. As part of laboratory classes, we will use *AVRO*, *RCFile*, *ORCFile* and *Parquet* formats.

NOTE: *All operations will be carried out on the database located on a virtual machine created for the purpose of laboratory classes.*

MySQL

db-name: retail_db
db-user: retail_dba
db-passwd: cloudera

The relation diagram in **retail_db** database looks as follows:



Static partition

- Place data from the customers (mysql) table in the partitioned **customers_cities** table. Use static partition key: **customer_state**. The values of the data depend on the query being created.

```
sqoop import \
--connect "jdbc:mysql://quickstart.cloudera:3306/retail_db" \
--username retail_dba --password cloudera \
--query "select customer_id, customer_fname, customer_lname from customers
where customer_state = 'NY' and \$CONDITIONS" \
--target-dir /user/cloudera/mysql/customers_cities \
--fields-terminated-by "\t" \
--hive-table customers_cities \
--hive-partition-key customer_state \
--hive-partition-value 'NY' \
--hive-import \
--split-by customer_id
```

- In case of data placement in subsequent partitions associated with the **customers_cities** table, the command should look like this:

```
sqoop import \
--connect "jdbc:mysql://quickstart.cloudera:3306/retail_db" \
--username retail_dba --password cloudera \
--query "select customer_id, customer_fname, customer_lname from customers
where customer_state = 'TX' and \$CONDITIONS" \
--target-dir /user/cloudera/mysql/customers_cities \
--fields-terminated-by "\t" \
--hive-table customers_cities \
--hive-partition-key customer_state \
--hive-partition-value 'TX' \
--hive-import \
--split-by customer_id
```

NOTE: Performing the above query will not delete the table and overwrite its value, but will only extend the scope of data with subsequent partitions.

Dynamic partition

- Import data from MySQL table **customers** to HIVE table **customers_all**.

```
sqoop import \
--connect "jdbc:mysql://quickstart.cloudera:3306/retail_db" \
--username retail_dba --password cloudera \
--table customers \
--fields-terminated-by "\t" \
--hive-import \
--hive-table customers_all
```

- Create table **customers_by_state** using dynamic partition on field *customer_state*.

```
CREATE TABLE customers_by_state (
 customer_fname STRING,
 customer_lname STRING,
 customer_email STRING
)
PARTITIONED BY (customer_state STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
```

- Import data from table **customers_all** to **customers_by_state** using dynamic partition.

```
insert overwrite table customers_by_state
partition (customer_state)
select customer_fname, customer_lname, customer_email, customer_state from
customers_all
```

- Before starting the import, set the appropriate environmental parameters for defining dynamically partitioned structures.

```
set hive.exec.dynamic.partition=true;
set hive.exec.dynamic.partition.mode=nonstrict;
```

File formats

This table presents the read and write procedures for various file formats. It is worth paying attention to the used codecs.

File Format	Action	Procedure and points to remember
TEXT FILE	READ	sparkContext.textFile(<path to file>);
	WRITE	sparkContext.saveAsTextFile(<path to file>,classOf[**compressionCodecClass**]); //use any codec here org.apache.hadoop.io.compress.(**BZip2Codec** or GZipCodec or SnappyCodec)
SEQUENCE FILE	READ	sparkContext.sequenceFile(<path location>,classOf[<class name>],classOf[<**compressionCodecClass** >]); //read the head of sequence file to understand what two class names need to be used here

	WRITE	rdd.saveAsSequenceFile(<path location>, Some(classOf[compressionCodecClass]))
		//use any codec here (**BZip2Codec**,GZipCodec,SnappyCodec)
		//here rdd is MapPartitionRDD and not the regular pair RDD.

PARQUET FILE	READ	//use data frame to load the file. sqlContext.read.parquet(<path to location>); //this results in a data frame object.

	WRITE	sqlContext.setConf("spark.sql.parquet.compression.codec","gzip") //use gzip, snappy, lzo or uncompressed here
		dataFrame.write.parquet(<path to location>);

ORC FILE	READ	sqlContext.read.orc(<path to location>); //this results in a dataframe
	WRITE	df.write.mode(SaveMode.Overwrite).format("orc") .save(<path to location>)
AVRO FILE	READ	import com.databricks.spark.avro._; sqlContext.read.avro(<path to location>); // this results in a data frame object

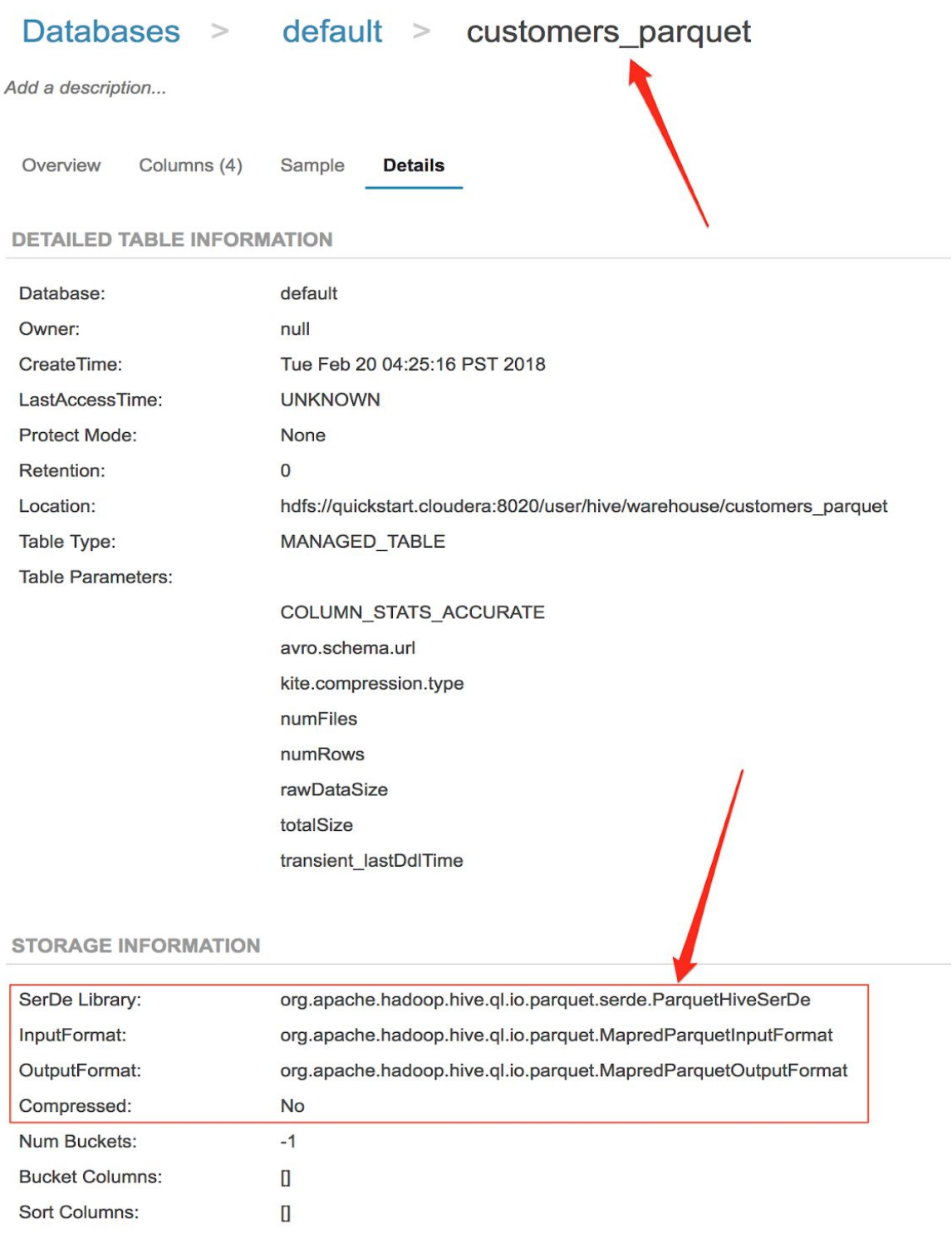
	WRITE	sqlContext.setConf("spark.sql.avro.compression.codec","snappy") //use snappy, deflate, uncompressed;
		dataFrame.write.avro(<path to location>);

JSON FILE	READ	sqlContext.read.json();
	WRITE	dataFrame.toJSON().saveAsTextFile(<path to location>,classOf[Compression Codec])

- Import data to customers_parquet table. The data is stored in the PARQUET format.

```
sqoop import \
--connect "jdbc:mysql://quickstart.cloudera:3306/retail_db" \
--username retail_dba --password cloudera \
--table customers \
--target-dir /user/cloudera/mysql/customers_parquet \
--fields-terminated-by "\t" \
--columns "customer_id, customer_fname, customer_email, customer_state" \
--hive-import \
--hive-table customers_parquet \
--as-parquetfile
```

- Checking whether the table was created in the appropriate file can be done by accessing the properties of the table in HUE.



- As part of laboratory classes, we use SQOOP in version Sqoop 1.4.6-cdh5.8.0, which does not support the ability to directly save data to Hive in AVRO format. The process of

writing data must be done in two stages. In the first stage, export data in AVRO format on HDFS. In the second stage, create a Hive table using the table structure saved in AVSC (Avro schema) file.

First stage:

```
sqoop import \
--connect "jdbc:mysql://quickstart.cloudera:3306/retail_db" \
--username retail_dba --password cloudera \
--table customers \
--target-dir /user/cloudera/mysql/customers_avro \
--fields-terminated-by "\t" \
--columns "customer_id, customer_fname, customer_email, customer_state" \
--as-avrodatafile
```

NOTE: Sqoop automatically generates the structure of the AVRO file in JSON format during import. The file is placed in the same location from which the import script was called. In our case it is HOME_FOLDER.

```
[cloudera@quickstart ~]$ cat customers.avsc
{
  "type" : "record",
  "name" : "customers",
  "doc" : "Sqoop import of customers",
  "fields" : [ {
    "name" : "customer_id",
    "type" : [ "null", "int" ],
    "default" : null,
    "columnName" : "customer_id",
    "sqlType" : "4"
  }, {
    "name" : "customer_fname",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "customer_fname",
    "sqlType" : "12"
  }, {
    "name" : "customer_email",
    "type" : [ "null", "string" ],
    "default" : null,
```

```
    "columnName" : "customer_email",
    "sqlType" : "12"
  }, {
    "name" : "customer_state",
    "type" : [ "null", "string" ],
    "default" : null,
    "columnName" : "customer_state",
    "sqlType" : "12"
  } ],
  "tableName" : "customers"
}
```

Second stage:

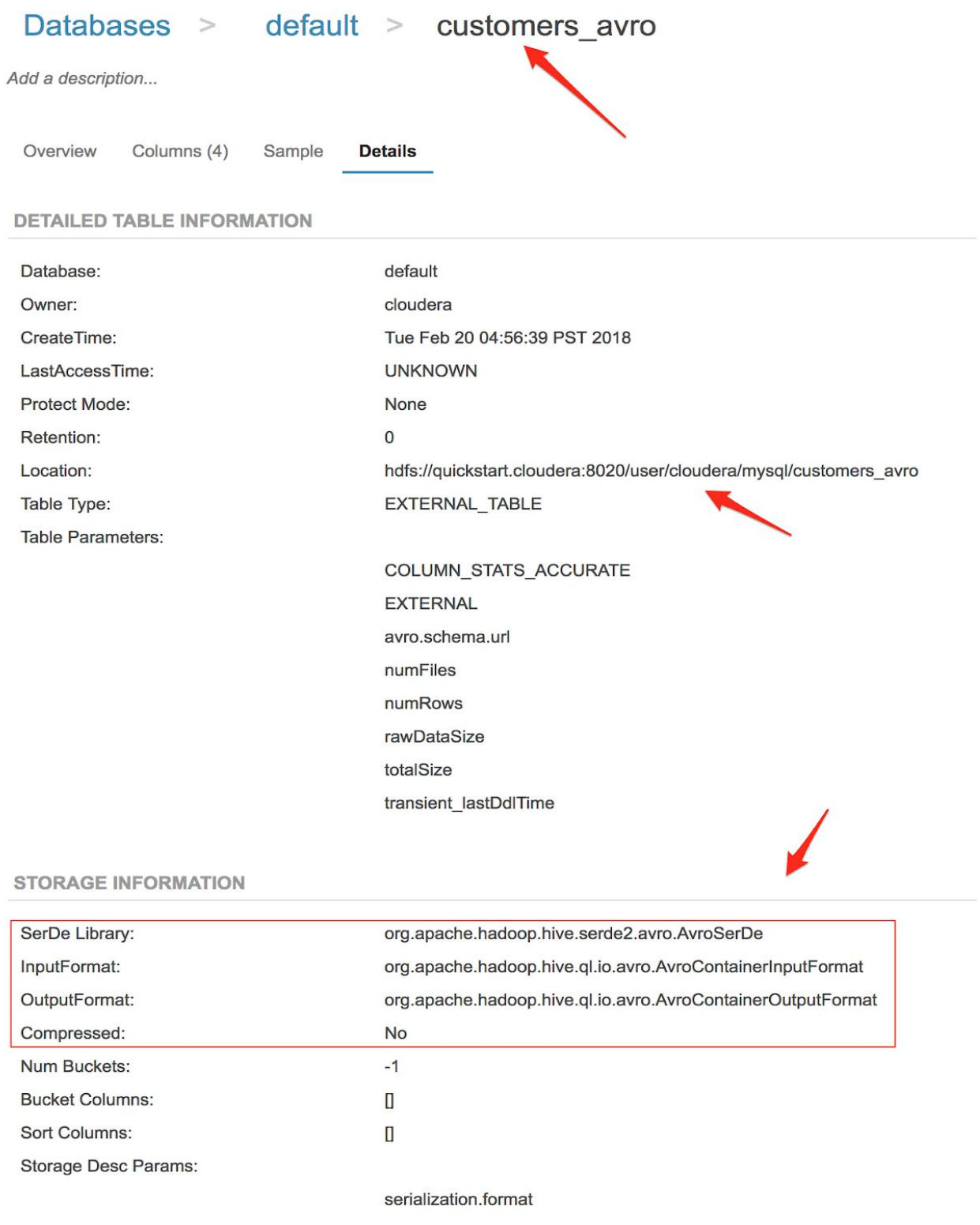
Place table definition file (AVCS file) on HDFS.

```
hadoop fs -put customers.avsc /user/cloudera/mysql/customers_avro_meta/
```

Then create table in the Hive editor using uploaded definition file.

```
CREATE EXTERNAL TABLE customers_avro
STORED AS AVRO
LOCATION '/user/cloudera/mysql/customers_avro'
TBLPROPERTIES ('avro.schema.url'=
'hdfs:/user/cloudera/mysql/customers_avro_meta/customers.avsc');
```

As a result of the operations carried out, we obtain the following table:



Exercise 1

Import data from the categories and products table (mysql). Divide the data into two files depending on the criterion used for import. For file no. 1 category_name = 'Basketball' for file no. 2 category_name = 'Hockey'. Files should be saved on HDFS. Columns in text files: **product_id**, **product_name**, **product_description**, **product_price**, **category_name**.

You should create a table with partitions for column **category_name**. After creating the table, you should add as separate partitions both categories.

Exercise 2

Import data from the categories and products table (mysql). Save the result of the join in the form of a Hive table called **products_categories**. The columns defined in the table are **product_id**, **product_name**, **product_description**, **product_price**, **category_name**. Create another table and use dynamic partitioning for the **category_name** field.

Exercise 3

Import the **products** table in parquet format using compression (org.apache.hadoop.io.compress.SnappyCodec). Location on HDFS **/home/kainos/mysql/products_parquet**. Use the separator '|'.

Exercise 4

Import the **products** table in AVRO format using compression (org.apache.hadoop.io.compress.SnappyCodec). Location on HDFS **/home/kainos/mysql/products_avro**. Use the separator '\001'. Table name in Hive **products_avro**.

Exercise 5

Create a MANAGED table using the AVRO format. The table should be created on the basis of data placed in **products_avro** and contain the following columns: **product_id**, **product_name**. Only 100 records should be imported to the table.