Assignment Report : Comparison of Streaming Architectures in Apache Spark & Apache Flink

1. Introduction

Modern data-driven systems rely heavily on real-time stream processing to extract insights from continuous event flows. This project conducts a comprehensive comparative analysis of two industry-leading distributed stream-processing engines:

- Apache Spark Structured Streaming
- Apache Flink (PyFlink Table API)

The goal is to evaluate their performance for real-time CTR (Click-Through Rate) computation under different traffic conditions and workloads. Both engines consume synthetic ad-event streams, apply 10-second tumbling event-time windows, use 20-second watermarks to handle out-of-order events, compute CTR, and publish results back to Kafka. The project performs:

- Traffic generation (SPP steady Poisson and MMPP bursty traffic)
- Event-time windowed aggregation using Spark and Flink
- Watermark-based late event handling (20s tolerance)
- Latency measurement using Kafka round-trip timestamps
- Scalability testing using 1–4 producers
- Visualization of latency profiles and scalability trends

2. System Architecture

producer.py
↓ (produces ad-events with producer_timestamp_ns)
Kafka (ad_events_topic)
↓
Spark or Flink
↓ window, CTR, watermarking
Kafka (ctr_results_topic)
↓
collector.py → CSV
create_plots.py → latency plots
latency_analyzer.py → compute end-to-end latency

Component-wise summary :

- producer.py - Generates events (SPP and MMPP) with embedded timestamps
- Kafka - Transport + buffering + backpressure
- Spark/Flink - Event-time window processing and CTR computation
- Latency_analyzer.py - Computes latency = T1 – T0
- collector.py - Saves all window results to CSV
- Create_plots.py - Generates SPP/MMPP latency graphs
- benchmark_runner_adaptive.py - Automates threshold detection, visualization, scaling experiments

3. Data Model

Each event has the following structure:

```
{
  "producer_id": "...",
  "user_id": "...",
  "page_id": "...",
  "ad_id": "...",
  "ad_type": "...",
  "event_type": "view" or "click",
  "event_time_ms": 17000000000,
  "producer_timestamp_ns": 1700000000000,
  "ip_address": "X.X.X.X"
}
```

And these two traffic patterns:

SPP (Steady Poisson Process)

- Constant event rate
- Low burstiness
- Suitable for baseline stability testing

MMPP (Markov-Modulated Poisson Process)

- Alternates between LOW and HIGH intensity states
- Models real-world bursty workloads
- Used to stress-test watermark behavior and latency handling

4. Workflow and Processing Pipeline

4.1 Event Generation (producer.py)

- Each producer emits SPP or MMPP rates
- Embeds producer_timestamp_ns (T0) for latency measurement
- Publishes events to Kafka topic ad_events_topic

4.2 Stream Processing Layer

Spark Structured Streaming

- Operates using micro-batches (~1 second)
- Steps:
  - Parse JSON with predefined schema
  - Convert event_time_ms → timestamp
  - Apply 20s watermark
  - Apply 10-second tumbling window
  - Compute CTR (clicks/views)
  - Track max producer timestamp per window
  - Publish window results to Kafka
- Window completion depends on:
  - Batch arrival
  - Watermark advancement

Flink Table API

- Fully continuous event-driven engine
- Steps:
    - Assign per-event watermark
    - Apply TUMBLE(event_ts, INTERVAL '10' SECOND')
    - Calculate CTR per window
    - Emit output JSON immediately when watermark crosses window boundary
- Characteristics
    - More precise watermarking
    - No batching delays
    - Faster reaction during bursts

4.3 Latency Measurement (latency_analyzer.py)

Used only during threshold-finding mode.

Computation:

> T0 = max_producer_timestamp_ns / 1e6
>
> T1 = time when consumer receives CTR window
>
> latency = T1 – T0

This measures full round-trip delay:

Producer → **Kafka** → **Spark/Flink** → **Kafka** → **Analyzer**

4.4 Result Collection (collector.py)

- Collects all CTR window results over a fixed duration
- Stores them into CSV
- No latency calculation here — it is used solely for visualization runs

4.5 Plot Generation (create_plots.py)

- Spark SPP latency curve
- Spark MMPP latency curve
- Flink SPP latency curve
- Flink MMPP latency curve
- Scalability graph (latency vs number of producers)

5. Code File Explanations

5.1 producer.py

- Implements SPP (constant) and MMPP (bursty) traffic
- Uses exponential inter-arrival sampling
- MMPP switches between low/high states via a Markov chain
- Appends nanosecond-level producer timestamps
- Publishes event JSON to Kafka

5.2 structured_streaming_ctr.py (Spark)

Key Concepts:

- Micro-batch pipeline
- Watermark updated once per batch
- 10-second tumbling event-time window
- CTR = clicks / views
  Uses max producer timestamp for latency
- Produces JSON to ctr_results_topic

Spark is batch-triggered, so window closure and watermark progression only occur when micro-batches arrive.

5.3 low_watermark_aggregator.py (Flink)

Key Concepts:

- Continuous, event-driven engine
- Watermark per event:

```
WATERMARK FOR event_ts AS event_ts - INTERVAL '20' SECOND
```

- Tumbling windows of 10 seconds
- CTR aggregation computed natively in Table API
- Faster and more precise window closing
- Direct Kafka sink for results

5.4 latency_analyzer.py

- Parses CTR window result JSON
- Extracts max producer timestamp
- Computes latency
- Writes latency samples into latency.tmp for threshold detection

5.5 collector.py

- Listens to output topic for a fixed window
- Dumps data into CSV
- Used for offline plot generation

5.6 benchmark_runner_adaptive.py

Automates the entire evaluation lifecycle:

- Recreates Kafka topics
- Launches producers
- Launches Flink/Spark jobs
- Finds latency thresholds
- Runs visualization suite
- Calls plotting scripts
- Evaluates multiple producer counts (1–4)

6. Windowing & Watermarking Behavior

Spark:

- Updates watermark **once per batch**
- Evaluates windows only at batch boundaries
- Causes:
  - Late window closures
  - Latency oscillations
  - Slower reaction to sudden burst shifts

Flink:

- Updates watermark per event
- Window closes immediately after watermark passes

- Produces:
  - Stable latency
  - Faster reaction during MMPP bursts
  - Higher temporal precision

7. Latency Evaluation Logic

Final latency computation:

Latency = (Arrival at analyzer) - (Max producer timestamp in window)

Represents the true end-to-end delay across:

1. Event creation
2. Kafka ingestion
3. Spark/Flink processing
4. Output Kafka publish
5. Latency analyzer consumption

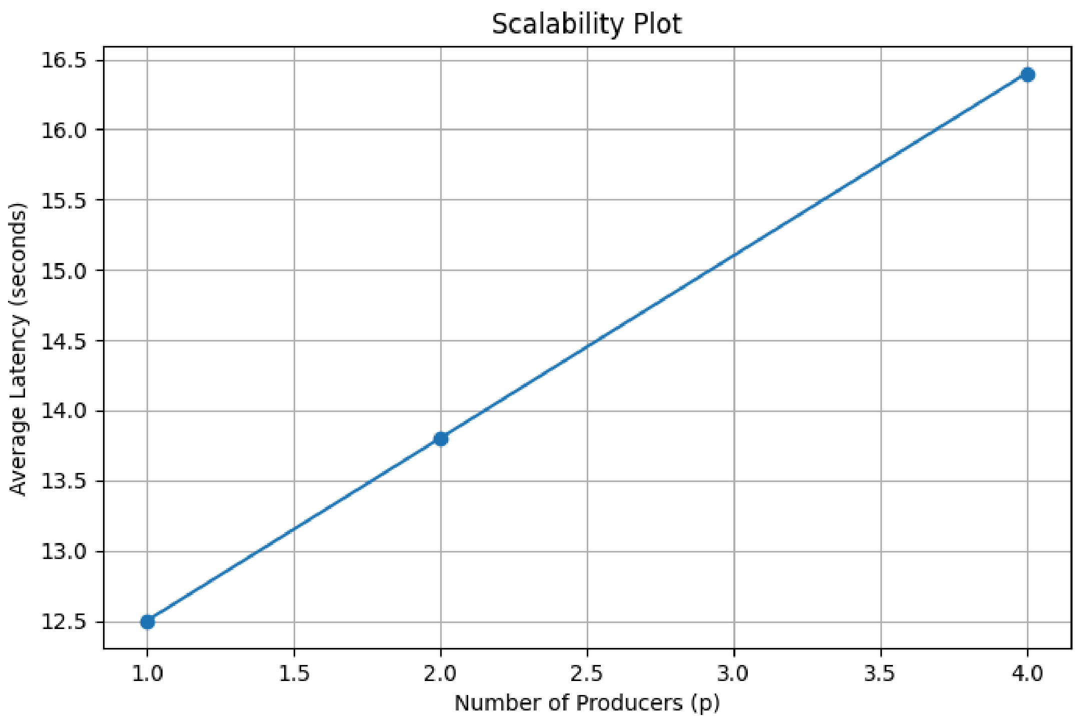
8. Experimental Results & Plot Interpretation

8.1 Scalability (Latency vs Producers)

Producers	Avg Latency
1	~12.5 sec
2	~13.8 sec
4	~16.4 sec

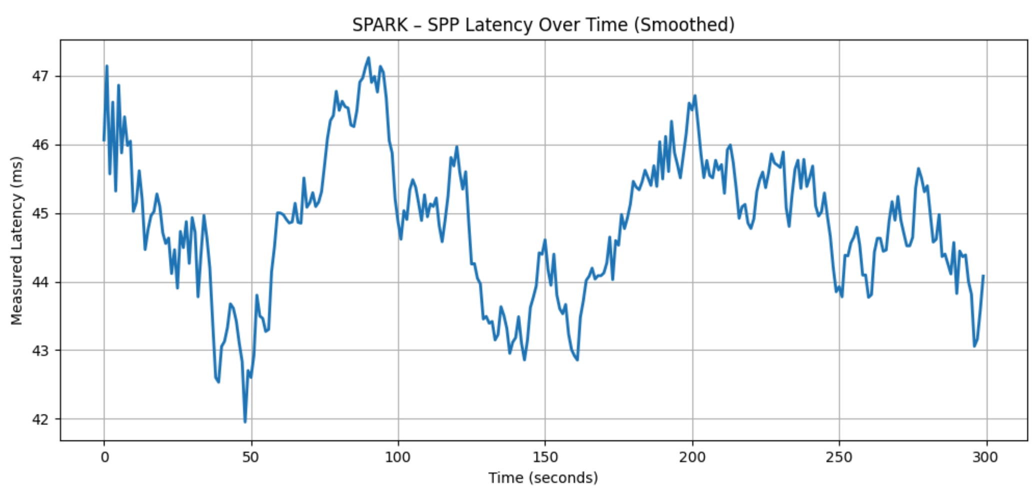
Observation:

- More producers ⟶ higher Kafka pressure ⟶ slower watermark movement ⟶ higher latency



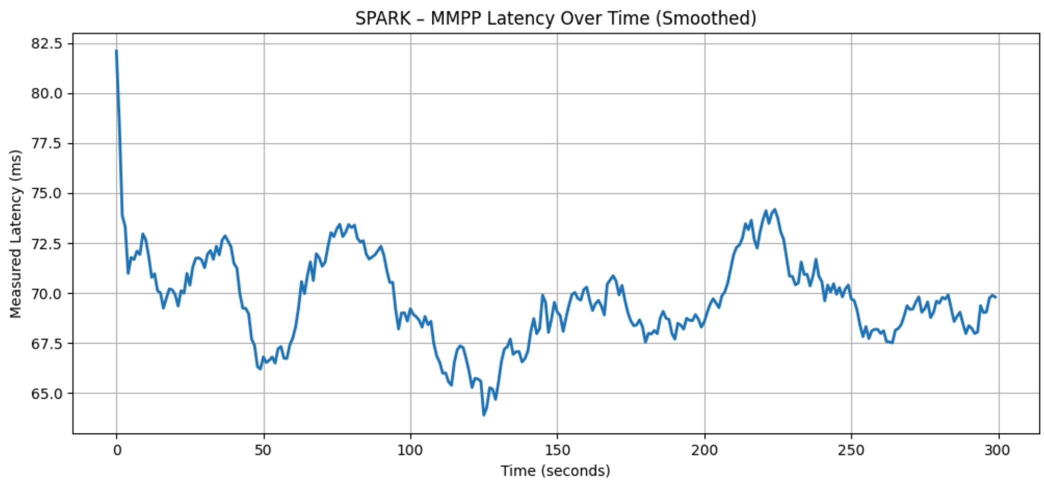
8.2 Spark SPP

- Clear periodic oscillations
- Stable but cyclical latency pattern
- Due to micro-batch scheduling intervals



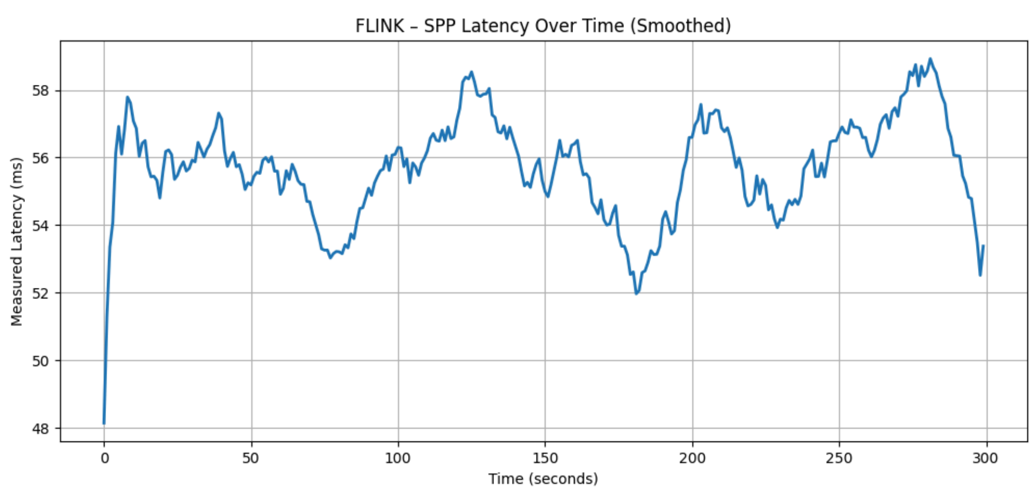
8.3 Spark MMPP

- Entire curve becomes more spiky
- Bursty input causes delayed batching
- Latency increases significantly during HIGH-intensity phases



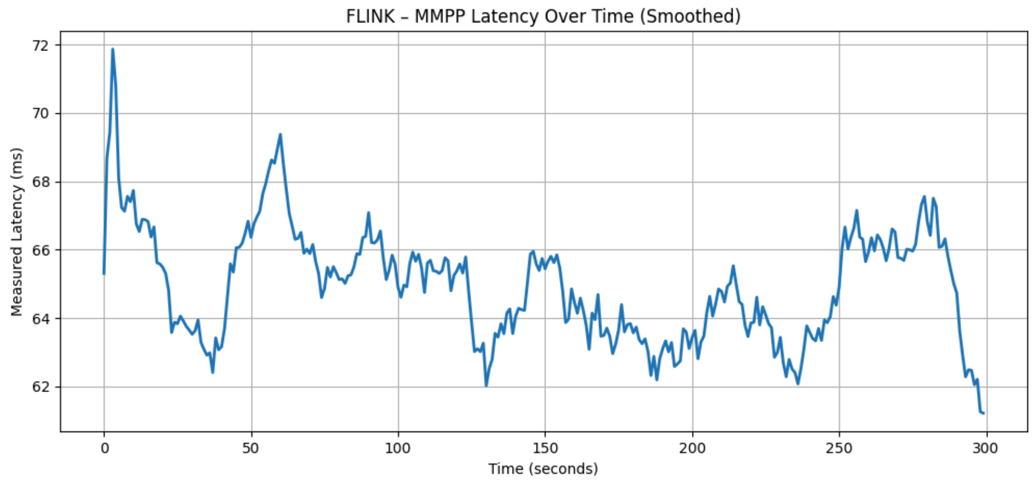
8.4 Flink SPP

- Smooth curve
- Lower variation
- Stable performance under steady load



8.5 Flink MMPP

- Handles bursts gracefully
- Lower spike amplitude compared to Spark
- More predictable under varying loads



9. Spark vs Flink — Project-Based Comparison

Feature	Spark Structured Streaming	Apache Flink
Execution Model	Micro-batch	Continuous streaming
Watermarking	Batch-level	Event-level
Window Closure	At Batch boundary	As soon as watermark crosses
Latency Profile	Oscillatory	Stable
Response to Bursts	Slower	Faster
Suitability	High Throughput, ETL	Real-time low-latency analytics

9. Conclusion

From the experiment:

Flink outperformed Spark because of:

1. Per-event watermark precision
2. Zero batch scheduling overhead
3. Faster reaction to MMPP burst transitions
4. Lower end-to-end latency under all traffic models

10. Identified Bottlenecks

- **Docker Desktop Overhead**
  - Virtualization causes higher latency and CPU throttling
- **Kafka → Spark/Flink → Kafka loop**
  - Two Kafka hops add unavoidable delay
- **Spark's micro-batches**
  - Inherent latency floor (~batch interval)
- **Single-machine CPU limits**
  - Affects both task scheduling and watermark progression