Final Project Report

Tatshini Ganesan
Caleb Hamblen
Rishi Mohan
Yi-Fang Tsai

MSDS697 Distributed Data Systems
Prof. Mahesh Chaudhari

Goals:

We wanted to take data from BigQuery with the goal of effectively integrating Google Cloud computing with MongoDB Atlas and Spark all while using Airflow DAGs to schedule and orchestrate jobs on a routine basis.

We wanted to follow the spirit of the project and complete everything on-cloud, so we sought to use Dataproc for Spark, Cloud Composer for Airflow, BigQuery for data warehousing, and GS as a storage backend. Our staging database would be MongoDB Atlas.

We also wanted to build a model that could add to our aggregations and be able to tell some kind of story. We wanted to simulate a real world cloud machine learning deployment as closely as possible - the focus being primarily on the interaction between cloud services and orchestration over model tuning.

Datasets and Selection:

We used three datasets all from BigQuery's set of public data records.

- The first dataset tabulates all San Francisco 311 service requests from July 2008 to the present, and is updated daily (link). 311 is a non-emergency number that provides access to non-emergency municipal services. This has close to 6.8 million records.
- The second dataset contains incidents from the San Francisco Police Department (SFPD) Crime Incident Reporting system, from January 2003 until 2018 (link). This only has about 2 million records.
- Lastly we used a dataset from SF Fire Department service calls (link). This data includes fire unit responses to calls from April 2000 to present and is updated daily. This has over 6.5 million records.

We predominantly used the 311 service requests and SF Fire Department data as the police data is no longer being updated. We chose this data because we wanted a large data set that was being continuously updated so that we can show that the DAG is updating our collections in MongoDB Atlas. Since the data source is the city of San Francisco, the aggregations are somewhat more relevant to us.

Overview of Pipeline:

Our pipeline consists of two stages - periodic data retrieval from BigQuery, and triggering aggregation and modeling jobs. These are done separately and on different time intervals, as rebuilding a model in the real-world is computationally very expensive and should not be done as frequently.

Our data transfer from BigQuery to our MongoDB staging database is designed to avoid re-inserting previously retrieved records. After solving the cold-start problem by manually migrating data, we designed a four-part system.

1. The first task in the DAG executes a query against a specified collection in MongoDB. This task returns a single string that represents the "newest" entry, i.e. the most recent datetime stored in a specified field. Since each of our entries in all datasets contain some form of timestamp, this helps us select only "newer" entries. We'll call this `newest_record`.
2. The second task in the DAG executes a query against a specified dataset in BigQuery, selecting rows that are *newer* than `newest_record`. We select these entries into a `temp_table` that we can then retrieve the contents of at our leisure.
3. The third task in the DAG retrieves the contents of `temp_table` and stores them as an XCom in the Airflow executor memory. Although it is a bit hacky, storing the retrieved content in-memory allows us to immediately pass it to the next step without additional network calls.
4. Finally, the last task in the DAG inserts the rows from the XCom into the appropriate collection.

This fetch-and-insert process is completed parallelly for both of the two daily-updating datasets that we chose to utilize in this project. In this way we avoid redundant network and compute expenses but still facilitate rapid data propagation from BigQuery to MongoDB. This job is triggered by an Airflow DAG daily at 9AM PST.

The second portion of the pipeline uses PySpark and the Spark-MongoDB connector on Dataproc to retrieve our stored records and generate aggregations. These PySpark jobs are

triggered via a second Airflow DAG at a different schedule to simulate the less-frequent interval of model rebuilds. We implemented several Spark jobs:

- One job uses the `sfpd_incidents` dataset described earlier and generates some aggregate statistics on crime by year, street, etc. This simulates the process of report generation.
- A second job uses the `sffd_service_calls` dataset described earlier and uses Facebook's Prophet model to forecast the monthly number of paramedic-involved calls that required transportation to a hospital. This simulates re-fitting a time-series model with updated data. We compared the performance of an ARIMA and a Prophet model in a distributed notebook on Dataproc to write this job.

Analytics & Aggregations:

Aggregation 1 (`sfpd_incidents`):

Group by stolen auto-mobiles and theft from automobiles, get a count for each, and calculate the total percent for each out of all incidents every month for the data range of 2000 to the present .

Aggregation 2 (`service_requests`):

Data Preparation: The 'year' column is added by extracting the year from the 'created_date' field. Grouping and Aggregation: Service requests are grouped by 'year' and 'category', and counts are aggregated. Identifying Most Common Category: For each year, the most common category is determined based on the highest count. Calculating Percentage: The percentage of the most common category is calculated relative to the total count for each year. Result Presentation: The analysis provides insights into the most common category and its percentage for each year.

Aggregation 3 (`sffd_service_calls`):

The analysis identifies the top call type for each month from 2018 onwards in the "sffd_service_calls" collection. Data is grouped by "year", "month", and "call_type", with counts calculated and sorted in descending order. Further grouping by "year" and "month" determines the top call type and its count. The resulting DataFrame includes columns for "year", "month", "top_call_type", "top_call_type_count", and "percentage". This analysis aids in decision-making and resource allocation for public safety operations.

Aggregation 4 (`sffd_service_calls`):

Filtered to be just an ambulance that actually went to the hospital. It then aggregates the count of calls per month-year and forecasts future call volumes using ARIMA and Prophet models, finally storing the forecasted data back into MongoDB Atlas.

Query Times Analysis:

We evaluated the performance of several queries on the original BigQuery public datasets and our staging collections in MongoDB Atlas, and found some interesting results. Although BigQuery's primary purpose is to act as a data warehouse (thus response speed is not quick), it outperformed MongoDB on several more complicated aggregation queries that we tested, in particular the aggregation described in Aggregation 4 above.

BigQuery was able to return the results in around 2.8 seconds, while MongoDB took closer to 5 seconds to return the documents we requested. One key difference in this case could be that the BigQuery dataset has indices created and available on the columns required by the query and was thus able to aggregate and return results faster, while the indices on our MongoDB instance were not sufficient and thus the system had to perform a full column scan.

We observed an interesting issue with MongoDB Atlas serverless when performing some queries after inserting new results via our ingestion DAG - sometimes we would have to allow some time (between 15 minutes and 3 hours!) to start seeing the newly inserted results in queries; aggregation pipelines and simple `find` operations alike. Perhaps this is a consequence of the backend scale-to-zero functionality of MongoDB serverless; good to keep in mind when considering database configurations for production purposes.

Conclusions:

Our project aimed to integrate Google Cloud computing with MongDB Atlas and Spark, orchestrated by Airflow DAG's. Through our pipeline, we successfully achieved our goals by retrieving data from BigQuery and storing it in MongoDB Atlas, then performing aggregations and modeling tasks using Spark on Dataproc. Throughout this project, we gained valuable experience in uploading data from BigQuery to Google Cloud Storage, orchestrating workflows with Airflow, implementing Spark and MongoDB aggregations, creating MongoDB Atlas clusters, and executing SQL queries on MongoDB dataframes.

These experiences have equipped us with a deeper understanding of distributed data systems and cloud computing technologies, preparing us for real-world scenarios where efficient data processing and analysis are vital.