Solving the 2021 DEBS Grand Challenge using Apache Flink

Baiqing Lyu
baiqing@bu.edu
Boston University
Boston, USA

Mina Morcos
minawm@bu.edu
Boston University
Boston, USA

Snigdha Kalathur
srk22@bu.edu
Boston University
Boston, USA

ABSTRACT

The DEBS Grand Challenge is an annual event in which different event-based systems compete to solve a real-world problem. For the year 2021, the challenge is computing information given air quality sensor data. Due to the pandemic many factories are forced to close down, and the aim is to find out the cities that have improved the most in air quality and cities that have achieved the longest sequence of good AQI values. This paper aims to solve the above challenges using Apache Flink, an open-source, unified stream-processing and batch-processing framework developed by the Apache Software Foundation.

ACM Reference Format:
Baiqing Lyu, Mina Morcos, and Snigdha Kalathur. 2021. Solving the 2021 DEBS Grand Challenge using Apache Flink. In *Proceedings of ACM Conference (Boston University)*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The DEBS Grand Challenge is a yearly event in which participants are given event-based data and a task to solve by using stream data processing techniques. The 2021 DEBS challenge consists of air quality sensors, and the goal is to compute the cities that improved in air quality and the cities that had the longest streak of a "good" AQI rating. Due to the pandemic, many factories were forced to shut down therefore consequentially the air pollution dropped as a result. This paper uses Apache Flink [1, 2] to connect with competition data sources, and uses custom windows to compute the solutions.[1].

The data provided by the competition includes a list of geo-locations that contains what cities are located at what locations, a batch of data with a custom data size containing PM2.5 and PM10 particle concentrations, and other options to adjust for local latency, accounting for any network overhead.

As for the reasoning of the competition, the outbreak of COVID-19 in 2020 has created numerous social, economic, and environmental repercussions. As governments announced lockdowns to minimize the spread of the virus, many businesses closed their doors and regular commuters stayed at home, resulting in a reduction in the amount of air pollution caused by traffic and business operations. In fact, the Air Quality Index (AQI) has improved throughout the world since lockdowns began. [4]

The goal is to compute two separate tasks, the first task is to figure out which cities improved the most in air qualities upwards to the top 50 cities, using the provided data. The second task is to find out which cities have the longest streak of good air quality using the provided data. Due to the data coming in batches, it acts as a data source as the batches come in one after another, therefore the answers for both tasks may change as more batches come in. This nature then requires a framework that can efficiently handle the batches of data, while performing transformations on them to determine the outputs of the two tasks.

2 BACKGROUND

Apache Flink is an open-source stream processing framework that allows for efficient computation of real-time events. Its advantages include reliable and stable performance, fast data processing, easy-to-use APIs, and an open-source community that enables development and help from all developers. As Flink is a mature and ready-to-use framework, we decided to use it for solving this year's grand challenge. Flink uses "operators", acting like layers of processing logic on an assembly line manipulating data coming from the stream.

Since each Flink operator can have an arbitrary amount of parallel instances working with the same logic, how data is passed down requires design considerations beforehand. This kind of specific partitioning of data after a transformation is supported by Flink, with the most common ones being shuffle, key by, rebalance, rescale and broadcast. Shuffle partitions data randomly downstream, and broadcast outputs every element to every partition. Key by allows for a specific part of the transformed data to be hashed and sent downstream to a specific instance. Rebalance uses the round-robin algorithm to equally distribute data among partitions, however, this will induce high memory usage. Rescale is essentially rebalance on a smaller scale, as it partitions elements round-robin to a sector of partitions. Rescale allows for lower memory usage compared against rebalance, but it sacrifices some performance in obtaining an equal distribution. These types of partitioning are important since it is necessary to prevent data re-computation and missing data during a transformation.

For our specific needs, we created operators and used two similar custom windows for our application. They are forms of Flink "global windows," which are windows that retain all objects as it gets input

[1]Due to time shortage, our submitted implementation may differ slightly from the design choices we present here. Our design choices have developed as we were working. We came to some realizations, but we didn't have time to implement them fully. We instead did quick fixes to what we had. These quick fixes should yield a very similar, if not an identical, output to what we describe here.

in the stream. Windows are a type of object that creates a finite bucket of data that can be computed within a stream of a supposed infinite amount of inputs[3]. This allows for computation to happen as if the stream was segmented into separate pieces of finite data while ingesting the continuous data stream.

The data is then partitioned with key by and rebalance methods, as we want to ensure each measurement with a specific city will be sent downstream to the correct operator, we need key by to achieve that result. Rebalance provides the best performance for data distribution, and since our design is not overly complex, the memory usage is worth it for the performance.

The properties of the custom windows are then manually defined using custom "triggers" and "evictors" (see subsections below). For both windows, we created custom triggers and the evictor using Java and the Flink Java framework.

2.1 The window trigger

In Flink, the "trigger" of a window is a piece of logic that is responsible for deciding when to invoke our window function or our window logic. For example, a timed trigger is a piece of logic we utilized. The window function can be invoked every second or any arbitrary time. In addition, it can also be triggered on every element received, which is mostly what we used.

2.2 The evictor

The evictor of the window decides what elements to drop out of the window. In our sliding window, an element participates in the calculation as long as it is still in the window. The evictor is used to determine when an element is considered to be outside the window and therefore is no longer needed. If that condition triggers, it will then be removed.

An evictor can be understood as being the opposite side of the trigger. If the trigger is the end of the sliding window, then the evictor is its beginning. Vice versa, if the trigger is the beginning of the sliding window the evictor will be the end.

We used the `evictAfter` function (as opposed to `evictBefore`). It runs after the process window function (as opposed to running before it).

2.3 The process window function

The process window function is the main logic of the window. We have defined its end (/beginning) when we defined the trigger, and we have defined its beginning (/end) when we defined the evictor. Now we need to define what it does every time it gets fired.

3 DESIGN

In this section, we discuss how we designed our solution. The next section discusses how we implemented the design in more detail.

Our solution is a data-stream-based solution. We model the incoming batches of data as a stream of data. Therefore, the first thing that we do is retrieve the batches of data. They are in the form of Measurement objects defined by the competition. We also retrieve an array of locations. The next thing we do is to calculate the city of every Measurement. A Measurement object has a longitude and a latitude. With the retrieved array of locations, it is possible to determine where that measurement is located. After locating which city this latitude and longitude pair belongs to, each Measurement object should either have a city associated with it or should have no city associated with it if no location was determined. The next step is that we simply filter out those measurements that have no city associated with them, and pass valid measurements downstream to be processed.

The next step is to transform the stream of Measurements into a stream of Snapshots. We use a window that triggers every batch received. Its size is at least 24 hours of data (according to the timestamps marked on the inputs). The window advances every watermark timestamp modulo 5 minutes. The output of this window is what we call a SnapshotDictionary object. It contains a watermark timestamp and a dictionary of cities. The watermark timestamp is the "latest timestamp modulo 5". The city dictionary (keyed by city) maps to the corresponding total p1 and p2 values of this year and their count, as well as the total p1 and p2 values of last year and their count.

Based on the data contained in the SnapshotDictionary, we then calculate the AQI values for each city in the SnapshotDictionary object. The stream is then split into query 1 and query 2, which will be processed respectively for each query's needs.

For query 1, there is another similar custom window. Its logic is to triggers every batch. And that means that it triggers every input to it. The evictor should keep the length of the window less than or equal to five days. The process function loops over the whole window. In the loop, we loop over all the countries. So, for each country, we calculate the sum of max(AQIp1, AQIp2) for this year and last year, as well as their count. We keep the latest AQI values of p1 and p2 of this year as well. We use the sum and the count to calculate the averages for this year and last year. We subtract them from each other to get the improvement. We then sort the list, and we finally return the top 50 cities along with their latest AQI values.

For query 2, we just have a process function that receives every SnapshotDictionary object from the stream. For each country, if the maximum AQI is less than 50,000, we keep the streak going (or start a new streak). Otherwise, we stop the streak and we store the latest timestamp at which we received a reading from each city. This helps us invalidate the inactive ones. Note that we adjust the width of the histogram to be the difference between the first-ever timestamp and the input's watermark. This is for the case when we have received data that is less than seven days. We finally divide the calculated width of the histogram into sections. We compare the streak's length against a section (simply using division). We get an array of such comparisons. We then count it and output the counts as a percentage.

4 IMPLEMENTATION

In this section, we discuss the fine details of the solution we presented.

4.1 Measurement retrieval

The measurements are stored using gRPC, which requires a generated protobuf stub, provided by the competition. The protobuf file lays out the structures and protocols needed to communicate. During the communication protocol, there is a sequence of latency adjustment pings to compensate and calculate network overhead in an attempt to get more accurate results in throughput. We then get the array of locations from the server, which is stored in a static variable. We then get the batches of sensor measurements. By wrapping each measurement in the batch in a wrapper measurement object, it is then sent downstream to be processed. The process of grabbing data through the gRPC stub is then created into a rich source function.

```
 DataStream<Team8Measurement> measurements
= env.addSource(grpc).rebalance();
```

4.2 Calculating the city

First we do a simple map operation on each measurement.

```
 measurements.map(new MapCity())
```

This map operation takes as input the measurement's longitude and latitude coordinates. We have an array of cities and their corresponding polygons. Each city can have one or more polygons. Here is an example of how a location looks like in its protobuf definition:

```
message Location{
string zipcode = 1;
string city = 2;
double qkm = 3;
int32 population = 4;
repeated Polygon polygons = 5;
}
```

We see that each location data structure contains a list of repeated polygons, which contains repeated amounts of longitude and latitude points. Now, the map function scans the whole array of polygons and tries to find the polygon that this point is in. If the point is found in the scanning process, we set the value of the city of the measurement to be the corresponding city of that polygon.

4.3 Filtering out the measurements with no city

Some measurements have a location that does not belong to any city. This is expected as we confirmed some points cannot be triangulated to any given polygons that consist of cities. Therefore, we simply filter those out and only pass down data with confirmed city locations.

```
 DataStream<Team8Measurement> calculateCityAndFilter
= measurements.map(new MapCity())
.filter(m -> !m.city.equals("CITYERROR"));
```

4.4 The Measurements to Snapshots window

After that, we then completely transform the objects of our stream. Our stream at this point is in objects called "Team8Measurement" objects. It will be transformed into objects called "SnapshotDictionary" objects. In later steps, we will be dealing with "Snapshot-Dictionary" objects.

```
 DataStream<SnapshotDictionary> snapshots =
calculateCityAndFilter.windowAll(GlobalWindows.create())
.trigger(new TriggerFiveMinutes())
.evictor(new EvictFiveMinutes())
.process(new MeasurementsToSnapshots());
```

4.4.1 The window trigger. For this window, the proposed idea is to trigger the window on every batch received. Our stream divided the batch into Team8Measurement objects. However, the logic of this window waits for all the elements of the batch to be received by it, and then fires a trigger.

4.4.2 The evictor. The Team8Measurement objects contain timestamps, and the evictor evicts elements relative to their timestamps. This means that it scans the elements' timestamps, and decides to evict them based on it.

The rule is this: the evictor keeps the window size at least 24 hours. This is our proposed implementation:

The evictor should look at the element with the highest timestamp. Then take the modulo 5 minutes of that timestamp and keep all elements within 24 hours of that "highest timestamp modulo 5 minutes". To summarize, the evictor drops all elements less than "('highest timestamp' modulo 5 minutes) - 24 hours".

The modulo 5 acts to ignore the most recent elements that do not complete 5 minutes. These elements are not considered in the computation and are just buffered until more elements are received past the next 5 minute time marker.

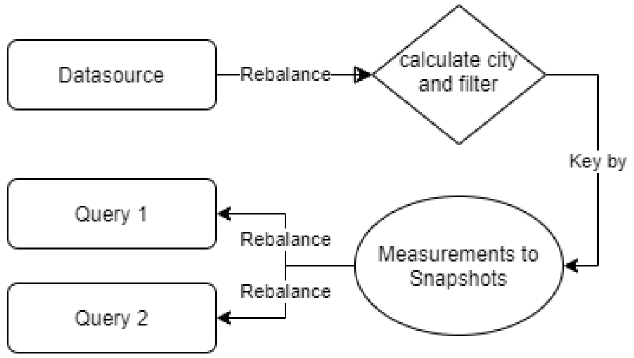
4.4.3 The process window function. The process window function looks at the most recent 24 hours of data modulo 5 minutes. That means it looks at the "most recent timestamp" value modulo 5 minutes. We only work with the data between this "timestamp modulo 5 minutes" and "timestamp modulo 5 minutes - 24 hours".

The data passed down is a list of Team8Measurement objects. We divide the objects by city. Then, for each city, we calculate the sum of all p1 values and the sum of all p2 values, and their count (the count is the same for p1 and p2).

The same logic applies for "last year" measurements. We shift everything by 365 days.

The output is a SnapshotDictionary object. This object contains a watermark timestamp and dictionary of cities. The watermark timestamp is the "latest timestamp modulo 5" that we have discussed. And the dictionary is keyed by city. Each city maps to its corresponding total p1 and p2 values of this year and their count, and the total of p1 and p2 values of last year and their count.

An abstract overview of the application can be seen in the figure below:



4.5 AQI calculation

We use a simple map operator on the snapshots to calculate the AQI. We don't change the type of data. We simply loop over the dictionary of cities. For each city, we calculate the AQI values for p1 and p2, for this year and last year. According to competition requirements, we compute the AQI values using those two particle concentrations, then take the max as the final value. Remember that we have calculated the sum and the count, and so, we use them to compute the average. We then calculate the AQI over these averages.

```
 DataStream<SnapshotDictionary> calculateAqi
= snapshots.map(new CalculateAqi());
```

4.6 Query 1

Query 1 is another window. It runs over the snapshots objects that we have made.

```
 calculateAqi.windowAll(GlobalWindows.create())
.trigger(new TriggerEveryElement())
.evictor(new EvictLastElement5Days())
.process(new SnapshotsToImprovement());
```

4.6.1 The window trigger. This trigger is simple. We need to send an output every batch. And the previous window sends an output every batch. Therefore we trigger on every input.

In code, all we do is `return TriggerResult.FIRE;`

4.6.2 The evictor. There are 1440 5 minute segments within 5 days. We use this information and keep the length of the window equal to 1440. Note that there can exist objects that are only there to trigger the function to send an output. They are either duplicate snapshots or just markers. Those should be ignored.

4.6.3 The process window function. The function's input is 5-minute snapshots for at most 5 days, plus duplicates/markers. There is an object received on every batch. That is because the output is expected on every batch. Therefore, the input object is either a 5-minute snapshot or just a marker/duplicate object. Marker/duplicate objects just make our window function send an output.

The whole window is then looped over because the size of the window should be 5 days (plus duplicates or markers). We have a condition in the loop that lets us ignore duplicates/markers. In this loop, we loop over all the countries in the SnapshotDictionary object. For each country, we calculate the sum of max(AQIp1, AQIp2) for this year and last year, as well as their count. We also keep the latest AQI values of p1 and p2 of this year.

The sums and the counts are then used to calculate the averages for this year and last year. We subtract them from each other to get the improvement. The list will then be sorted, and return the top 50 city outputs as the final result. If there are not 50 cities yet it will return the entire list.

4.7 Query 2

Query 2 is a process function that runs on every input.

```
calculateAqi.process(new SnapshotsToHistograms());
```

We have these if conditions:

For each country: if the maximum AQI is less than 50,000, keep the streak going (or start a new one). Otherwise stop the streak.

We also store the latest timestamp at which we received a reading from each city, in order to be able to ignore the inactive ones. We also adjust the width of the histogram to be the difference between the first-ever time stamp and the input's watermark. We limit it to be up to 7 days.

Finally, our solution then divides the width of the histogram into sections and compares the streak's length against a section (using division). We get an array of comparisons and count them. The final counts get outputted as a percentage in the results.

5 RESULTS

For our Flink application, we used a machine with 12 cores and 32 GB of memory and a Flink instance with the max parallelism set to 12. We ran Flink version 1.12 on an Arch Linux installation with kernel version 5.11. The run took about 17 minutes, and in total received and processed 330,000 batches of size 10,000. This means that in total we processed around 3.3 billion measurements.

During the 15 minutes time window provided by the competition, the Flink application was able to process 330,000 batches and produced around 20 results. During the running process, we observed that up to 40% of points did not end up being in any city. This observation was made by looking at how many messages entered the city filter operator and how many were sent out.

Accounting for the drop, that means we still process around 1.98 billion measurements, since we know the competition time frame for the server side is 15 minutes, that means the Flink application would have processed 2.2 million measurements per second.

As for the output, we had three cities that were produced but not found in the sample result data set. In addition, we also had on average around 19% difference in calculating the AQI. We got this result by looking at our city AQI output and comparing it to the sample results provided by the competition website. This may be due to how we are taking calculations for the AQI, and the rate at which we ingest data. There could also be a possibility that we had an issue within the city locator logic code that can result in dropping more measurements than it actually should be.

6 CONCLUSION

In this paper, we presented the design and implementation of a custom window Apache Flink application built to solve the AQI calculation and city ranking task. Our main focus was to implement a working Flink program that would resolve data from the challenger API, and a streaming flow that can handle large amounts of data in real-time.

We think that there are some improvements that can be made. One of them is sending markers down the stream instead of duplicate objects. This can be more efficient. We use duplicate objects in the case where we receive a batch, but the watermark doesn't advance. We need to output a result every batch, so the duplicate object lets us output a result in this case.

In addition, we are still unsure if the output amount rate is correctly implemented as specified by the competition. Although we tried our best to follow all noted instructions, there were still some issues, mainly the lower than expected batch output rate despite processing massive amounts of data. Still, by showing the amount of data we were able to process, we hope to demonstrate the efficiency of Flink, as even running on consumer-grade hardware was able to output very respectable results.

REFERENCES

[1] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1718–1729.

[2] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[3] F. Hueske and V. Kalavri. 2019. *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications.* O'Reilly Media, Incorporated. https://books.google.com/books?id=64GHAQAACAAJ

[4] Open Data Stuttgart. 2021. . https://sensor.community/en/