What is Spark Streaming

“**Spark Streaming**” is generally known **as an *extension of the core Spark API***. It is a unified engine that natively supports both batch and streaming workloads. Spark streaming enables scalability, high-throughput, fault-tolerant stream processing of live data streams.

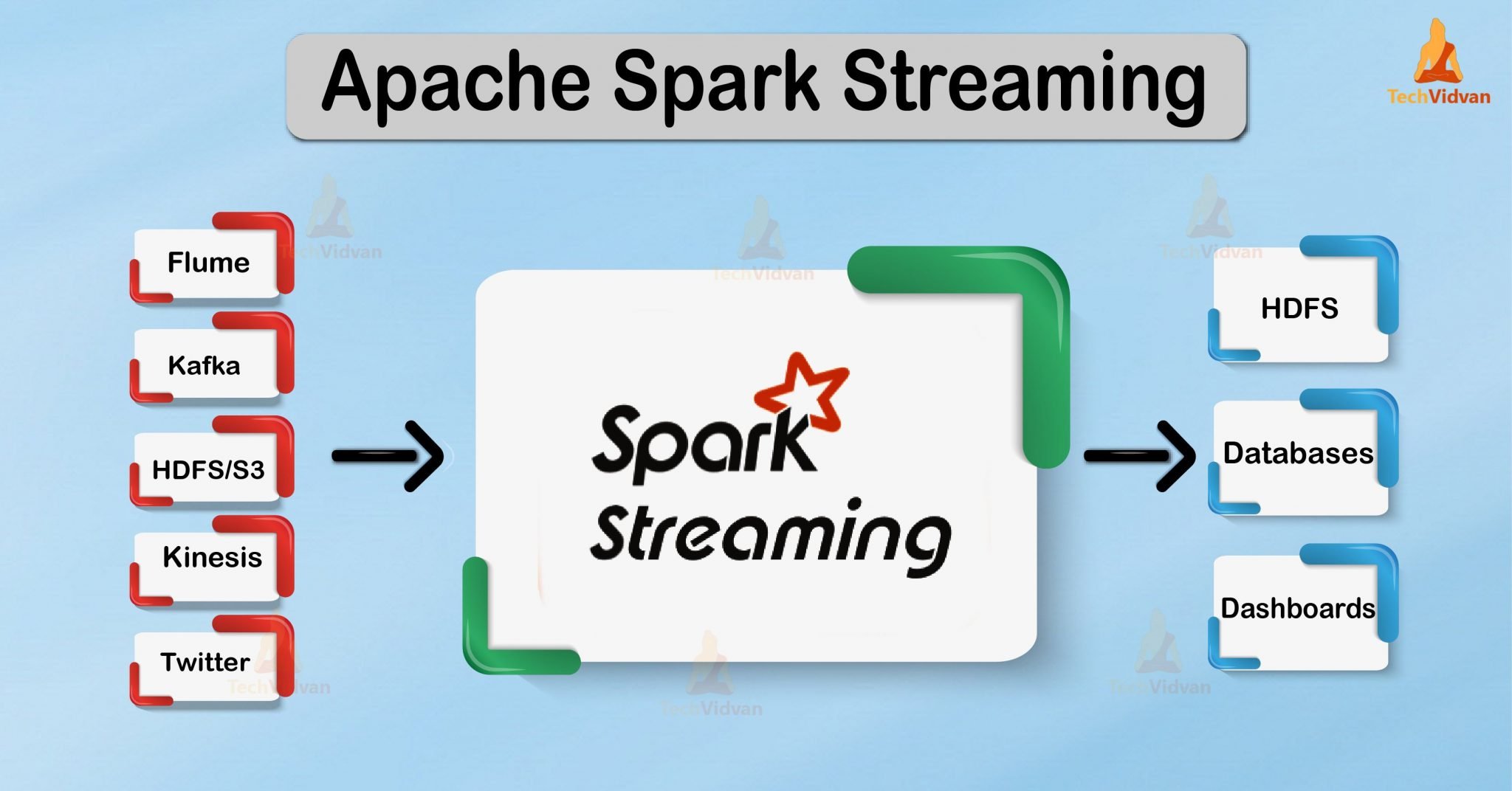
It is a different system from others. Some engines either have streaming or have similar batch and streaming APIs, yet they compile internally to different engines.

Therefore, streaming came into the picture. This model offers both execution and unified programming for batch and streaming. There are some advantages over other traditional streaming systems. Majorly, there are 4 aspects of it.

* It recovers very fast from failures and stragglers.
* Resource usage and better load balancing in spark streaming.
* Streaming data can be combined with static datasets as well as interactive queries.
* We can also integrate it with advanced processing libraries, such as SQL, machine learning, graph processing.

**Data ingestion is possible from many sources, such as Apache Flume, Kinesis, Kafka or TCP sockets. Processing is also possible by using complex algorithms expressed with high-level functions, such as map, reduce, join and window.**

Afterwards, data which is processed can be pushed out to databases, filesystems, and live dashboards. Although, can apply spark’s machine learning & graph processing algorithms on data streams.



A diagram of a computer

AI-generated content may be incorrect.

In **Spark Streaming**,

* **Input sources** = where the streaming data comes from. (TCP socket,kafka…)
* **Output sinks** = where Spark sends the processed results. (HDFS,DB,s3)

Here’s the breakdown:

**1. Input Sources (Data Ingestion into Spark Streaming)**

Spark Streaming supports two main types:

**A. Basic Sources (built-in, easy to use)**

* **File Streams** → Reads text files from HDFS, S3, or local file systems (new files only).
* **Socket Streams** → Reads text data from a TCP socket (useful for testing/demo).
* **Rate Source** → Generates data at a fixed rate (for testing).

val lines = ssc.textFileStream("hdfs://path/to/directory")

**B. Advanced / External Sources**

These require **connectors** or **integrations**:

* **Kafka** → Most common source for event streaming.
* **Flume** → Used for log aggregation.
* **Kinesis** → AWS real-time streaming.
* **Custom Receivers** → User-defined data sources.
* **MQ systems** like RabbitMQ, ActiveMQ, Pulsar (via connectors).

val df = spark.readStream

.format("kafka")

.option("kafka.bootstrap.servers", "localhost:9092")

.option("subscribe", "topic1")

.load()

**2. Output Sinks (Where the Processed Data Goes)**

**A. Built-in Output Operations (DStreams API)**

* **print()** → Print to console (debugging).
* **saveAsTextFiles() / saveAsObjectFiles()** → Save to files.
* **foreachRDD()** → Apply arbitrary actions (write to DB, API, etc.).

**B. Structured Streaming Sinks**

* **Console Sink** → For debugging (prints in batches).
* **File Sink** → Writes to HDFS, S3, or local FS in CSV/JSON/Parquet format.
* **Kafka Sink** → Send processed results back to Kafka.
* **Foreach Sink** → Fully custom sink logic (e.g., write to databases, call APIs).
* **Memory Sink** → Stores data in-memory tables for querying (useful for testing).

val query = df.writeStream

.format("console")

.outputMode("append")

.start()

In **Spark Structured Streaming**,  
the **processing mode** tells Spark **what part of the result table** should be written to the sink each time there’s new data.

**1. Append Mode**

* **What it does:** Only the **new rows** added since the last trigger are written to the sink.
* **When to use:**
  + Data is **immutable** (once written, it will not change later).
  + No need to update existing rows.
* **Example use cases:**
  + Streaming logs from Kafka to a file.
  + Sensor readings where each event is independent.
* **Restrictions:**
  + Not allowed for aggregations with non-watermarked event time unless watermark is defined.

df.writeStream

.outputMode("append")

.format("console")

.start()

**2. Update Mode**

* **What it does:** Outputs **only the rows that changed** (new rows or updated aggregates) since the last trigger.
* **When to use:**
  + You have aggregations or stateful operations where existing results can be updated.
  + You don’t need the full result every time.
* **Example use cases:**
  + Running count of events in the last 10 minutes.
  + Live dashboard metrics.
* **Restrictions:**
  + Not supported for some sinks like **file sink** (use complete mode instead).

df.writeStream

.outputMode("update")

.format("console")

.start()

**3. Complete Mode**

* **What it does:** Writes the **entire result table** to the sink every time there’s an update.
* **When to use:**
  + You need a full snapshot each time.
  + Useful for aggregations where the whole state is important.
* **Example use cases:**
  + Leaderboards (e.g., top 10 users every minute).
  + Full summary tables.
* **Restrictions:**
  + Supported only for certain sinks (e.g., console, memory, some DB sinks).
  + Not supported for Kafka sink.

df.writeStream

.outputMode("complete")

.format("console")

.start()

Watermarking

**1. Event Time vs Processing Time**

**Event Time**

* The timestamp **inside the data itself** indicating when the event actually occurred.
* Comes from the **source system** (e.g., IoT device, Kafka message, log file).
* May be delayed or out-of-order when arriving at Spark.

A sensor sends:

{"value": 45, "eventTime": "2025-08-12T21:00:00"}

Even if Spark receives it at **21:05**, the event time is still **21:00**.

**Processing Time**

* The time **when Spark processes** the data.
* Comes from the **system clock** on the Spark cluster.
* Does not account for delays or late arrivals.

Example:  
That same sensor event arrives at **21:05** — processing time is 21:05.

**Why Watermarking is Needed**

In real life:

* Events can arrive **late** (network delay, retries, batching).
* Spark needs to maintain **state** for aggregations like *count per window*.
* If Spark **never discards old state**, memory usage grows endlessly.

**Watermarking** = a mechanism to tell Spark:

“I will tolerate late data up to X minutes. After that, drop it.”

**How Watermarking Works**

* You define a **watermark delay threshold** based on event time.
* Spark will keep state for windows **up to that threshold** past the maximum event time seen.
* Events older than that threshold are considered **too late** and are ignored for aggregations.

**ey Points to Remember**

* Use **event time** for accuracy in time-based analytics.
* Use **watermarking** to handle late data + control state size.

Basic Streaming Operations: Detailed Explanations

1. Simple Streaming Transformations

Streaming transformations apply operations to data as it arrives in real-time from the source (e.g., Kafka, database, files). Examples include filtering, mapping, and restructuring records.

These transformations are often implemented through SQL or DataFrame operations. For instance, you can add metadata, derive new columns, or enrich incoming data with additional context using Spark SQL functions.

Notably, streaming transforms are valuable for both cleaning and preparing raw event data as it flows into downstream applications, without the need for batch processing.

* *Example*: Using Spark SQL to add a processing timestamp and reorganize event data as it lands.
* *Use case*: Real-time fraud detection, cleansing incoming logs, or adding contextual tags to clickstream events.

2. Stateless Operations in Streaming

Stateless operations process each incoming record or micro-batch independently, without retaining information from previous records. Since no historical context is required, stateless operations are simple, fast, and resource-efficient.

Common examples include filtering (e.g., rows where "value > 10"), selective mapping, and transformation of data fields. They make streaming pipelines straightforward but lack the ability to perform aggregations or pattern detection that span across multiple batches.

* *Example*: Filtering out low-priority messages in a log stream; mapping incoming events to a new structure.
* *Limitations*: Cannot compute running totals, distinct counts over time, or session analytics.

3. Output Modes and Streaming

Output modes dictate how and when streaming results are written or emitted to sinks (e.g., Delta Lake, console, cloud storage). There are three principal output modes in systems like Spark Structured Streaming:

| **Output Mode** | **Description** | **Typical Use Case** |
| --- | --- | --- |
| Append | Only new rows are emitted after each trigger/micro-batch. | Stateless transformations, late-arriving data handled via watermark. |
| Update | All changed rows (since last trigger) are emitted. | Stateful transformations with incremental updates. |
| Complete | All rows produced so far are emitted each time. | Aggregations, full-table snapshots. |

Watermarks are often configured with stateful operators to control latency and determine when results are finalized.

4. Monitoring Streaming Applications

Monitoring is essential to ensure streaming pipelines operate reliably and efficiently. Effective monitoring includes tracking metrics like throughput, processing latency, micro-batch delays, and error rates.

Tools such as Spark's built-in metrics system, Amazon CloudWatch, or Databricks Streaming tab provide dashboards and alerts. Additional key aspects include:

* Tracking SLA metrics like end-to-end latency and data loss rates.
* Monitoring consumer group lag in systems like Kafka to ensure timely processing.
* Using application performance monitoring (APM) and log management solutions for real-time alerting and root-cause analysis.

Enterprise-grade monitoring enables auto-scaling, timely recovery, and continuous improvement of streaming applications, ensuring SLAs are met and operational issues are promptly addressed.

1. Tumbling Windows

Explanation:  
A tumbling window divides the stream into fixed, non-overlapping intervals. Each event belongs to one and only one window. For example, with a 10-minute tumbling window, all events within each 10-minute interval are grouped together for aggregation.

* Each element falls into *exactly one* window interval.

[Download the Free Nmap Security Scanner for Linux/Mac/Windows](https://nmap.org/download.html#windows)

Ncat -lk 9999

Ncat localhost 9999

2. Sliding Windows

Explanation:  
Sliding windows overlap; each data point can belong to multiple windows (depending on slide interval). For example, with a 10-minute window and 5-minute slide, each window starts every 5 minutes, capturing events from the previous 10 minutes.

3. Session Windows

Explanation:  
Session windows group events based on session activity. The window duration is dynamic—it grows with continued activity and closes after a specified period of inactivity ("gap duration").

val sessionAggDF = df

.groupBy(session\_window($"timestamp", "30 minutes")) // Gap duration

.count()

sessionAggDF.writeStream

.outputMode("update")

.format("console")

.start()

.awaitTermination()

4. Micro-batching vs. Continuous Processing

Explanation:

* Micro-batching: Spark collects data for a short fixed interval (“batch”), processes it together—offering near real-time latency (seconds).
* Continuous Processing: Each event is processed as it arrives, offering sub-second latencies (more complex setup, fewer features in Spark).

microbatching

df.writeStream

.format("console")

.outputMode("append")

.trigger(processingTime = "2 seconds") // micro-batch every 2s

.start()

.awaitTermination()