International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 12 Issue: 01 | Jan 2025 www.irjet.net p-ISSN: 2395-0072

Modern Data Engineering with Apache Spark Structured Streaming and Apache Flink

Sreyashi Das

Netflix, USA

Abstract - *Over the last decade, Apache Spark Structured Streaming and Apache Flink have emerged as leading frameworks for real-time data processing. This paper proposes advanced techniques for designing and building reliable streaming components that seamlessly integrate into data pipelines, thereby simplifying data transformations and enhancing query performance. By leveraging Spark Structured Streaming's capabilities, such as micro-batch processing, adaptive query execution, and enhanced state management, we demonstrate how to ensure scalability and resilience in handling large-scale streaming applications. Additionally, Flink's support for temporal table joins, stateful stream processing, and dynamic table management empowers data practitioners and engineers to efficiently and reliably transform data, meeting the demands of modern data engineering teams and organizations. This paper provides valuable insights into the art of moving and transforming data, offering practical guidance for building robust streaming applications.*

Key Words: (streaming data pipelines, spark structured streaming, flink

1.INTRODUCTION

In the realm of big data, real-time data processing has become essential for organizations seeking to derive immediate insights and maintain a competitive edge. Deploying streaming data pipelines efficiently with high throughput and low latency present challenges due to complex event processing, inconsistent data quality and resource management to optimize query performance. Addressing these challenges, requires a deep-dive into how streaming data pipelines[Fig-1] are implemented for real-world applications. A comparative analysis reveals shifting strengths, with Flink's true stream processing model offering distinct advantages in latency and state management while Spark structured streaming's integration with existing data ecosystem.

This paper explores the intricacies of deploying streaming data pipelines using two prominent frameworks: Apache Spark Structured Streaming and Apache Flink. By examining their historical evolution, technical capabilities, and recent advancements, the paper provides a comprehensive understanding of how these technologies can be leveraged to overcome the inherent challenges of real-time data processing. Methodologies for building scalable, resilient streaming applications are discussed, highlighting best practices for optimizing performance and ensuring data quality. Through practical applications the benefits of each framework are illustrated, offering insights into their unique strengths and trade-offs.

Furthermore, the paper discusses the future directions of stream processing, including the integration of AI/ML and edge computing, and how these trends are shaping the next generation of data pipelines. This paper serves as a valuable resource for data practitioners and engineers, providing guidance on selecting and implementing the right streaming solutions to meet the demands of modern data-driven organizations.

2. Historical context and evolution

2.1 The emergence of Stream processing

The early 2000s marked the beginning of a shift towards real-time data processing, driven by the need for immediate insights from data as it was generated. Initial solutions were often limited in scalability and flexibility. The introduction of Apache Spark in 2009, followed by Spark Streaming in 2013, provided a robust framework for processing live data streams using a micro-batch model[Fig-3].

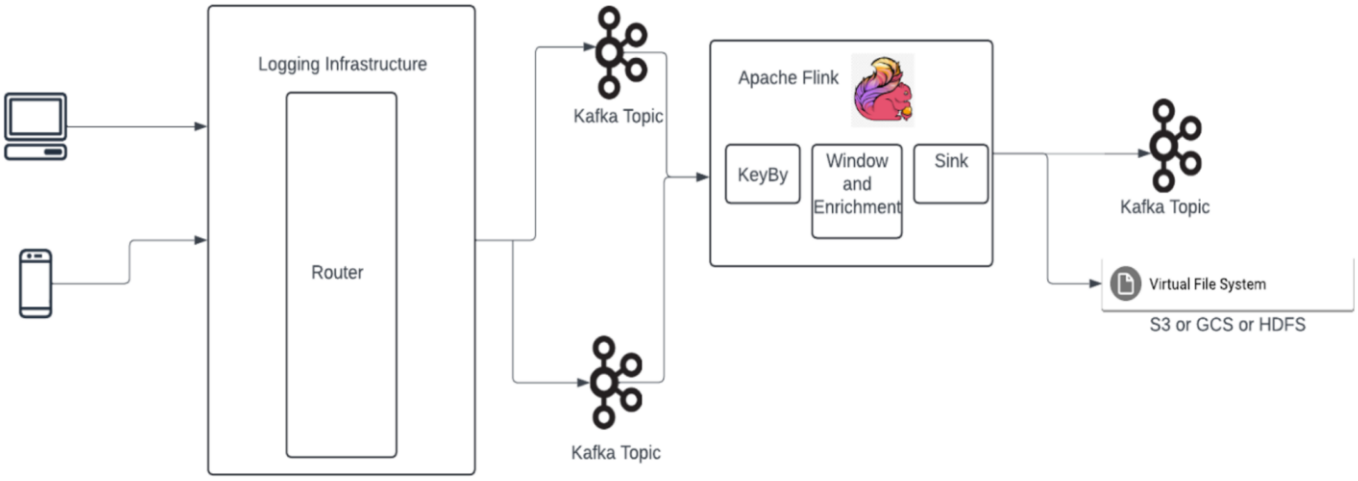
Apache Flink, initially developed as Stratosphere in 2009, introduced a true stream processing model in 2014. Flink's ability to handle data in real-time with event time semantics and stateful computations distinguished it from other frameworks.

Table -1: Key Milestones of stream processing

Key Milestones	
2013	Spark Streaming introduced micro-batch processing, enabling seamless integration with the Spark ecosystem
2014	Apache Flink released with a focus on low-latency, high-throughput stream processing
2016	Flink introduced exactly-once state consistency, revolutionizing stateful stream processing

2018	Spark Structured Streaming offered a unified and declarative API for stream processing
2020s	Both frameworks continued to evolve, enhancing state management, fault tolerance, and integration capabilities

Fig -1: Example of real-time data pipeline



3. Key aspects of Spark Structured Streaming

3.1 Declarative API

Spark Structured Streaming provides a high-level declarative API that allows users to express streaming computations in a SQL-like syntax. This approach simplifies the development of streaming applications by abstracting the complexities of stream processing. Users familiar with batch processing in Spark can easily transition to streaming applications without needing to learn a new programming model. The declarative nature of the API enables automatic optimization of query execution plans, improving performance and resource utilization.

3.2 DataFrames and Datasets

In Spark Structured Streaming, stream data is represented as DataFrames and Datasets, which are distributed collections of data organized into named columns. This representation allows streaming data to be processed using the same APIs and optimizations available for batch data. By leveraging DataFrames and Datasets, users can seamlessly integrate streaming data with Spark's rich ecosystem, including machine learning (MLlib), graph processing (GraphX), and data manipulation libraries. This integration facilitates complex data processing workflows that combine real-time and historical data analysis.

3.3 Exactly-once Processing

One of the critical features of Spark Structured Streaming is its ability to provide exactly-once processing semantics. This means that each record in the input data is processed exactly once, even in the presence of failures. This guarantee is achieved through a combination of checkpointing and write-ahead logs, which ensure that the system can recover to a consistent state after a failure. Exactly-once processing is crucial for applications that require high reliability and accuracy, such as financial transactions and monitoring systems.

3.3 Advanced Windowing

Spark Structured Streaming supports advanced windowing operations that enable users to perform time-based aggregations and analyses on streaming data. Windowing functions allow users to group data into fixed-size or sliding windows based on event time or processing time. This capability is essential for applications that require temporal analysis, such as trend detection, anomaly detection, and real-time reporting. The framework provides a variety of windowing options, including tumbling windows, sliding windows, and session windows, allowing users to choose the most appropriate windowing strategy for their use case.

4. Key aspects of Flink

4.1 True Stream Processing

Apache Flink is designed as a true stream processing framework, meaning it processes data as it arrives, rather than in micro-batches. This architecture allows

Flink to achieve low-latency processing, making it ideal for applications that require real-time insights and immediate responses, such as fraud detection, monitoring, and alerting systems. By processing each event individually, Flink can provide more granular control over data processing and time management[Fig-2], which is crucial for applications with stringent latency requirements.

4.2 Event Time Semantics

Flink offers robust support for event time semantics, which is essential for accurately processing events that may arrive out of order or with delays. Event time processing allows Flink to use timestamps embedded in the data to determine the order of events, rather than relying on the time they are processed. This capability is particularly important for applications that require precise time-based operations, such as billing systems or time-series analysis. Flink's event time semantics are complemented by its support for watermarks, which help manage the trade-off between latency and completeness by indicating the progress of event time in the stream.

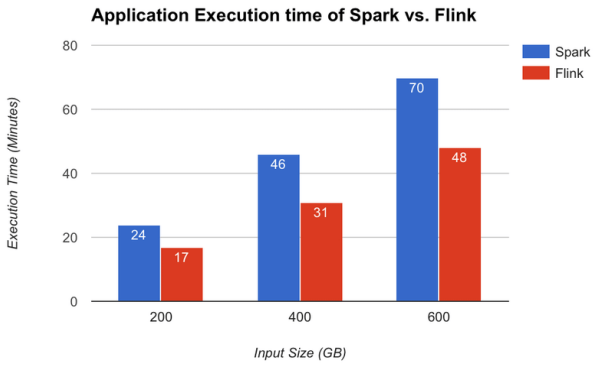
4.3 Stateful Stream Processing

Flink excels in stateful stream processing, providing advanced state management capabilities with exactly-once consistency guarantees. This means that Flink can maintain and update state information across events, enabling complex computations that depend on historical data, such as aggregations, joins, and pattern detection. Flink's state management is highly efficient, allowing it to handle large state sizes with minimal overhead. The framework supports various state backends, such as in-memory, RocksDB, and custom implementations, giving users flexibility in choosing the most suitable storage mechanism for their application's needs.

4.4 Fault Tolerance

Flink provides robust fault tolerance mechanisms to ensure reliable stream processing even in the presence of failures. It achieves this through a combination of checkpointing and distributed snapshots, which capture the state of the application at regular intervals. In the event of a failure, Flink can recover to the last consistent state using these checkpoints, ensuring that no data is lost and that exactly-once processing semantics are maintained. This fault tolerance is critical for applications that require high availability and reliability, such as financial services and critical infrastructure monitoring.

Fig -2: Execution time variability[7]



5. Comparison between Spark structured streaming and Flink

5.1 Data enrichment

Data enrichment involves augmenting streaming data with additional information, often by calling external APIs or databases. This process can enhance the value of the data by providing more context or insights.

Calling an External API

Both Spark and Flink allow calling external APIs from UDFs, but there are important considerations regarding performance and scalability.

High Request Rate: UDFs are called for every record processed, potentially leading to a high request rate to the external API.

Parallel Execution: In production, UDFs run in parallel across multiple nodes, further amplifying the request rate.

Asynchronous I/O: To maintain high throughput, especially for I/O-bound operations, consider using asynchronous I/O.

Spark Structured Streaming Example

Example: Enriching Weather Data with External API

Suppose you have a streaming DataFrame of weather station readings, and you want to enrich each record with additional weather information from an external API.

```
def call_weather_api(station_id):
# Simulate an external API call to get weather details
    response = get_weather_details(station_id)
    return response
# Define a UDF for the external API call
```

```
enrich_weather_data = udf(call_weather_api, StringType())

weather_readings = ... # Streaming DataFrame with schema: station_id, reading_time, temperature

# Enrich weather readings with external data

enriched_weather_readings = weather_readings.withColumn("enriched_data", enrich_weather_data(weather_readings.station_id))
```

In this Spark example, a UDF enrich_weather_data is defined to call an external API for each station_id. The enriched data is added as a new column to the weather_readings DataFrame.

Apache Flink Example

Example: Enriching Traffic Data with External API

Suppose you have a stream of traffic sensor data, and you want to enrich each record with additional traffic information from a REST API.

```
// Define a UDF for the external API call

public class CallTrafficAPIUDF extends ScalarFunction {

    public String eval(String sensor_id) {

// Simulate an external API call to get traffic details

       String response = getTrafficDetails(sensor_id);

        return response;

    }

}
```

```
// Register and use the UDF in a Flink SQL query

StreamTableEnvironment tableEnv = ...;

tableEnv.createTemporarySystemFunction("callTrafficAPIUDF", CallTrafficAPIUDF.class);
```

```
tableEnv.executeSql("SELECT sensor_id, callTrafficAPIUDF(sensor_id) AS enriched_traffic " +

    "FROM traffic_sensors"

);
```

In this Flink example, a UDF CallTrafficAPIUDF is defined to call an external API for each sensor_id. The enriched data is retrieved using a Flink SQL query.

Best Practices

Use UDFs Judiciously: Improperly implemented UDFs can slow down processing, cause backpressure, and stall the application.

Asynchronous I/O: Consider using asynchronous I/O for UDFs, especially when dealing with external resources like databases or REST APIs. This approach can help maintain high throughput and reduce latency.

Rate Limiting and Caching: Implement rate limiting and caching strategies to manage the request rate to external APIs and reduce redundant calls.

5.2 Data processing

Handling Late-Arriving Data

Both Spark Structured Streaming and Apache Flink support event time processing, allowing users to define time windows based on event timestamps rather than processing time. Watermarking is a key mechanism used to manage late-arriving data, setting a threshold for how long the system should wait for delayed events before considering them too late to process.

Spark Structured Streaming Example

Example: Processing E-commerce Order Events

Consider a streaming DataFrame of e-commerce order events, where each event includes an order_id, order_time, and amount. You want to calculate the total order amount in 10-minute windows, allowing for late arrivals up to 5 minutes.

```
orders = ... # Streaming DataFrame with schema: order_id, order_time, amount

# Calculate total order amount in 10-minute windows, allowing for 5 minutes of lateness
```

```
totalOrderAmount = orders \

   .withWatermark("order_time", "5 minutes")\

 .groupBy(window(orders.order_time, "10 minutes")) \

    .sum("amount")
```

In this example, the withWatermark method specifies a 5-minute lateness threshold. Events arriving within 5 minutes of the window's end will be included in the aggregation, while those arriving later will be discarded.

Apache Flink Example

Example: Analyzing Social Media Posts

Consider a stream of social media posts, where each post includes a post_id, post_time, and content. You want to

count the number of posts in 15-minute windows, allowing for late arrivals up to 2 minutes.

```
CREATE TABLE social_media_posts (
  post_id STRING,
  post_time TIMESTAMP(3),
  content STRING,
  WATERMARK FOR post_time AS post_time - INTERVAL '2' MINUTE
);

SELECT window_start, COUNT(post_id) AS post_count
FROM TABLE(
    TUMBLE(TABLE social_media_posts, DESCRIPTOR(post_time), INTERVAL '15' MINUTES))
GROUP BY window_start;
```

In this Flink SQL example, the watermark is defined in the DDL with a 2-minute lateness threshold. This allows the system to include posts arriving up to 2 minutes late in the 15-minute window count.

Windowing

Spark Structured Streaming Tumbling Window

Example: Aggregating Website Clicks

Suppose you want to calculate the total number of clicks on a website in 15-minute intervals.

```
clicks = ...  # Streaming DataFrame with schema: user_id, click_time

# Count clicks in a tumbling window of size 15 minutes
clicksByWindow = clicks \
   .withWatermark("click_time", "5 minutes") \
   .groupBy(
      window(clicks.click_time, "15 minutes")) \
   .count()
```

In this example, a tumbling window of 15 minutes is used to count the number of clicks, with a watermark to handle late data.

Apache Flink Tumbling Window

Example: Summing Sales Transactions

Consider a stream of sales transactions where you want to calculate the total sales amount every 30 minutes.

a stream of social media posts, where each post includes a post_id, post_time, and content. You want to count the number of posts in 15-minute windows, allowing for late arrivals up to 2 minutes.

```
SELECT window_start, SUM(amount) AS total_sales
FROM TABLE(
    TUMBLE(TABLE sales_transactions, DESCRIPTOR(transaction_time), INTERVAL '30' MINUTES))
GROUP BY window_start;
```

In this Flink SQL example, a tumbling window of 30 minutes is applied to sum the amount of sales transactions.

Spark Structured Streaming Sliding Window

Example: Monitoring Temperature Changes

Suppose you want to monitor temperature changes using a sliding window of 20 minutes with a slide interval of 10 minutes.

```
temperature_readings = ...  # Streaming DataFrame with schema: sensor_id, temperature, reading_time

# Calculate average temperature in a sliding window of size 20 minutes and slide interval of 10 minutes
avgTemperatureByWindow = temperature_readings \
   .withWatermark("reading_time", "10 minutes") \
   .groupBy(
      window(temperature_readings.reading_time, "20 minutes", "10 minutes")) \
   .avg("temperature")
```

In this example, a sliding window is used to calculate the average temperature, allowing for overlapping analysis.

Apache Flink Sliding Window

Example: Analyzing Network Traffic

Consider a stream of network traffic data where you want to calculate the average data rate over a 10-minute window with a 5-minute slide interval.

```
SELECT window_start, AVG(data_rate) AS avg_rate
FROM TABLE(
    HOP(TABLE network_traffic, DESCRIPTOR(event_time), INTERVAL '5' MINUTES, INTERVAL '10' MINUTES))
GROUP BY window_start;
```

In this Flink SQL example, a sliding window is used to compute the average data_rate, providing insights into network performance over time.

6. Spark vs Flink: How to choose

6.1 Latency Sensitivity:
Evaluate the importance of low-latency processing in the application. Flink's true stream processing model may be more suitable for applications requiring immediate data processing and low-latency responses.

6.2 Stateful Processing Needs:
Consider the complexity and scale of stateful computations required by the application. Flink's advanced state management capabilities may offer advantages for applications with complex stateful processing requirements.

6.3 Event Time Handling:
Assess the importance of precise event time processing and handling of out-of-order events. Flink's robust support for event time semantics may be beneficial for applications that rely heavily on accurate time-based operations.

6.4 Development and Maintenance:
Consider the ease of development and ongoing maintenance. Spark's larger ecosystem and community support may simplify development and troubleshooting, while Flink's intuitive APIs may reduce development time for stream processing tasks.

6.5 Resource Management and Cost:
Evaluate the resource management capabilities and potential cost implications of each framework. Consider how each framework's resource utilization aligns with the project's budget and infrastructure constraints.

6.6 Scalability and Throughput:
Determine the scalability and throughput requirements of the application. Both frameworks are designed to scale, but specific architectural features may influence their performance at scale.

6.6 Use Case Alignment:
Align the choice of framework with the specific use cases and industry requirements of the project. Consider how each framework's strengths align with the unique demands of the application domain.

6.7 Future Growth and Flexibility:
Consider the potential for future growth and the need for flexibility in the data processing architecture. Evaluate how each framework can adapt to evolving data processing needs and emerging technologies.

3. CONCLUSIONS

In conclusion, Apache Spark and Apache Flink stand as titans in the realm of distributed data processing, each offering a distinct set of strengths that cater to the diverse needs of modern data engineering. Spark, with its robust batch processing capabilities and extensive language support, provides a comprehensive platform for a myriad of use cases, from ETL processes to complex machine learning pipelines. Its mature ecosystem and widespread adoption make it a reliable choice for organizations seeking a versatile and well-supported framework.

Conversely, Apache Flink emerges as the vanguard of stream processing, delivering unparalleled low-latency performance and sophisticated windowing capabilities that are indispensable for real-time analytics. Flink's architecture is meticulously crafted to handle the intricacies of event time processing and stateful computations, making it the framework of choice for applications that demand immediacy and precision.

The decision to adopt Spark or Flink should be guided by a nuanced understanding of the specific requirements and constraints of the project at hand. Experienced data engineers must weigh factors such as processing paradigms, latency tolerances, iterative processing needs, and the existing technological landscape. Moreover, the expertise of the team and the strategic goals of the organization play pivotal roles in this decision-making process.

Conducting rigorous proof-of-concept evaluations and benchmarking both frameworks against real-world scenarios can provide invaluable insights into their performance and suitability. Such empirical assessments, coupled with a strategic vision, will empower organizations to make informed choices that align with their data processing objectives.

As the field of big data continues to evolve, both Spark and Flink are poised to adapt and innovate, driven by vibrant open-source communities and cutting-edge research. Staying abreast of the latest advancements and leveraging the unique capabilities of these frameworks

will enable data engineers to push the boundaries of what is possible, transforming data into actionable insights and driving the next wave of technological innovation.

REFERENCES

[1] Hesse, G., & Lorenz, M. (2015). Conceptual Survey on Data Stream Processing Systems. 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS). doi: 10.1109/icpads.2015.106

[2] Real-time Data Stream Processing - Challenges and Perspectives. (2017). International Journal of Computer Science Issues, 14(5), 6–12. doi: 10.20943/01201705.612

[3] Kolajo, T., Daramola, O., & Adebiyi, A. (2019). Big data stream analysis: a systematic literature review. Journalof Big Data, 6(1). doi: 10.1186/s40537-019-0210-7

[4]https://www.macrometa.com/event-stream-processing/spark-vs-flink

[5]https://www.dataversity.net/spark-vs-flink-key-differences-and-how-to-choose/l Intelligence Volume 82, June 2019, Pages 126147

[6]https://learning.oreilly.com/library/view/streaming-systems/9781491983867/

[7]https://www.researchgate.net/figure/Execution-time-comparison-for-different-input-sizes_fig2_309192139

[8]"Benchmarking Streaming Computation Engines at... —
Yahoo Engineering," [accessed 6-January-2016]. [Online].

Available:
http://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

[9]R. Lu, G. Wu, B. Xie, and J. Hu, "Stream Bench:Towards Benchmarking Modern Distributed Stream Computing Frameworks," in 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing. IEEE, dec 2014, pp. 69–78.

[Online].Available:http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=7027482

[10]"Apache Flink: Scalable Batch and Stream Data Processing," [accessed 6-January-2016]. [Online]. Available: https://flink. apache.org/

[11]"Apache SparkTM - Lightning-Fast Cluster Computing," [accessed 6-January-2016]. [Online]. Available: https://spark. apache.org/ [3]

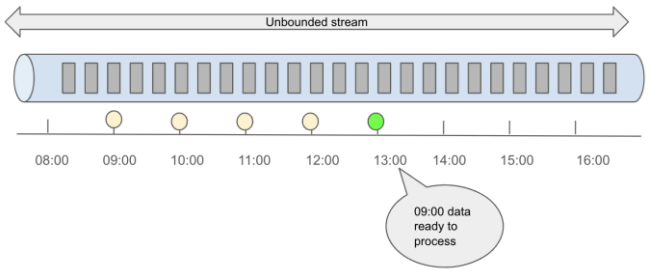


Fig -3: Micro-batch in Spark Streaming