CS 631- Final Project Report - Structured Streaming

TEJAS CHINTALA, University of Waterloo, Student Id: 21012861

ACM Reference Format:
Tejas Chintala. 2023. CS 631- Final Project Report - Structured Streaming. In . ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

The use of advanced analytical methods to large, varied datasets comprising unstructured, semi-structured, and structured data with the goal of extracting valuable patterns, insights, and information is known as big data analytics. It usually entails gathering, storing, and analyzing enormous amounts of data to find insightful patterns and insights. Datasets are defined by their volume, velocity, variety, and, occasionally, truthfulness and variability.

Using big data tools has a significant impact on several businesses. For instance, evaluating huge datasets in the healthcare industry can result in better patient outcomes, tailored treatment, and more effective healthcare delivery. Big data analytics in banking makes it possible to detect fraud, better manage risks, and improve client relationships. It is also used by tech firms to improve user experiences, streamline processes, and spur innovation.

The recent exponential growth in data has fueled the demand for advanced processing methods and structures. Numerous sources, such as social media interactions, internet purchases, sensor data from Internet of Things devices, scientific studies, and more, are contributing to this data explosion [6]. Because of this, the sheer amount, velocity, and variety of data generated makes traditional methods of data processing and analysis insufficient.

2 SCOPE

Big data processing necessitates the use of distributed and parallel computing frameworks, like Apache Spark and Hadoop. These frameworks spread the workload over several nodes, making it possible to process large datasets efficiently. This facilitates the application of analytics tools and machine learning algorithms to identify patterns, correlations, and trends in the data.

2.1 Spark Structured Streaming

In contrast to Apache Flink and Kafka, Spark Structured Streaming is a streaming API that is built on the Spark SQL engine[3], offering high throughput and being scalable and fault-tolerant[5]. Compared to other open-source streaming APIs, it provides a high-level API for stream processing and was first introduced in Apache Spark in 2016. With Spark's SQL and DataFrame APIs, structured streaming automatically incrementalizes queries on static datasets. For data sources and sinks, the API provides a straightforward transactional model that by default permits "exactly-once" computation[4].

Internally, a micro-batch processing engine is used by default to process Structured Streaming queries. This engine breaks down data streams into smaller batch jobs, allowing for exactly-once fault-tolerance guarantees and end-to-end

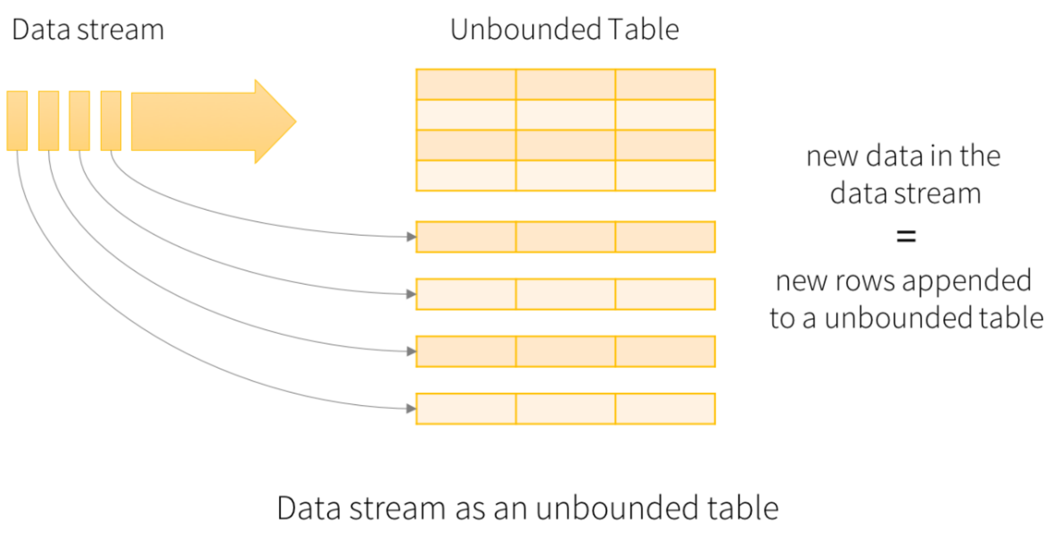
Data stream
Unbounded Table
new data in the data stream = new rows appended to a unbounded table
Data stream as an unbounded table

Fig. 1. Data stream as an unbounded table (https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-modes).

latencies as low as 100 milliseconds. Utilizing Spark Streaming's discretized streams execution model, the micro-batch mode gains access to its advantages, including dynamic load balancing, rescaling, straggler mitigation, and fault recovery without requiring a complete system rollback[2].

The data stream is treated as a table that is continuously appended. This results in a model of stream processing that is highly comparable to a model of batch processing. Spark executes the streaming computation as an incremental query on the unbounded input table, even though it can be expressed as a typical batch-like query on a static table [1]. The entire table is not materialized as it updates the result by reading the most recent data from the streaming data source, processing it piecemeal, and discarding the original data. It only stores the essential intermediate state data that is needed to update the outcome. Figure 1 depicts how the data stream is treated in an unbounded table.

There are three basic parts to spark streaming:

- `Input sources` - Connection to various input sources to generate data like Apache Kafka, Amazon Kinesis, File sources etc.
- `Streaming Engine` - Compute the transformations on the data coming from the input sources.
- `Sink` - It stores output data from the streaming engine.

In this report, we present a demonstration of the effective utilization of Spark Structured Streaming for real-time analysis of streaming data, providing instantaneous and continuous insights. This implementation was executed seamlessly within the Google Colab environment, and you can access the notebook through the following link.

3 METHODOLOGY

3.1 Dataset Description

The data used for this report is from a Kaggle dataset called the New York Times - Covid-19 Dataset. Although this Kaggle link has 5 csv files, we use only the “us_counties.csv” for our streaming implementation. This csv file has 6 columns:

- `date` - Recorded date ranging from 20-01-2020 to 12-05-2022
- `county` - The name of the county
- `state` - the name of the state for that particular county
- `fips` - county fips or the unique 5-digit code for that county
- `cases`- number of Covid-19 cases for that particular county on that date
- `deaths`- number of Covid-19 deaths for that particular county on that date

3.2 Streaming Overview

The dataset is ingested into a Spark data frame to facilitate Spark transformations. During the inspection of column data types, it is observed that the "date" column is in date format. To demonstrate streaming capabilities, the date column is converted to a timestamp datatype with the format **YYYY-MM-DD HH:MM: SS**.

To showcase Spark transformations, an aggregation is performed on the data frame. The goal is to obtain the count of COVID-19 cases and deaths for each state on a day-to-day basis. This aggregated data provides valuable insights into the impact of Covid-19-related tragedies. To enhance the real-time aspect of the analysis, the main CSV file is partitioned into multiple subsequent CSV files. Each of these files represents a streaming input source and contains data on COVID-19 cases and deaths for all counties in the USA, organized by date. This approach not only highlights the transformation capabilities of Spark but also allows for more dynamic and real-time analysis of the COVID-19 situation, as the data is now structured in a way that mirrors the streaming nature of the information. These subsequent CSV files **(total of 1690)** present in this directory will now portray the streaming input source needed for Structured Streaming.

Now, we establish the input source by specifying the directory containing multiple CSV files. Our job initialization involves leveraging the readStream functionality, allowing us to process each file as a stream. Given the substantial volume of data, we configure the "maxFilesPerTrigger" parameter to 10. This setting ensures that each micro-batch will encompass 10 files, effectively managing the data influx. Additionally, we explicitly define the schema to ensure uniform data types across all columns, aligning them with the main data frame. Since we haven't specified a trigger duration, the next micro-batch will commence as soon as the preceding one concludes or when new data becomes available in the source.

Following the data ingestion, a critical aggregation step is undertaken. Similar to the previous aggregation, our goal is to ascertain the total number of cases and deaths per state. However, a subtle modification is introduced in this process—now, the data is grouped solely based on the state column. This adjustment enables us to derive comprehensive insights into the cumulative cases and deaths for each state.

Once all the computations are executed on the data, we define the sink responsible for storing the output in external systems using the writeStream method. It's essential to specify the destination for the data within the streaming sink. Here, we employ the memory format, which stores the output in an in-memory table. Furthermore, the sink's output mode is set to “complete,” ensuring the entire result is written to the in-memory table at once. A query name is established to facilitate interaction with the in-memory table.

To inspect how the data is streamed, we initialize a for loop to showcase the first 100 batches of data and observe how the query evolves with each execution, reflecting actions based on the input stream of data. The code snippet below provides an overview:

```
#Setting a input source by creating readStream
stream = spark.readStream.schema(schema).option("maxFilesPerTrigger", 10)\
          .csv("input source file directory")

#Creating the Transformation/Aggregration for the stream
total = stream.groupBy("state")/
        .agg(F.sum("cases").alias("Total Cases"), F.sum("deaths").alias("Total Deaths"))/
        .sort(F.asc("state"))

#Setting the output sink
output_sink = total.writeStream.queryName("total")\
        .format("memory")\
        .outputMode("complete")\
        .start()

# Checking the in-memory table getting updated with the incoming micro-batches with Spark SQL interactions.
for x in range(100):
  sql = spark.sql("SELECT * FROM total")

  if sql.count() > 0:
    print(f"Batch: {x+1}")
    sql.show()
  else:
    print(f"Batch: {x+1}")
    print("No data passed")
  time.sleep(0.3)
output_sink.stop() #Shutting the stream down
```

This streaming job continuously updates the aggregated counts of total cases and deaths for each state in real time, dynamically responding to incoming data from the specified input source. This approach provides a valuable mechanism for conducting ongoing, up-to-the-minute analysis of COVID-19 statistics across different states, ensuring that the reported figures stay current and allowing stakeholders to make informed decisions based on the latest information available.

For a more in-depth analysis and trend identification, leading COVID-19 dashboards often rely on statistics computed over specific time periods. To facilitate this deeper analysis, we leverage window operations within Spark Streaming. The different types of window aggregations available are:

Table 1. Total number of Cases and Deaths per State (Batch 25)

state	Total Cases	Total Deaths
Alabama	10030367	191158
Alaska	1098655	5356
Arizona	16211704	327778
Arkansas	6211616	101364
California	67582505	1067103

Table 2. Total number of Cases and Deaths per State (Batch 100)

state	Total Cases	Total Deaths
Alabama	67827958	1222514
Alaska	7498620	35300
Arizona	106511682	2140690
Arkansas	41885584	684266
California	442514630	6827495

Table 3. Tumbling Window: Number of Cases and Deaths per 14 day period for Ohio (Batch 25)

start	end	state	total_cases	total_deaths
2020-03-05 00:00:00	2020-03-19 00:00:00	Ohio	130	0
2020-03-19 00:00:00	2020-04-02 00:00:00	Ohio	4049	86
2020-04-02 00:00:00	2020-04-16 00:00:00	Ohio	24281	916
2020-04-16 00:00:00	2020-04-30 00:00:00	Ohio	39116	1698
2020-04-30 00:00:00	2020-05-14 00:00:00	Ohio	82971	4516

Table 4. Tumbling Window: Number of Cases and Deaths per 14 day period for Ohio (Batch 100)

start	end	state	total_cases	total_deaths
2020-03-05 00:00:00	2020-03-19 00:00:00	Ohio	298	0
2020-03-19 00:00:00	2020-04-02 00:00:00	Ohio	14364	284
2020-04-02 00:00:00	2020-04-16 00:00:00	Ohio	74673	2798
2020-04-16 00:00:00	2020-04-30 00:00:00	Ohio	191916	8679
2020-04-30 00:00:00	2020-05-14 00:00:00	Ohio	307739	16977

- `Tumbling Window` - Fixed-sized and non-overlapping windows, where each element is associated with a single window. In our case, we set a window duration of 14 days to capture data within distinct 14-day intervals.
- `Sliding Window` - Overlapping windows, necessitating the specification of a sliding offset and interval. Similar to the tumbling window, we set a window duration of 14 days. However, a sliding offset of 7 days is introduced to define overlapping intervals.

To compute the number of cases and deaths within each 14-day time period, we maintain the job parameters consistent with the previous aggregation setup. The main change is in how these metrics are computed for individual states, with Ohio as the focus.

4 RESULTS

Tables 1 and 2 illustrate the initial aggregation process, focusing on obtaining the total number of cases and deaths per state for the first 5 states in alphabetical order. An intriguing observation emerges as we progress from batch 25 to the final batch of 100: there's a notable exponential increase in both cases and deaths. This dynamic reflects the real-time nature of our analysis, where each subsequent micro-batch from the input source promptly updates the in-memory

Table 5. Sliding Window: Number of Cases and Deaths for Ohio per 14 day period with 7 day slide interval (Batch 25)

start	end	state	total_cases	total_deaths
2020-02-27 00:00:00	2020-03-12 00:00:00	Ohio	6	0
2020-03-05 00:00:00	2020-03-19 00:00:00	Ohio	146	0
2020-03-12 00:00:00	2020-03-26 00:00:00	Ohio	1291	17
2020-03-19 00:00:00	2020-04-02 00:00:00	Ohio	6689	137
2020-03-26 00:00:00	2020-04-09 00:00:00	Ohio	14031	381

Table 6. Sliding Window: Number of Cases and Deaths for Ohio per 14 day period with 7 day slide interval (Batch 100)

start	end	state	total_cases	total_deaths
2020-02-27 00:00:00	2020-03-12 00:00:00	Ohio	10	0
2020-03-05 00:00:00	2020-03-19 00:00:00	Ohio	298	0
2020-03-12 00:00:00	2020-03-26 00:00:00	Ohio	2890	32
2020-03-19 00:00:00	2020-04-02 00:00:00	Ohio	14364	284
2020-03-26 00:00:00	2020-04-09 00:00:00	Ohio	40141	1147

table, facilitating the continuous monitoring of Covid-19 statistics. This real-time updating mechanism is particularly advantageous for handling large volumes of streaming data.

Moving forward, Tables 3 and 4 showcase the output of the tumbling window operation. This provides a comprehensive view of the number of Covid-19 cases and deaths within distinct 14-day periods. Notably, the fixed data received in each window, synchronized with the micro-batch updates, enables nuanced trend analysis, offering insights into the evolving patterns of the pandemic.

Tables 5 and 6 introduce the sliding window operation, a distinctive feature where the computation spans a 14-day interval, but due to a sliding offset of 7 days, some data spans across multiple windows. An illustrative example is found in Table 6, where the interval starting on “2020-02-27 00:00:00” accumulates 10 cases. The subsequent interval, commencing on “2020-03-05 00:00:00,” is just 7 days apart from the start of the previous cycle, effectively incorporating data from the prior window showcasing the overlapping window nature. Furthermore, the sliding window operation with a 14-day window duration and a 14-day slide interval produces an equivalent result as in tumbling window operations depicted in Tables 3 and 4. Furthermore, the sliding window operation with a 14-day window duration and a 14-day slide interval produces an equivalent result as in tumbling window operations depicted in Tables 3 and 4.

5 CONCLUSION

This report showcases Spark Structured Streaming’s extensive functionality, showcasing how well it handles intricate use cases by skillfully utilizing its window operations and APIs. Because of its adaptable source and sink capabilities, which guarantee consistency and transactional compatibility with a wide range of storage systems, its intrinsic simplicity of use places it in a position of strength.

Spark Structured Streaming is unique because of its operational strength, which makes it possible to process continuous streams of data reliably and efficiently. In addition, it combines easily with the larger Spark ecosystem, making it easier to add machine learning libraries and run interactive queries on streaming states. With the help of this seamless integration, users can create complex business applications that unlock a multitude of opportunities for real-time insights and advanced analytics thanks to the combination of streaming capabilities and the larger Spark ecosystem.

REFERENCES

[1] Databricks. -. *Apache Spark™ Tutorial: Getting Started with Apache Spark on Databricks.* Retrieved - from https://www.databricks.com/spark/getting-started-with-apache-spark/streaming#:~:text=Update%20Mode%3A%20Only%20the%20rows,is%20equivalent%20to%20Append%20mode

[2] Haoyuan Li Timothy Hunter Scott Shenker Matei Zaharia, Tathagata Das and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. *SOSP 13: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Nov. 2013). https://doi.org/10.1145/2517349.2522737

[3] Cheng Lian Yin Huai Davies Liu Joseph K. Bradley Xiangrui Meng Tomer Kaftan Michael J. Franklin Ali Ghodsi Michael Armbrust, Reynold S. Xin and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. *SIGMOD '15: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), 1383–1394. https://doi.org/10.1145/2723372.2742797

[4] Joseph Torres Michael Armbrust, Tathagata Das and Burak Yavuz. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. *SIGMOD '18* (2018), 601–613. https://doi.org/10.1145/3183713.3190664

[5] Apache Spark. -. *Structured Streaming Programming Guide.* Retrieved - from https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#operations-on-streaming-dataframesdatasets

[6] Rashid Mehmood Yasir Arfat, Sardar Usman and Iyad Katib. 2020. Big Data Tools, Technologies, and Applications: A Survey. *Smart Infrastructure and Applications* (2020), 453–490. https://doi.org/10.1007/978-3-030-13705-2_19