IST769 Lab G

# Wide-Column Model: Cassandra

### In this lab, we will explore Apache Cassandra the BASE Wide-Column database. Like most wide-column databases tables are dependent on row key. We will demonstrate how to create tables and perform CRUD operations in Cassandra, as well as importing and exporting data with Apache Spark.

### Learning Outcomes

At the end of this lab you should be able to:

* Create properly designed Cassandra tables with adequate cluster and partition keys.
* Use Apache Spark to query, import, and export data from Cassandra
* Create indexes and materialized views over Cassandra data to improve query performance and avoid ALLOW FILTERING activity.

### Pre-Requisites

Before you begin:

* Open a terminal window in the lab environment
* Set the current working directory to **advanced-databases**
* Start the following services required by the lab Spark and Cassandra:   
  **jupyter cassandra**

### Tools Used In this Lab

The following tools will be used in this lab:

1. To access Jupyter Lab from your Windows host:  
   [http://localhost:8888](http://localhost:8888/)   
   The password is **SU2orange!**
2. To access the Cassandra Shell:

PS> docker-compose exec cassandra cqlsh

# Lab Problem Set

Your employer (weather.com) would like you store weather sensor and forecast data. Eventually you will get readings from 2,000 cities worldwide every minute. That's 2.88 million rows each day and 1 billion rows a year! Since the data does not need to be read immediately when written across all nodes, you decide Cassandra is a good choice for this project! This data will be accessible by users so they can get weather information and historical trends for they cities they live in and visit. This should help you figure out how the data will be queried.

**QUESTIONS:   
  
For each question, include a copy of the code required to complete the question along with a screenshot of the code and a screenshot of the output.**

1. InCQL, create a Keyspace called **glab** with a replication factor of 1 and a Simple replication strategy. Use the keyspace.

**create keyspace glab with replication = { 'class': 'SimpleStrategy', 'replication\_factor' : 1 };**

**use glab;**

**describe glab**

1. In Spark, setup a spark session that is ready to talk with Cassandra.

**import pyspark**

**from pyspark.sql import SparkSession**

**# CASSANDRA CONFIGURATION**

**cassandra\_host = "cassandra"**

**# Spark init**

**spark = SparkSession \**

**.builder \**

**.master("local") \**

**.appName('jupyter-pyspark') \**

**.config("spark.cassandra.connection.host", cassandra\_host) \**

**.config("spark.jars.packages","com.datastax.spark:spark-cassandra-connector-assembly\_2.12:3.1.0")\**

**.getOrCreate()**

**sc = spark.sparkContext**

**sc.setLogLevel("ERROR")**

1. To deal with the amount of data associated with the weather.com dataset, you decide to start with a smaller sample data set. The dataset contains 7 days of weather information for major US Cities, with one row being weather information for a single city on a single day. Load the dataset Located at **/home/jovyan/datasets/weather/weather.json** and use printSchema() to inspect the schema.

**weather = spark.read.json("file:///home/jovyan/datasets/weather/weather.json")**

**weather.printSchema()**

1. Look at rows of data in the sample data set. Profile the data to determine what should be used as the partition and cluster key:
   1. First: Find the minimal candidate key - which columns serve as a key for each row?   
      NOTE: You can NOT use 2020census as that is a population figure and coincidentally unique.

**Just by trying to understand the data, we can say that a candidate key can be composed by date, city, and state. There are 1600 distinct records, when we select distinct this 3 columns, which is the same number as the number of rows in our dataset,**

**print(weather.select("date").distinct().count())**

**print(weather.select("city", "state").distinct().count())**

**print(weather.select("date","city", "state").distinct().count())**

* 1. Next: Prove your key works, in Spark:
     1. Get a count of rows in the entire DataFrame.

**print(weather.count())**

* + 1. Get a count rows when you select your key columns and use distinct() to remove duplicates.

**print(weather.select("date").distinct().count())**

**print(weather.select("city", "state").distinct().count())**

**print(weather.select("date","city", "state").distinct().count())**

* + 1. If the row counts are the name, that’s a candidate key. include the code and output in the screenshot.
  1. A Cassandra row key consists of a partition and cluster key.   
     For this example, use the column that will guarantee to be storing data in increasing order over time (append only) as your cluster key. The other column (or columns) should be the partition key

**In this case, the partition key will be State and City, while the cluster key will be date since it is the column that when storing data, it will be increased by time.**

1. With your keys figured out, its time to create your table. Using the CQL Shell, write an CQL Query to create a table called **daily\_city\_weather**. Include all columns in the source data set, and make sure to set your partition and cluster keys, as designed. Show the CQL query and the output in the screenshot. Include an additional screenshot of the describe command on this table.  
   ADVICE: Write your create table in a text editor then paste it into CQL, as the command line can be a tad unforgiving.

**sql = '''**

**create table glab.daily\_city\_weather (**

**census2020 int,**

**city text,**

**condition text,**

**weather\_date date,**

**description text,**

**dew\_point decimal,**

**latitude decimal,**

**longitude decimal,**

**moon\_phase decimal,**

**pct\_clouds int,**

**pct\_humidity int,**

**pressure int,**

**rainfall decimal,**

**snowfall decimal,**

**state text,**

**temperature\_day decimal,**

**temperature\_eve decimal,**

**temperature\_max decimal,**

**temperature\_min decimal,**

**temperature\_morn decimal,**

**temperature\_night decimal,**

**timezone text,**

**uv\_index decimal,**

**wind\_direction\_deg int,**

**wind\_gust decimal,**

**wind\_speed decimal,**

**primary key ( (state, city), weather\_date )**

**);**

**'''**

**from cassandra.cluster import Cluster**

**with Cluster([cassandra\_host]) as cluster:**

**session cluster.connect()**

**session.execute("sql")**

1. Write spark code to save the json dataframe into your Cassandra table. Make sure the column names are the same. Read the data back out and make sure you have the same number of rows in the dataframe and in the Cassandra table. This will be further proof that your Cassandra row key is setup correctly. Provide spark code to save the data to Cassandra and then a screenshot of a select statement and output in the CQL Shell.

**w = weather.toDF( "census2020", "city",**

**"condition",**

**"weather\_date",**

**"description",**

**"dew\_point",**

**"latitude",**

**"longitude",**

**"moon\_phase",**

**"pct\_clouds",**

**"pct\_humidity",**

**"pressure",**

**"rainfall",**

**"snowfall",**

**"state",**

**"temperature\_day",**

**"temperature\_eve",**

**"temperature\_max",**

**"temperature\_min",**

**"temperature\_morn",**

**"temperature\_night",**

**"timezone",**

**"uv\_index",**

**"wind\_direction\_deg",**

**"wind\_gust",**

**"wind\_speed")**

**w.write.format("org.apache.spark.sql.cassandra") \**

**.mode("Append") \**

**.option("table", "daily\_city\_weather") \**

**.option("keyspace", "glab") \**

**.save()**

1. Write a CQL Shell query to get the condition, description and daytime temperatures for "Syracuse, NY" include all dates.

**SELECT city, state, weather\_date, condition, description, temperature\_day**

**FROM daily\_city\_weather**

**WHERE city = 'Syracuse' AND state = 'New York';**

1. Write the same query as 7. But using Spark SQL. Register the data From Cassandra as the Temp View **daily\_city\_weather**, then use Spark SQL To filter on “Syracuse, NY”. Instead of showing the output, **explain()** the spark query to prove the filter is being passed-through to Cassandra (The filter should NOT be happening in spark – Welcome to big data country!)

**w.createOrReplaceTempView("daily\_city\_weather")**

**query = '''**

**SELECT city, state, weather\_date, condition, description, temperature\_day**

**FROM daily\_city\_weather**

**WHERE city = 'Syracuse' AND state = 'New York';**

**'''**

**spark.sql(query).explain()**

1. Your company would like to now allow users to find cities where it is raining on a specific date. Specifically, they would like a query to show the city and state name, date, condition, and description for only those cities where it is raining on the given date. Write this query in Spark or Spark SQL. Which Cassandra filters are used? Show with explain and highlight in your screenshot.

**query = '''**

**SELECT city, state, weather\_date, condition, description**

**FROM daily\_city\_weather**

**WHERE condition = 'Rain' and weather\_date = '2021-10-23';**

**'''**

**spark.sql(query).toPandas()**

**spark.sql(query).explain()**

The cassandra filters that are used is only the cluster key which in this case is weather\_date.

1. Run the same query in 9 from the CQL command line, obviously it requires ALLOW FILTERING. Figure out how you can do an index or materialized view to avoid a costly ALLOW FILTERING operation. Include your CQL to create the index or materialized view and then include a query demonstrating it works in CQL.   
   NOTE: Our version of Cassandra and the Spark Connector does not support Materialized Views.

**CREATE MATERIALIZED VIEW daily\_city\_weather\_by\_condition\_date AS**

**SELECT city, state, weather\_date, condition, description**

**FROM daily\_city\_weather**

**WHERE condition IS NOT NULL AND weather\_date IS NOT NULL AND city IS NOT NULL AND state IS NOT NULL**

**PRIMARY KEY ((condition, weather\_date), city, state);**

**SELECT city, state, weather\_date, condition, description**

**FROM daily\_city\_weather\_by\_condition\_date**

**WHERE condition = 'Rain' AND weather\_date = '2021-10-23';**

**IMPORTANT:** When you are finished with the lab, execute:

PS:> docker-compose stop

To turn off all running services, then shut down your Azure Lab instance.