International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 09 Issue: 01 | Jan - 2025 SJIF Rating: 8.448 ISSN: 2582-3930

Data Streaming with Apache Spark

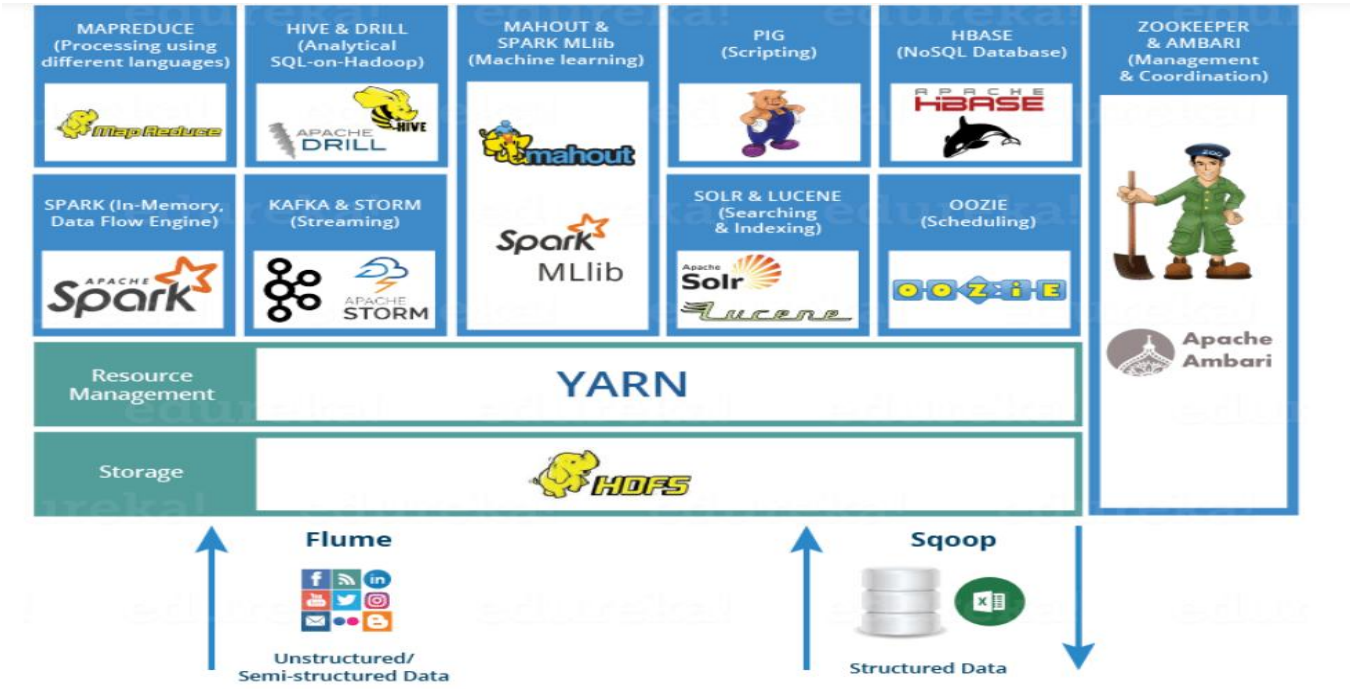
Author
Ranjith Martha

Abstract
In today's fast-paced world, real-time data streaming is crucial for many businesses and industries. With tools like Apache Spark and Hadoop, organizations can process vast amounts of data as it's created, gaining insights in seconds rather than hours. This article provides a comprehensive look at real-time streaming with Spark and Hadoop, including use cases, architecture, and step-by-step integration.

1. Introduction
In the era of big data, real-time data processing has become critical for businesses to gain immediate insights and make timely decisions. Real-time streaming applications enable organizations to process and analyze data as it arrives, facilitating applications such as fraud detection, network monitoring, and real-time recommendations. This guide demonstrates how to build a real-time data streaming application using Hadoop HDFS and Apache Spark for Data streaming.

2. Hadoop Ecosystem Overview
The Hadoop ecosystem [1] is a collection of open-source tools and frameworks designed for scalable storage, processing, and analysis of large datasets. It is centered around Hadoop Distributed File System (HDFS) and MapReduce for data storage and processing, and is highly extensible, allowing integration with other components to manage, access, process, and analyze big data.

Figure 1: Hadoop Eco system



3. Key Components of the Hadoop Ecosystem

The below components work together in the Hadoop ecosystem to create a powerful and flexible environment for handling big data.

1. HDFS (Hadoop Distributed File System)

A distributed storage system that breaks down large files into blocks and stores them across a cluster. Provides fault tolerance by replicating blocks across multiple nodes.

2. YARN (Yet Another Resource Negotiator)

Hadoop's resource manager that allocates cluster resources, manages node health, and handles job scheduling and execution. Enables multiple data-processing engines (such as Spark and MapReduce) to run on the same cluster.

3. MapReduce

A programming model for processing large datasets in parallel, based on dividing tasks into Map (processing individual records) and Reduce (aggregating results) phases.While robust, MapReduce can be slower and less flexible compared to modern engines like Spark.

4. Hive

A data warehousing and SQL-like query engine on top of Hadoop that provides a high-level interface for querying and managing large datasets. Allows SQL-like queries to run on HDFS data using HiveQL, which translates into MapReduce or Spark jobs.

5. HBase

A NoSQL database for real-time, random read/write access to large datasets. Typically used for low-latency applications where quick access to specific data points is essential.

6. Pig

A high-level scripting language for data analysis that runs on Hadoop, using its own syntax (Pig Latin) to translate complex tasks into MapReduce jobs. Known for its ease of use for ETL (extract, transform, load) workflows.

7. Spark

A high-speed, distributed processing engine that can work with Hadoop data in HDFS. Spark provides an alternative to MapReduce for more complex, memory-intensive, iterative tasks.

8. Zoo Keeper

A coordination and synchronization service for distributed applications, often used to manage configurations and metadata across Hadoop components.

9. Oozie

A workflow scheduler for managing and scheduling Hadoop jobs (e.g., MapReduce, Pig, Hive) in a sequence, allowing the automation of complex data processing pipelines.

10. Flume and Sqoop

Flume is a service for collecting, aggregating, and transporting large volumes of log data into Hadoop and Sqoop is a tool for transferring structured data between Hadoop and relational databases.

4. Apache Spark Overview

Apache Spark[2] is an open-source tool. It is a newer project, initially developed in 2012, at the AMPLab at UC Berkeley. It is focused on processing data in parallel across a cluster, but the biggest difference is that it works in memory.

Apache Spark is a unified analytics engine for large-scale data processing, known for its speed, ease of use, and sophisticated analytics. Spark is designed to work with the Hadoop ecosystem but can also stand alone. Unlike traditional Hadoop MapReduce, Spark provides a more versatile architecture that allows for in-memory processing, iterative algorithms, and real-time streaming.

It is designed to use RAM for caching and processing the data. Spark performs different types of big data workloads like:

- Batch processing.
- Real-time stream processing.
- Machine learning.
- Graph computation.

- Interactive queries.

5. Core Components of Spark

There are mainly 5 core components[3] of Apache Spark:

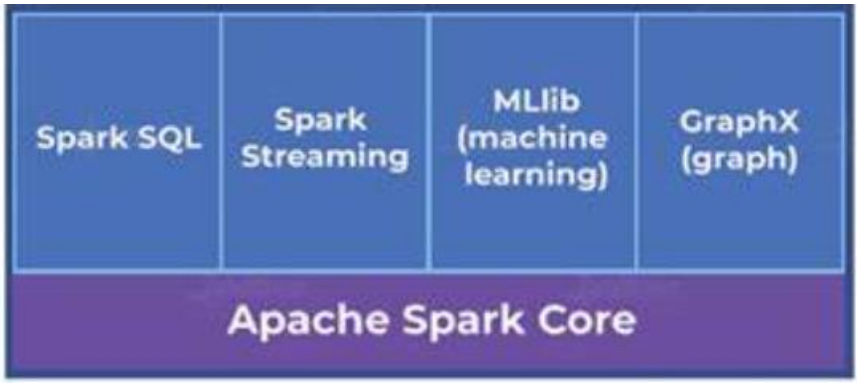


Figure 2: Spark Core components

1. Apache Spark Core:

The foundation of the Spark platform, responsible for task scheduling, memory management, fault recovery, and other basic functions. Provides an API for resilient distributed datasets (RDDs), the main abstraction for working with distributed data.

2. Spark SQL

A module for working with structured and semi-structured data using DataFrames, Datasets, and SQL queries. Provides optimizations through Catalyst (Spark's query optimizer) and integrates well with Hive, JDBC, and other SQL-compatible tools.

3. Spark Streaming

Enables real-time data processing of streaming data from sources like Apache Kafka, Flume, or HDFS. Processes data in small micro-batches to handle continuous data streams.

4. MLlib (Machine Learning Library)

A scalable machine learning library offering a variety of algorithms for classification, regression, clustering, collaborative filtering, etc. Provides tools for feature extraction, transformation, and selection.

5. GraphX

A library for processing and analyzing large-scale graphs, providing a framework for graph analytics and graph-parallel computations. Integrates graph processing with Spark's overall data workflow.

6. Spark Streaming Architecture

Spark Streaming [4]is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window. Finally, processed data can be pushed out to file systems, databases, and live dashboards. In fact, you can apply Spark's machine learning and graph processing algorithms on data streams.

Micro-Batching: Spark Streaming collects data from Kafka or another source and processes it in small batches. Each batch represents a small "window" of real-time data.

DStream (Discretized Stream): Spark Streaming processes data through a DStream, a continuous series of RDDs (Resilient Distributed Datasets) that Spark processes sequentially.

Check pointing: For fault tolerance, Spark supports checkpointing, where data and state information

are stored to recover in case of failures.

Figure 3: Spark Streaming Hadoop Eco system

Figure 4: Spark Streaming batch process

Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

7. Key Components for Real-Time Streaming

The architecture for a real-time data streaming application typically involves the following components and depends on the use cases. As described in section 6 Apache kafka is used for real time streaming in combination with Spark Streaming. To demonstrate near real time data streaming Hadoop HDFS is integrated with Spark Streaming for implementation as part of this article.

Apache Kafka: A distributed messaging system that acts as a buffer for incoming data streams, allowing for decoupled data producers and consumers.

Hadoop HDFS: A distributed file system that provides high-throughput access to application data and serves as the storage layer for processed data.

Apache Spark: A fast, in-memory data processing engine that can handle real-time data processing through Spark Streaming.

8. Methods - Implementation of Spark Structured Streaming

As discussed in section 6 (Spark Streaming Architecture), there are multiple ways of streaming data from sources and targets. Based on the need and use cases various utilities can be used in the spark streaming [4].

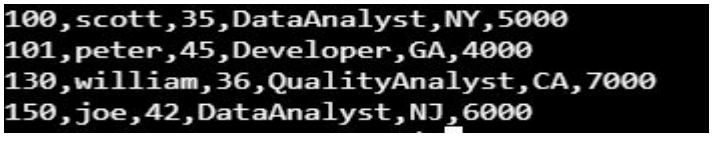
Below are some of the utilities/softwares that are required to archive this.

Apache Spark: Download and install from Apache Spark.
Hadoop: Download and install from Apache Hadoop.

Data Streaming can be performed using python, scala or java programming languages. Examples for implementing streaming is provided at [6] and the newer streaming engine in Spark called Structured Streaming [7] for streaming applications and pipelines.

Sample customer csv files are saved in Hadoop file system with 2 files and 2 records each in customer directory.

Figure 5: Hadoop source data



Python/pyspark code is used for this article as part of structured streaming use case.

This code captures the data from Hadoop files system stored in csv format and writes the data to output console after applying transformation in spark.

Code:

```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark import SparkConf, SparkContext

app_name = 'readfromcsv'
num_cores= '*'
configuration = (SparkConf()
                    .set("spark.master", "local[" + str(num_cores) + "]")
                    .set("spark.dynamicAllocation.enabled", "true"))

spark = (SparkSession
                  .builder
                  .appName(app_name)
                  .enableHiveSupport()
                  .config(conf=configuration)
                  .getOrCreate())

cust_schema=StructType([StructField('id',IntegerType(),True),
                    StructField('name',StringType(),True),
                    StructField('age',IntegerType(),True),
                    StructField('profession',StringType(),True),
                    StructField('city',StringType(),True),
                    StructField('salary',DoubleType(),True)])

#input stream - reading data
customer_stream=spark.readStream.format("csv").schema(cust_schema)  \
               .option("header",False)  \
               .option("maxFilesPerTrigger",1)  \
               .load("customer")

avg_sal=customer_stream.groupBy("profession") \
.agg((avg("salary").alias("average_salary")), \
(count("profession").alias("count"))) \
.sort(desc("average_salary"))

#writing data on console
query=  avg_sal.writeStream.format("console").outputMode("complete").start()

print("Started Streaming Data..")
```

Figure 6: Spark Streaming code

Results: Count of employees for each profession and respective average salaries of profession in descending order are captured on console as part of streaming data in batches using Apache Spark.

```
In [3]: Started Streaming Data..
    ...: -------------------------------------------
    ...: Batch: 0
    ...: -------------------------------------------
    ...: +-----------+--------------+-----+
    ...: | profession|average_salary|count|
    ...: +-----------+--------------+-----+
    ...: |DataAnalyst|        5000.0|    1|
    ...: |  Developer|        4000.0|    1|
    ...: +-----------+--------------+-----+
    ...:
    ...: -------------------------------------------
    ...: Batch: 1
    ...: -------------------------------------------
    ...: +--------------+--------------+-----+
    ...: |    profession|average_salary|count|
    ...: +--------------+--------------+-----+
    ...: |QualityAnalyst|        7000.0|    1|
    ...: |   DataAnalyst|        5500.0|    2|
    ...: |     Developer|        4000.0|    1|
    ...: +--------------+--------------+-----+
```

Figure 7: Spark Streaming results

Discussion

Data is loaded into Streaming Data frame using readStream method. The maxFilesPerTrigger option in Spark Structured Streaming is used to control the number of files that Spark will process in each micro-batch when reading data from a file-based source (such as a directory). Format() is used for streaming data to a destination in our case it is console.

OutputMode() method is used for data to be written to a sink.

There are three options for outputMode() method:

1) append: Only new rows will be written to the sink.
2) complete: All rows will be written to the sink, every time there are updates.
3) update: Only the updated rows will be written to the sink, every time there are updates.

Streaming is kicked off using the start() method in below code.

9. Use Cases for Real-Time Data Streaming with Spark and Hadoop

Various businesses and organization can use the Real time Data streaming to identify fraud detection/user activity and monitoring any health patterns for medical treatments. Capture any outages or latency issues.

10. Advantages of Spark and Hadoop for Real-Time Data Streaming

Scalability: Hadoop's distributed architecture and Spark's in-memory processing make it suitable for processing terabytes of data in real time.

Cost-Effective: Open-source and based on commodity hardware, Spark and Hadoop are cost-efficient compared to proprietary solutions.

Reliability and Fault Tolerance: Both Spark and Hadoop are built with fault-tolerant mechanisms, ensuring that streaming data pipelines are reliable.

11. Key Considerations and Best Practices

When working with real-time data processing using Apache Spark Streaming, several key considerations and best practices [5] can enhance performance, reliability, and maintainability. Below is a comprehensive list of these practices:

Data Partitioning and Parallelism: Ensure data is well-partitioned to take advantage of Spark's distributed architecture for faster processing.

Latency Management: Tune micro-batch intervals and system parameters to balance latency and throughput.

Data Serialization: Choose Efficient Formats: Use efficient serialization formats like Avro, Parquet, or ORC to optimize storage and processing speed. These formats support schema evolution and provide better compression and query performance and Avoid Python Serialization Overhead by minimizing the use of Python UDFs (User Defined Functions) and use built-in Spark functions whenever possible.

Resource Utilization: Executor Configuration: Tune the number of cores per executor (spark.executor.cores) and the number of executors to match the cluster resources and workload characteristics. Avoid oversubscription of cores to prevent context switching overhead.

Error Handling and Monitoring: Set up robust logging and monitoring to identify issues in real-time data streams quickly.

Conclusion

Apache Spark streaming is a powerful solution for real-time data streaming and processing. This enables organizations to capture and analyze data instantly, leading to more timely insights and decisions. With the increasing demand for real-time data applications, these technologies will continue to play a pivotal role in the big data landscape. As businesses increasingly adopt IoT, mobile applications, and connected devices, the importance of real-time data streaming will grow.

References

[1] Hadoop Eco System
https://www.edureka.co/blog/hadoop-ecosystem

[2] Apache spark
https://spark.apache.org/

[3] Spark Core Components
https://intellipaat.com/blog/tutorial/spark-tutorial/spark-sql/

[4] .Spark Streaming
https://spark.apache.org/docs/latest/streaming-programming-guide.html

[5] Efficient Data Processing with Apache Spark
https://doi.org/10.36227/techrxiv.173194439.98776144/v1

[6] Spark Streaming Example
https://docs.cloudera.com/runtime/7.2.18/developing-spark-applications/topics/spark-streaming-example.html

[7] Structured Streaming guide
https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

ABOUT THE AUTHOR

Ranjith Martha is an accomplished IT professional specialized in Data Engineering and Cloud Technologies. With expertise in Big Data, Hadoop, and various cloud platforms. Solution Designer of Data Warehousing and Data Analytical Solutions. Certified in Cloud, HCTA and PCAP Python Programming Technologies.