# Apache Spark vs Flink, a detailed comparison

## Spark vs. Flink

Apache Spark and Apache Flink are two of the most popular data processing frameworks. Both enable distributed data processing at scale and offer improvements over frameworks from earlier generations. Flink is newer and includes features Spark doesn't, but the critical differences are more nuanced than old vs. new. We'll take an in-depth look at the differences between Spark vs. Flink. Let's start with some historical context.

### Stream and batch processing

Modern data processing frameworks rely on an infrastructure that scales horizontally using commodity hardware. In this category, there are two well-known parallel processing paradigms: batch processing and stream processing. Batch processing refers to performing computations on a fixed amount of data. This means that we already know the boundaries of the data and can view all the data before processing it, e.g., all the sales that happened in a week.

Stream processing is for "infinite" or unbounded data sets that are processed in real-time. Common use cases for stream processing include monitoring user activity, processing gameplay logs, and detecting fraudulent transactions.

## The evolution of distributed data processing

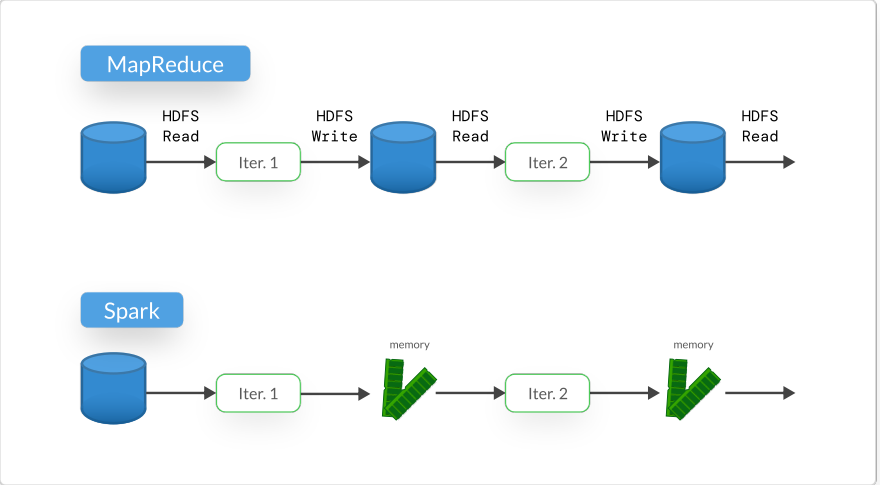
Spark and Flink are third and fourth-generation data processing frameworks. To understand how the industry has evolved, let's review each generation to date.

MapReduce (https://en.wikipedia.org/wiki/MapReduce) represents the first generation of distributed data processing systems. This framework processes parallelizable  data and computation on a distributed infrastructure that scales

horizontally. MapReduce also abstracts all the system-level complexities of the distributed system from developers and provides fault tolerance, in addition to parallelization and data distribution. It supports batch processing.

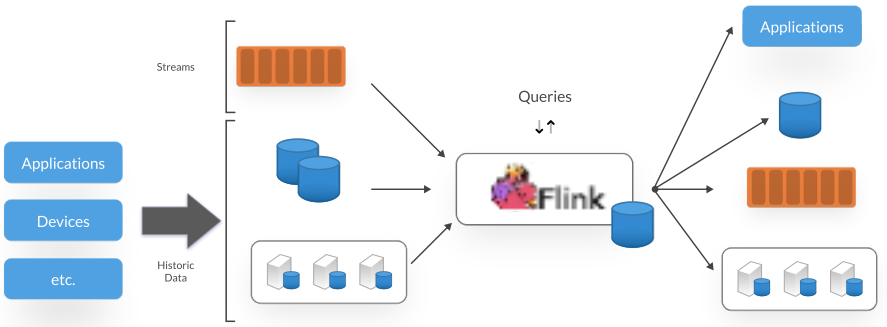
Some second-generation frameworks of distributed processing systems offered improvements to the MapReduce model. For example, Tez (https://tez.apache.org/) provided interactive programming *and* batch processing.

Spark is considered a third-generation data processing framework, and it  natively supports batch processing and stream processing. Spark leverages micro batching for streaming which provides near real-time processing. Micro batching divides the unbounded stream of events into small chunks (batches) and triggers the computations. Spark enhanced the performance of MapReduce by doing all the processing in memory instead of making each step write the results back to the disk.




Traditional MapReduce writes to disk, but Spark can process in-memory. Source (https://id2221kth.github.io/slides/2019/06_parallel_processing_part2.pdf).

Flink is a fourth-generation data processing framework andis one of the top Apache projects. Flink supports both batch and stream processing and is designed for stream processing natively. It promotes continuous streaming where event computations are triggered as soon as the event is received.




A high-level view of the Flink ecosystem. Source. (https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/learn-flink/overview/)

Macrometa

Stateful geo-replicated stream processing keeps globally distributed data consistent

One integrated platform for streams, key values, docs, graphs, and search simplifies development

Declarative configuration using JavaScript and SQL avoids the need to learn a new syntax

## Spark vs. Flink: feature comparison

The feature set in Spark and Flink differs in many ways. The table below summarizes the feature sets, and in the next section, we'll take a detailed look at each framework.

| Data Processing | Batch/Stream (micro Batch) | Batch/Stream (Native) |
| --- | --- | --- |

| | | |
|---|---|---|
| Iterations | No | Yes |
| SQL | Spark SQL | Table & SQL API |
| Fault tolerance | Yes (WAL) | Yes (Chandy-Lamport) |
| Optimization | Manual | auto |

## Spark vs. Flink: an in-depth look

A table of features only shares part of the story. While Spark and Flink have similarities and advantages, we'll review the core concepts behind each project and pros and cons.

### Streaming engine

Spark offers a scalable fault tolerant data processing engine that supports both batch and stream processing. Spark's consolidation of disparate system capabilities (batch and stream) is one of the key reasons for its popularity. Although it provides a single framework to satisfy all processing needs, it isn't the best solution for all use cases.

Flink offers true native streaming, while Spark uses micro batches to emulate streaming. That means Flink processes each event in real-time and provides very low latency. Spark, by using micro-batching, can only deliver *near* real-time processing. For many use cases, Spark provides acceptable performance levels.

Flink's low latency outperforms Spark consistently, even at higher throughput. Spark can achieve low latency with lower throughput, but increasing the throughput will also increase the latency. This tradeoff means that Spark users need to tune the configuration to reach acceptable performance, which can also increase the development complexity.

### Iterative processing

Data processing systems don't usually support iterative processing, an essential feature for most machine learning and graph algorithm use cases. To accommodate these use cases, Flink provides two iterative operations – iterate and

delta iterate. Spark, however, doesn't support any iterative processing operations. These operations must be implemented by application developers, usually by using a regular loop statement.

However, Spark does provide a cache operation, which lets applications explicitly cache a dataset and access it from the memory while doing iterative computations. Since Spark iterates over data in batches with an external loop, it has to schedule and execute each iteration, which can compromise performance. Flink instead uses the native loop operators that make machine learning and graph processing algorithms perform much better than Spark.

## SQL

SQL support exists in both frameworks to make it easier for non-programmers to leverage data processing needs. Spark SQL lets users run queries and is very mature. It also provides a Hive-like query language and APIs for querying structured data. Similarly, Flink's SQL support has improved. It started with support for the Table API (https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/table/overview/) and now includes Flink SQL support as well.

## Fault tolerance

Both systems are distributed and designed with fault tolerance in mind. Spark can recover from failure without any additional code or manual configuration from application developers. Spark recently introduced write-ahead logs (https://www.waitingforcode.com/apache-spark-streaming/spark-streaming-checkpointing-and-write-ahead-logs/read)(WAL) that store all the data received to the logs files in the checkpoint folder. Data is always written to WAL first so that Spark will recover it even if it crashes before processing. Since Spark has RDDs (Resilient Distributed Dataset) as the abstraction, it recomputes the partitions on the failed nodes transparent to the end-users.

Flink is a fault tolerance processing engine that uses a variant of the Chandy-Lamport algorithm to capture the distributed snapshot. This algorithm is lightweight and non-blocking, so it allows the system to have higher throughput and strong consistency guarantees. It checkpoints the data source, sink, and application state (both windows state and user-defined state) in regular intervals, which are used for

failure recovery. Flink can run a considerable number of jobs for months and stay resilient, and it also provides configuration for end developers to set it up to respond to different types of losses.

## Optimization

Programs (jobs) created by developers that don't fully leverage the underlying framework should be further optimized.

Spark jobs need to be optimized manually by developers. It has an extensible optimizer, Catalyst (https://databricks.com/glossary/catalyst-optimizer), based on Scala's functional programming construct. Spark simplifies the creation of new optimizations and enables developers to extend the Catalyst optimizer.

Flink provides an optimizer that optimizes jobs before execution on the streaming engine. The Flink optimizer is independent of the programming interface and works similarly to relational database optimizers by transparently applying optimizations to dataflows.

## Windowing

Streaming refers to processing an "infinite" amount of data, so developers never have a global view of the complete dataset at any point in time. Hence, we must divide the data into smaller chunks, referred to as windows (https://www.macrometa.com/event-stream-processing/stream-processing), and process it. Generally, this division is time-based (lasting 30 seconds or 1 hour) or count-based (number of events).

Both Flink and Spark provide different windowing strategies that accommodate different use cases. Spark offers basic windowing strategies, while Flink offers a wide range of techniques for windowing.

Most of Flink's windowing operations are used with keyed streams only. A keyed stream is a division of the stream into multiple streams based on a key given by the user. This allows Flink to run these streams  in parallel on the underlying distributed

infrastructure. Spark has sliding windows but can also emulate tumbling windows with the same window and slide duration. However, Spark lacks windowing for anything other than time since its implementation is time-based.

Flink supports tumbling windows, sliding windows, session windows, and global windows out of the box. Furthermore, users can define their custom windowing as well by extending WindowAssigner. Flink windows have start and end times to determine the duration of the window. Flink manages all the built-in window states implicitly.

### State management

Suppose the application does the record processing independently from each other. In that case, there is no need to store the state. This scenario is known as stateless data processing. An example of this is recording data from a temperature sensor to identify the risk of a fire. However, most modern applications are stateful and require remembering previous events, data, or user interactions.

There are usually two types of state that need to be stored, application state and processing engine operational states. Application state is the intermediate processing results on data stored for future processing. Operation state maintains metadata that tracks the amount of data processing and other details for fault tolerance purposes. When we say the state, it refers to the application state used to maintain the intermediate results.

Spark only supports HDFS-based state management. Incremental checkpointing, which is decoupling from the executor, is a new feature. On the other hand, Spark still shares the memory with the executor for the in-memory state store, which can lead to OutOfMemory issues. Also, the same thread is responsible for taking state snapshots and purging the state data, which can lead to significant processing delays if the state grows beyond a few gigabytes.

In comparison, state is a first-class citizen in Flink and is frequently checkpointed based on the configurable duration. These checkpoints can be stored in different locations, so no data is lost if a machine crashes. Flink supports in-memory, file system, and RocksDB as state backend. If a process crashes, Flink will read the

state values and start it again from the left if the data sources support replay (e.g., as with Kafka (https://www.macrometa.com/event-stream-processing/kafka-alternatives) and Kinesis).

## Language support

Choosing the correct programming language is a big decision when choosing a new platform and depends on many factors. Teams will need to consider prior experience and expertise, compatibility with the existing tech stack, ease of integration with projects and infrastructure, and how easy it is to get it up and running, to name a few. Luckily, Spark and Flink support major languages - Java, Scala, Python.

Spark is written in Scala and has Java support. Both languages have their pros and cons. For example, Java is verbose and sometimes requires several lines of code for a simple operation. Also, Java doesn't support interactive mode for incremental development. Scala, on the other hand, is easier to maintain since it's a statically-typed language, rather than a dynamically-typed language like Python. Spark supports R, .NET CLR (C#/F#), as well as Python.

Flink is natively-written in both Java and Scala. With Flink, developers can create applications using Java, Scala, Python, and SQL. These programs are automatically compiled and optimized by the Flink runtime into dataflow programs for execution on the Flink cluster. Although Flink's Python API, PyFlink, was introduced in version 1.9, the community has added other features. PyFlink has a simple architecture since it does provide an additional layer of Python API (https://www.alibabacloud.com/blog/pyflink-architecture-and-applicable-business-scenarios_597663) instead of implementing a separate Python engine.

## Cost

Both Spark and Flink are open source projects and relatively easy to set up. However, since these systems do most of the executions in memory, they require a lot of RAM, and an increase in RAM will cause a gradual rise in the cost. Spark has a couple of cloud offerings to start development with a few clicks, but Flink doesn't have any so far. This means that Flink can be more time-consuming to set up and run.