Grow Data Skills

APACHE Spark™

What is Stream Processing?

Stream processing is a type of computing method where inputs are processed as continuous flows of data instead of in batch jobs. In other words, it allows for real-time processing of "streams" of data that are generated continuously from different sources.

Here's a detailed explanation:

1. **Real-Time Processing:** Stream processing is designed to act on real-time and streaming data with "event time" processing. This means the system can make decisions and take actions based on the current data coming in, not on data that's been stored and processed.

2. **Continuous and Infinite Data Sets:** Unlike batch processing, where data is collected over a period of time and then processed all at once, stream processing works on live, constantly updating data. It treats data as an unending stream and makes computations on the fly.

3. **Fault-Tolerance:** In stream processing, data is processed as it arrives, so if there's a failure, you don't lose all your progress. Most stream processing systems have in-built recovery mechanisms.

4. **Statefulness**: Stream processing is often stateful, which means it maintains some information about its past in order to process the present. For example, a stream processing system might keep a running total of the number of events it has seen so far.

5. **Scalability**: Stream processing systems can often handle large volumes of incoming data, and can be designed to scale with the size of the input.

6. **Windowing**: Stream processing systems often use a concept called "windowing," where they treat the data in the stream as if it were broken up into chunks (or "windows") of time. This allows them to perform calculations over those windows.

What is Stream Processing?

Grow Data Skills

Assume you are running a popular news website, and you want to track the number of views on your articles in real time. Here, every click on an article is data that can be processed. Instead of waiting to gather all the data and then processing it, stream processing would allow you to track these views as they happen.

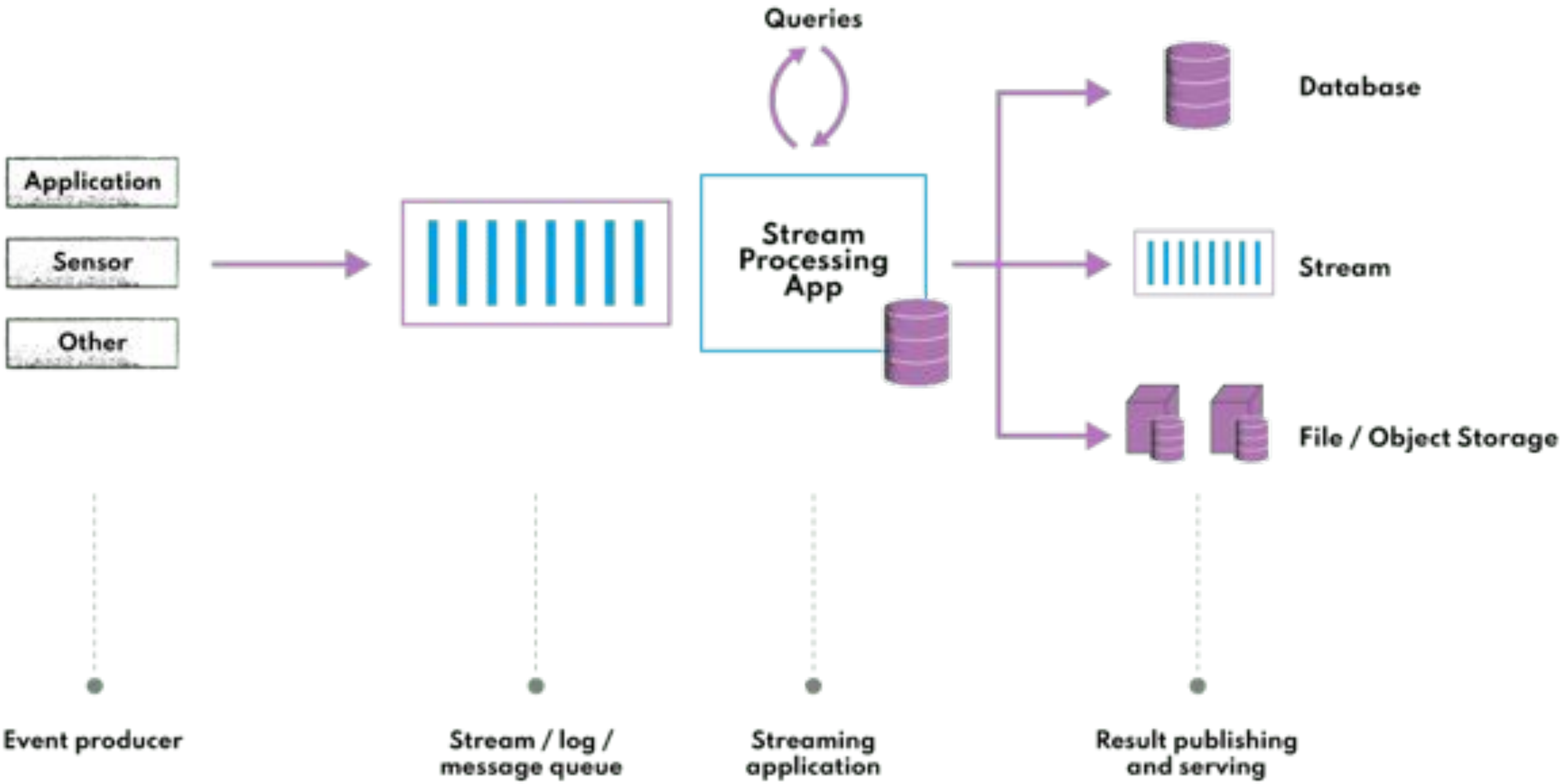
The stream processing system might work like this:

- An event is generated every time someone views an article.
- This event is immediately sent to the stream processor.
- The stream processor maintains a count of views for each article.
- When an event comes in, the stream processor updates the count for the relevant article.
- The updated count is then immediately available for use elsewhere in the system, for example, in a real-time dashboard showing the most popular articles.

In this way, you can keep a real-time track of your article views, which would not be possible with a batch processing system.

What is Stream Processing?

Grow Data Skills



Spark Structured Streaming

Apache Spark Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. It allows you to express computations on streaming data in the same way you would express them on static data. The Spark SQL engine will take care of running it incrementally and continuously, and updating the final result as streaming data keeps arriving.

Here are some key features of Spark Structured Streaming:

- **Ease of Use:** The high-level APIs in the Scala, Java, Python, and R programming languages, and built-in support for structured data using Datasets and DataFrames, make it easy to build complex streaming analytics tasks.

- **Event Time Processing:** It supports event-time based processing, which means you can handle out-of-order or late data, and perform window-based operations based on the event-time.

- **Fault Tolerance**: It guarantees end-to-end exactly-once fault-tolerance through checkpointing and Write-Ahead Logs, which means any operation you perform on your streaming data will provide the same result even after a failure.

- **Integration**: It's fully integrated with Spark ecosystem components, like MLlib and Spark SQL, enabling powerful interactive analytics, machine learning model application, and more.

- **Multiple Data Sources and Sinks**: It can consume from and produce to multiple data sources like Kafka, Flume, Kinesis, or TCP sockets, and write outputs to file systems, databases, and live dashboards.

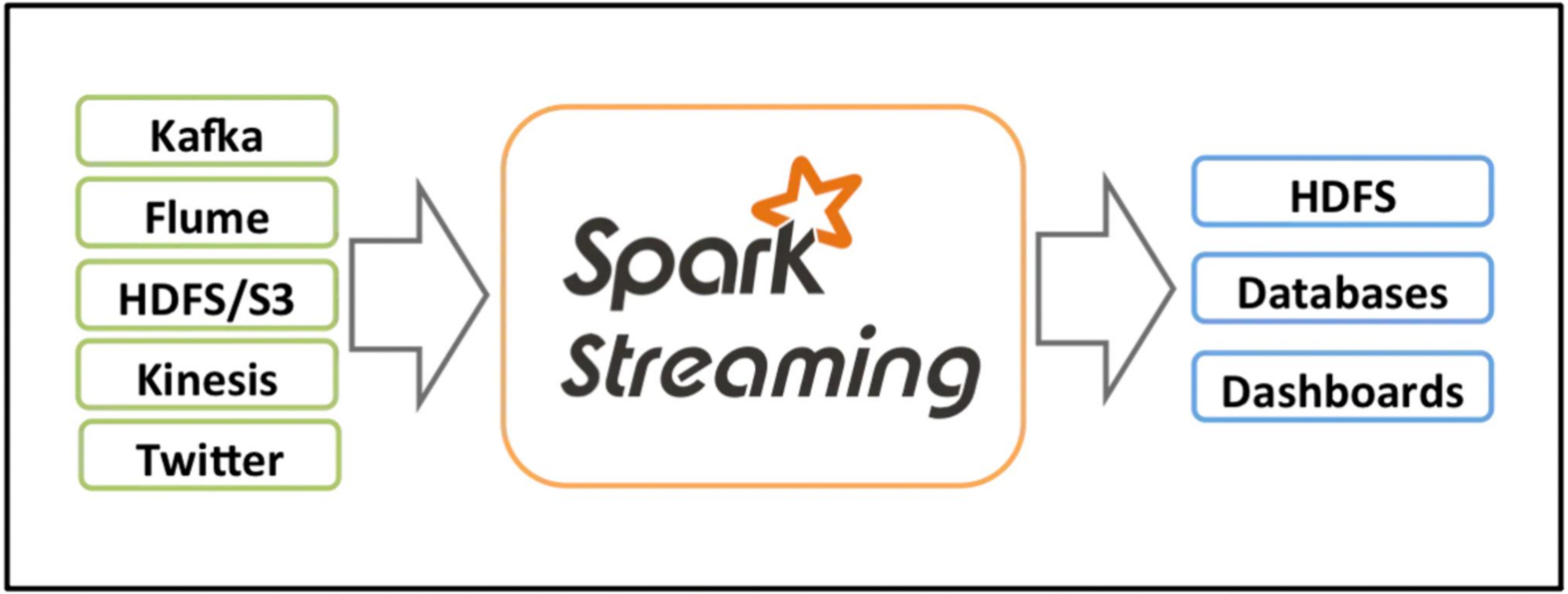
- **Backpressure Handling**: It can handle backpressure automatically, meaning it can slow down the data ingestion rate as needed.

- **Stateful Stream Processing**: It allows stateful stream processing with operations like ***mapGroupsWithState*** and ***flatMapGroupsWithState***, where you can maintain arbitrary state while continuously updating it with new information.

Spark Structured Streaming

Grow Data Skills

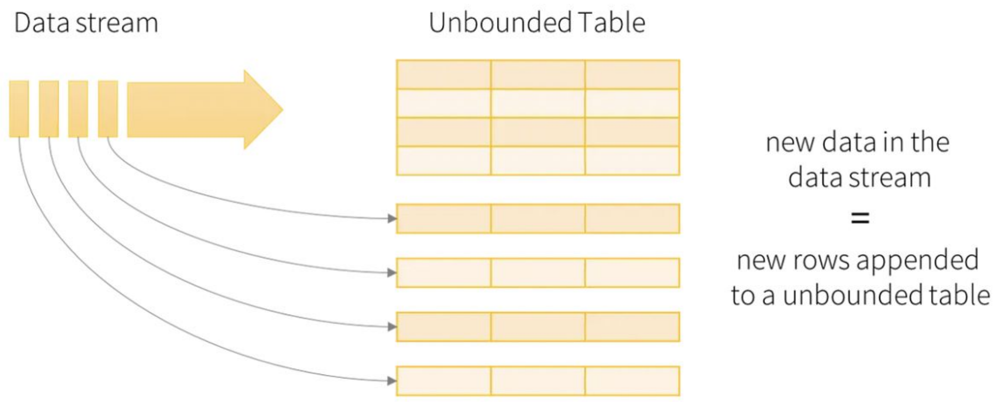
Spark Streaming has 3 major components: input sources, streaming engine, and sink. Input sources generate data like Kafka, Flume, HDFS/S3, etc. Spark Streaming engine processes incoming data from various input sources. Sinks store processed data from Spark Streaming engine like HDFS, relational databases, or NoSQL datastores.



Spark Structured Streaming

Let's conceptualise Spark Streaming data as an **unbounded table** where new data will always be appended at the end of the table.

Spark will process data in **micro-batches** which can be defined by **triggers**. For example, let's say we define a trigger as 1 second, this means Spark will create micro-batches every second and process them accordingly.



Data stream as an unbounded table

Word Count - Spark Structured Streaming

Grow Data Skills

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

# Create a SparkSession
spark = SparkSession \
    .builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()

# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()

# Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Word Count - Spark Structured Streaming

Grow Data Skills

- A SparkSession is first created using SparkSession.builder.getOrCreate()
- Then a streaming DataFrame is created on the input data (in this case, a socket source is used which listens to localhost:9999).
- This DataFrame (lines) represents the unbounded table containing the streaming data. This table contains one column of strings (value), and each line in the streaming text data becomes a row in the table.
- Next, the split function splits each line into words, and the explode function is used to turn the array of words into a dataset of words.
- Finally, groupBy is called on this dataset to count the occurrences of each word.

When you start a streaming query, Spark will continually fetch data from the source into memory and append it to an "input table". Every trigger interval, a new batch of data is appended to the input table, as if you're adding more rows to a table in a database.

For a word count, Spark will recompute the counts each time new data arrives based on the aggregation query you've defined (i.e., groupBy("word").count() in the previous example). This computation generates a Result Table, and the current result is written out to the console (or any other sink you've defined).

Output Modes - Spark Structured Streaming

There are three output modes which define what will be written to the sink:

- **Complete Mode**: The entire updated Result Table will be written to the sink after every trigger. This is what we're doing in our word count example. This mode is applicable to aggregations where the result is expected to be small, such as counts.

- **Append Mode**: Only the new rows appended to the Result Table since the last trigger will be written to the sink. This is applicable when you have a Result Table that's growing incrementally, like a running total.

- **Update Mode**: Only the rows in the Result Table that were updated since the last trigger will be written to the sink.

let's consider a simple Structured Streaming job that reads data from a stream, performs some computation on it, and writes the result to some sink. The input data is a stream of numbers, and the computation is a running count of how many times each number has been seen.

Here's what the first few batches of input and output might look like:

Batch 1: Input: [1, 2, 2, 3, 3, 3]
Output (running count): [(1, 1), (2, 2), (3, 3)]

Batch 2: Input: [1, 2, 2, 4]
Output (running count): [(1, 2), (2, 4), (3, 3), (4, 1)]

Output Modes - Spark Structured Streaming

And here's how each output mode would handle these batches:

- **Complete Mode:** In complete mode, the entire updated Result Table is written to the sink after every trigger. This means that after batch 1, the output would be **[(1, 1), (2, 2), (3, 3)]**, and after batch 2, the output would be **[(1, 2), (2, 4), (3, 3), (4, 1)]**.

- **Append Mode**: In append mode, only the new rows appended in the Result Table since the last trigger will be written to the sink. After batch 1, the output would be **[(1, 1), (2, 2), (3, 3)]** (same as complete mode, since all these rows are new). However, after batch 2, the output would just be **[(4, 1)]**, because that's the only completely new row. Append mode wouldn't output the updated counts for 1 and 2 because those aren't new rows, they're existing rows that have been updated.

- **Update Mode**: In update mode, only the rows in the Result Table that were updated since the last trigger will be written to the sink. So after batch 1, the output would be **[(1, 1), (2, 2), (3, 3)]**. After batch 2, the output would be **[(1, 2), (2, 4), (4, 1)]** - it includes the updated counts for 1 and 2, as well as the new row for 4, but not the unchanged count for 3.

So in summary:

- **Complete** mode gives you the entire Result Table every time. It's like taking a complete snapshot of your computation's current state after every batch.

- **Append** mode only gives you completely new rows. It's suitable for when your Result Table is effectively growing over time with new data, such as when you're just adding new rows and not updating existing ones.

- **Update** mode gives you any rows that are new or have been updated. It's a middle ground between complete and append mode, giving you a view of what's changed in your Result Table since the last batch.

If we are calling it as a unbounded table then at which stage it will cause memory issue?

And here's how each output mode would handle these batches:

While Spark Structured Streaming conceptualizes the stream as an "unbounded table", it doesn't literally store all the data of the stream in memory indefinitely. This is because doing so would indeed lead to memory issues over time, especially for long-running streams with a lot of data.

The way Spark Structured Streaming handles this issue depends on the type of operations being performed on the stream:

- **Stateless operations**: For stateless operations (like mapping and filtering), where the processing of each record is independent of the processing of other records, Spark doesn't need to keep any old data around. Once it has processed a piece of data, it can free up the memory that data was occupying.

- **Stateful operations**: For stateful operations (like aggregations and windowed computations), where the processing of some records depends on the processing of other records, Spark uses a combination of techniques to manage memory:

    - **Incremental updates**: For many stateful operations, Spark can update the state incrementally as new data arrives, without needing to keep all past raw data. For example, for a word count, it can just maintain a running count for each word, updating the counts as new data comes in.

    - **Watermarking**: For windowed computations and handling late data, Spark uses a technique called watermarking to limit how much old data it needs to remember. The watermark is a threshold in event-time, and any data older than this threshold is considered "too late" and is dropped.

    - **State expiration**: For arbitrary stateful operations, Spark allows you to specify a time-to-live for the state data. Any state data that has not been updated for this length of time will be cleared.

What if memory is full due to state maintenance after Aggregation function ?

Grow Data Skills

- **Scaling horizontally**: You can add more worker nodes to your Spark cluster, increasing the total memory available for state maintenance.

- **State Expiration**: If your use case allows it, you can define a timeout after which the state for old keys will be removed. This can be done using the mapGroupsWithState operation in Structured Streaming, which allows you to define a timeout for the state.

- **Aggregations**: For certain types of queries, you can make use of built-in aggregations that maintain a small state, like counts, sums, or averages. You can also use approximate data structures (like HyperLogLog for unique count, or t-digest for median) that give you an approximate result but use fixed memory.

- **State Store Compression:** Spark also supports compressing the stored data, which can help to reduce memory footprint. You can enable this by setting the configuration spark.sql.streaming.stateStore.compression.codec to a valid compression codec, such as lz4, lzf, snappy, zstd, or none.

- **Increase memory overcommit**: Overcommit is a scenario where the operating system allows more memory to be allocated to processes than the physical memory actually available. By increasing memory overcommit, you can use more memory for Spark tasks, but be careful as it may cause out-of-memory errors.

- **Checkpointing**: You can enable checkpointing in order to periodically save the state of a stream. However, this does not directly solve the memory issue, but it does provide fault tolerance so that if your job fails due to memory problems or any other reason, it can be restarted from the checkpoint.

- **Optimize for Network Traffic**: If your task generates a large amount of network traffic (which can also consume significant memory), you might consider decreasing the batch interval, or repartitioning your data.

DStreams vs Spark Structured Streaming

Grow Data Skills

Feature	DStreams	Spark Structured Streaming
API Level	Low-level API (built over RDDs)	High-level API (built over DataFrames/Datasets)
Processing Type	Micro-Batch Processing	Both Continuous and Micro-Batch Processing
Operation Style	Requires understanding of Transformations and Actions	SQL-like operations
Late Data Handling	Not as efficient	Handles gracefully with watermarking
Processing Model	Each micro-batch results in an RDD and processed independently, resulting in higher latency	Efficient model leads to lower end-to-end latency
Data Sources and Sinks	Supports fewer sources and sinks	Supports a wider variety of sources and sinks

File Source - Spark Structured Streaming

Grow Data Skills

File1.json

```
{"userId": "user1", "timestamp": "2023-07-23T00:00:00Z"}
{"userId": "user2", "timestamp": "2023-07-23T00:01:00Z"}
```

File2.json

```
{"userId": "user1", "timestamp": "2023-07-23T00:02:00Z"}
{"userId": "user3", "timestamp": "2023-07-23T00:03:00Z"}
```

We can use Structured Streaming in Spark to process files as they arrive in a directory. We'll use the **readStream** method to **monitor** a directory for new files. It does not process the files that were present in the directory before the streaming computation started.

This is different from batch processing, where you might use the **read** method to process all files present in a directory at once.

Keep in mind that **"new files"** means files that are newly added to the directory. Files that are already present when the streaming computation starts, or files that are simply modified after the streaming computation has started, will not be processed. It is recommended to move or copy atomic files into the directory to avoid Spark picking up incomplete files.

We can configure **spark.sql.streaming.schemaInference** property to get the source schema during run time itself.

File Source - Spark Structured Streaming

Grow Data Skills

```
from pyspark.sql import SparkSession

# Create a SparkSession with schema inference enabled
spark = SparkSession \
    .builder \
    .appName("StructuredStreamingUserLogCount") \
    .config("spark.sql.streaming.schemaInference", "true") \
    .getOrCreate()

# Read JSON files from directory
logs = spark \
    .readStream \
    .format("json") \
    .option("path", "/path/to/your/directory") \
    .load()

# Generate running count of logs for each user
userLogCounts = logs.groupBy("userId").count()

# Start running the query that prints the running counts to the console
query = userLogCounts \
    .writeStream \
    .format("json") \
    .outputMode("complete") \
    .option("path","output_dir") \
    .queryName("Logging Count") \
    .start()

query.awaitTermination()
```

Triggers - Spark Structured Streaming

In Spark Structured Streaming, a trigger determines when the system should check for new data and process it. By default, it's set to process the data as soon as the system has completed processing the last batch of data, but you can set it to a fixed time interval or to only process one batch of data and then stop.

Here are three types of triggers:

- **Default Trigger (Processing Time):** If no trigger setting is explicitly specified, the system will check for new data as soon as it finishes processing the last batch. This is called "micro-batch" processing. If the system is idle (i.e., there is no new data to process), it will just wait until new data arrives.

- **Fixed Interval Micro-Batches**: The system will check for new data at a fixed interval, regardless of whether the system has finished processing the last batch of data.

- **One-Time Micro-Batch:** The system will process one batch of data and then stop. This is useful for testing and debugging.