Apache Flink – A Big Data Processing Platform

The Big Data came into limelight when **Google published a paper** (long ago in 2004) about MapReduce a programming paradigm it is using to handle the huge volume of contents for indexing the web(Learn more from here about History of big data).

It has been accepted by each and every domain (Telecom, Retail, Finance, Healthcare, Banking etc.) that Big Data is a must to handle the growing data and analytics requirements. Overall we can say Big Data is a must for each and every business to survive or grow.

After Hadoop we keep getting tons of technologies to handle Big Data, like MapReduce – Batch processing engine, Apache Storm – Stream processing engine, Apache Tez – Batch and interactive engine, Apache Giraph – Graph processing engine, Apache Hive – SQL engine.

Every framework is a specialized engine which solves some specific problems. But to solve real-world problems we need to combine multiple frameworks. Integration of multiple frameworks is powerful but making them work on the same platform is non-trivial, costly and complex.

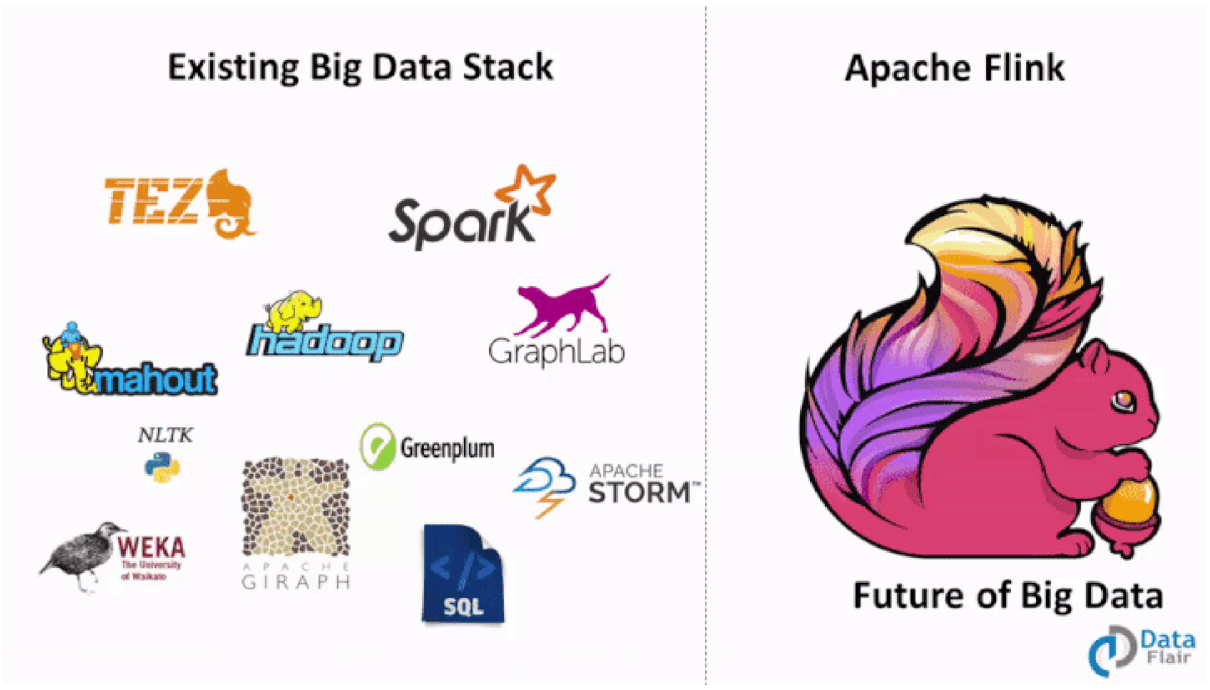
What is the solution??

The industry needs a generalized platform which alone can handle diverse workloads like:

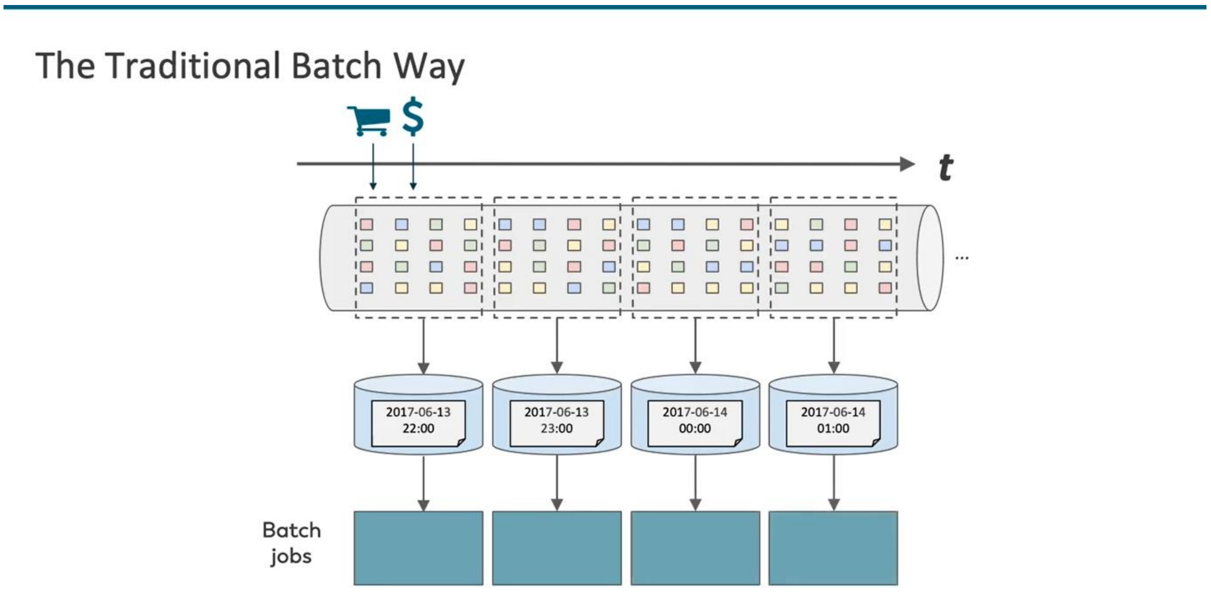
- Batch Processing
- Interactive Processing
- Real-time (stream) Processing
- Graph Processing
- Iterative processing
- In-memory processing

Thus. the platform should also provide distributed computing, fault tolerance, high availability, ease of use and lightning fast speed.

Apache Flink – The Unified Platform



Apache Spark has started the new trend by offering a diverse platform to solve different problems but is limited due to its underlying batch processing engine



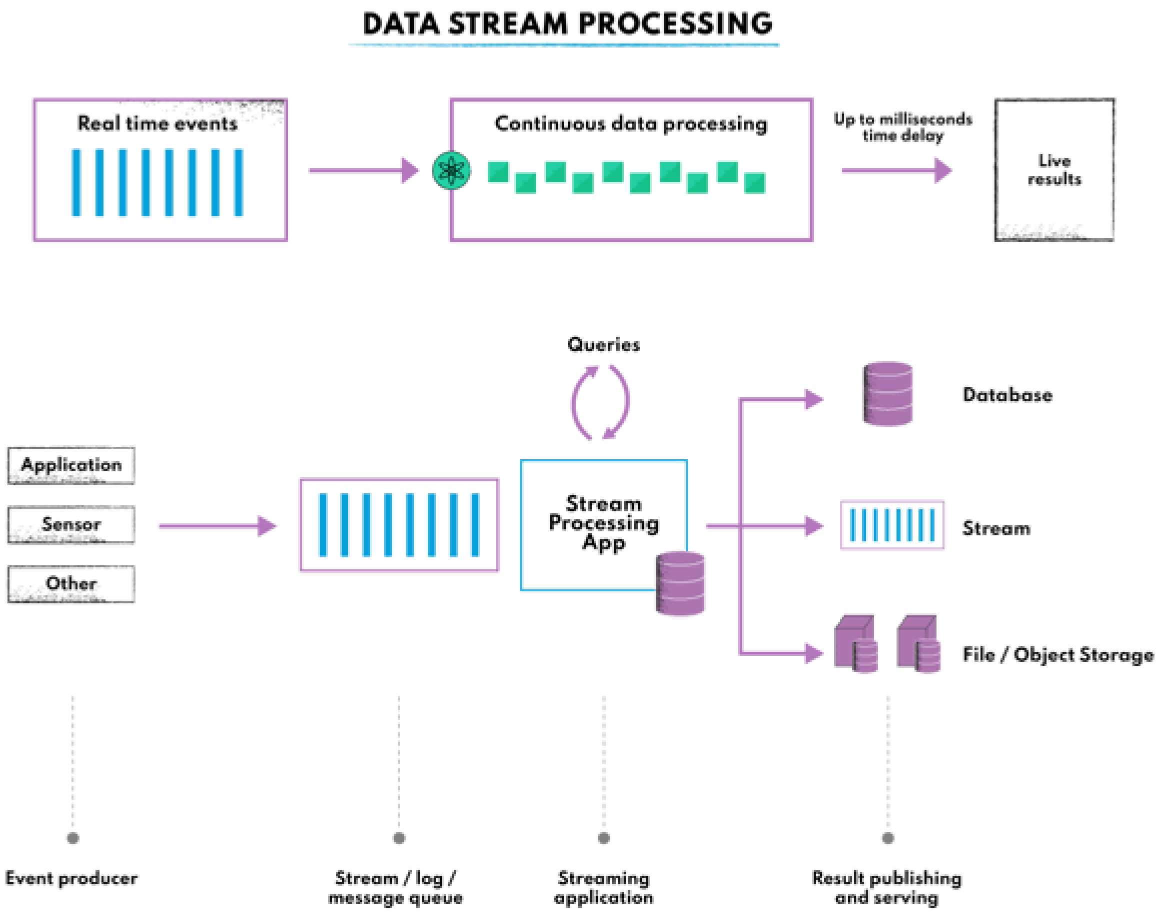
Processing based on the data collected over time is called Batch Processing. For example, a bank manager wants to process past one-month data (collected over time) to know the number of cheques that got cancelled in the past 1 month.

Processing based on immediate data for instant result is called Real-time Processing. For example, a bank manager getting a fraud alert immediately after a fraud transaction (instant result) has occurred.

What is Stream Processing?

Before we get into Apache Flink, it's essential to understand stream processing. Stream processing is a type of data processing that deals with **continuous, real-time data streams.**

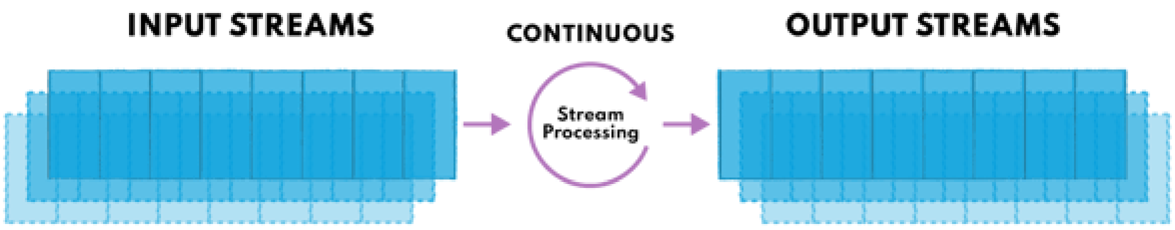
Batch processing can be thought of as dealing with "data at rest," while stream processing deals with "data in motion."



stream processing has several benefits over batch processing:

- **Lower latency**: Since stream processors deal with data in **near-real-time**, the overall latency is lower and offers the opportunity for multiple specific use cases that need in-motion checks.
- **Flexibility:** Stream process transaction data is generally more flexible than batch, as a wider variety of end applications, data types, and formats can easily be handled. It can also accommodate changes to the data sources (e.g., adding a new sensor to an IoT application).

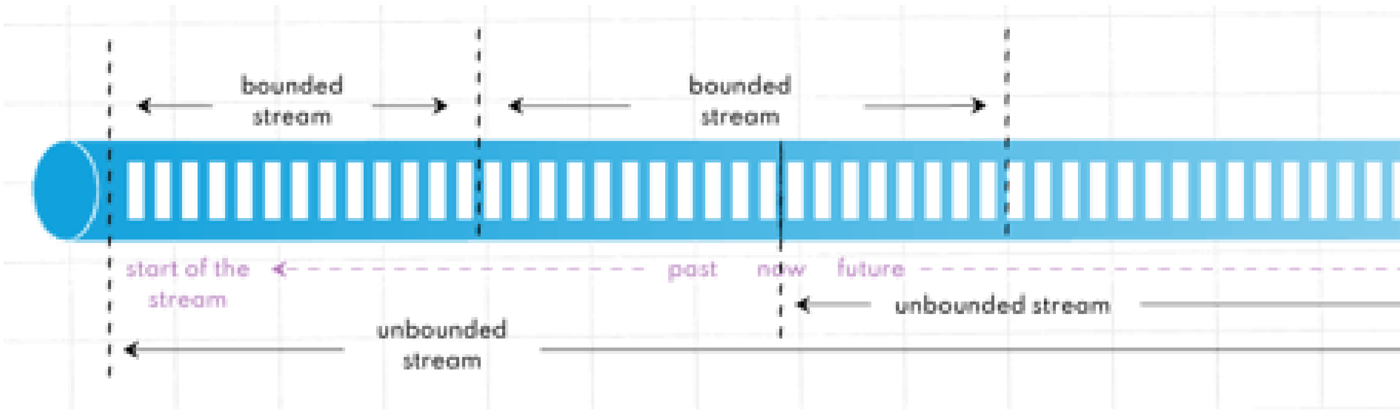
- **Less expensive**: Since stream processors can handle a continuous data flow, the overall cost is lower (lack of a need to store data before processing it).



Apache Flink is a general purpose cluster computing tool, which can handle batch processing, interactive processing, Stream processing, Iterative processing, in-memory processing, graph processing.

Process Unbounded and Bounded Data Streams

Apache Flink allows for both bounded and unbounded data stream processing. Bounded data streams are finite, while unbounded streams are infinite.



Bounded and unbounded streams

Bounded Data Streams

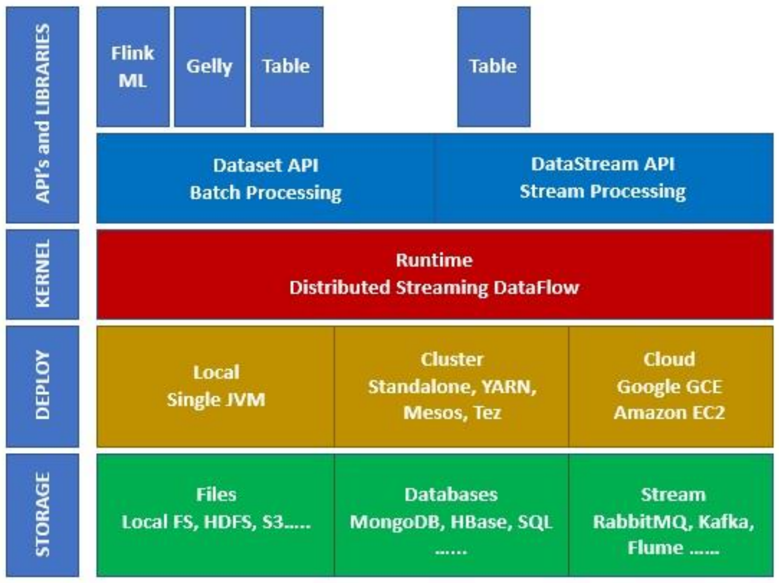
Bounded data streams have a **defined beginning and end**; they can be processed in one batch job or multiple parallel jobs. Apache Flink's **DataSet API** is used to process bounded data sets, consisting of individual elements over which the user iterates. This type of system is often used for batch-like processing of data that is already present and known ahead of time - such as a customer database or log files.

Unbounded Streams

Unbounded data streams, on the other hand, have no start or end point; they continuously receive new elements that need to be processed right away. This type of processing requires a system that is always running and ready to accept incoming elements as soon as they arrive. To accomplish this, Apache Flink offers a **DataStream API** for real-time processing of the streaming data, allowing users to write applications that process unbounded streams of data.

Ecosystem on Apache Flink

The diagram given below shows the different layers of Apache Flink Ecosystem –


API's and LIBRARIES
Flink ML
Gelly
Table
Table
Dataset API Batch Processing
DataStream API Stream Processing
KERNEL
Runtime Distributed Streaming DataFlow
DEPLOY
Local Single JVM
Cluster Standalone, YARN, Mesos, Tez
Cloud Google GCE Amazon EC2
STORAGE
Files Local FS, HDFS, S3.....
Databases MongoDB, HBase, SQL ......
Stream RabbitMQ, Kafka, Flume ......

Storage

Apache Flink has multiple options from where it can Read/Write data. Below is a basic storage list –

- HDFS (Hadoop Distributed File System)
- Local File System
- S3
- RDBMS (MySQL, Oracle, MS SQL etc.)
- MongoDB
- HBase
- Apache Kafka
- Apache Flume

Deploy

You can deploy Apache Fink in local mode, cluster mode or on cloud. Cluster mode can be standalone, YARN, MESOS.

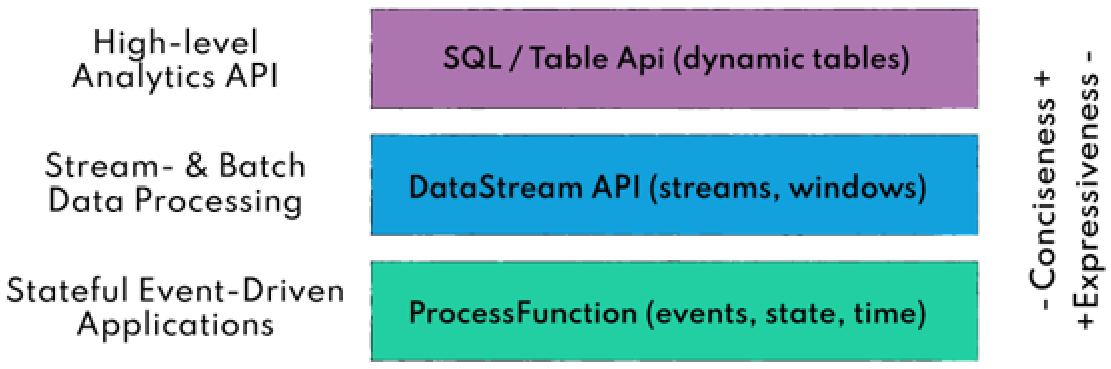
On cloud, Flink can be deployed on AWS or GCP.

Kernel

This is the runtime layer, which provides distributed processing, fault tolerance, reliability, native iterative processing capability and more.

APIs & Libraries

This is the top layer and most important layer of Apache Flink. It has Dataset API, which takes care of batch processing, and Datastream API, which takes care of stream processing. There are other libraries like Flink ML (for machine learning), Gelly (for graph processing), Tables for SQL. This layer provides diverse capabilities to Apache Flink.


High-level Analytics API
SQL / Table Api (dynamic tables)
Stream- & Batch Data Processing
DataStream API (streams, windows)
Stateful Event-Driven Applications
ProcessFunction (events, state, time)
- Conciseness +
+Expressiveness -

Complex Event Processing (CEP)

Flink’s Complex Event Processing library allows users to specify patterns of events using a regular expression or state machine.

SQL & Table API

The Flink ecosystem also includes APIs for relational queries - SQL and Table APIs.

Gelly

Gelly is a versatile graph processing and analysis library that runs on top of the DataSet API

FlinkML

FlinkML is a library of distributed machine learning algorithms that run on top of the DataSet API.

Key Use Cases for Flink

Apache Flink is a powerful tool for handling big data and streaming applications. It supports both bounded and unbounded data streams, making it an ideal platform for a variety of use cases, such as:

- **Event-driven applications:**

fraud detection, anomaly detection, rule-based alerting, business process monitoring, financial and credit card transactions systems, social networks, and other message-driven systems

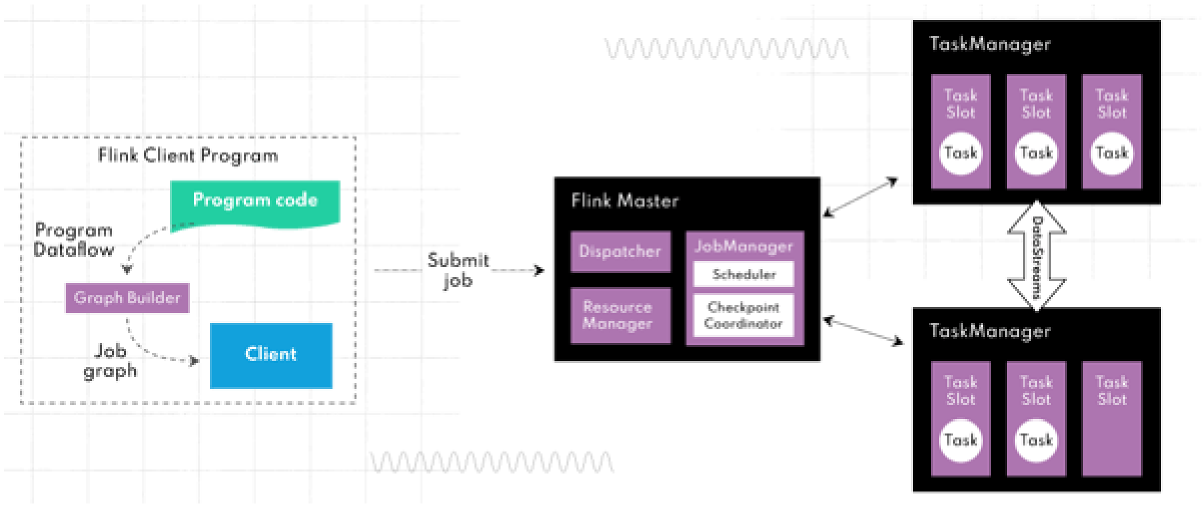
Continuous data pipelines:

Real-time data analytics:

ad-hoc analysis of live data in various industries, customer experience monitoring, large-scale graph analysis, and network intrusion detection.

master/slave architecture

Flink uses a master/slave architecture with **JobManager** and **TaskManagers**. The Job Manager is responsible for scheduling and managing the jobs submitted to Flink and orchestrating the execution plan by allocating resources for tasks. The Task Managers are accountable for executing user-defined functions on allocated resources across multiple nodes in a cluster.



WORD COUNT EXAMPLE

all core classes of the Java DataStream API can be found in org.apache.flink.streaming.api .

the Flink cluster manager will execute your main method and `getExecutionEnvironment()` will return an execution environment for executing your program on a cluster.

For specifying data sources the execution environment has several methods to read from files using various methods: you can just read them line by line, as CSV files, or using any of the other provided sources.

```
final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> text = env.readTextFile("file:///path/to/file");
```

This will create a new DataStream by converting every String in the original collection to an Integer.

You apply transformations by calling methods on DataStream with a transformation functions

```
DataStream<String> input = ...;

DataStream<Integer> parsed = input.map(new MapFunction<String, Integer>() {
    @Override
    public Integer map(String value) {
        return Integer.parseInt(value);
    }
});
```

The `execute()` method will wait for the job to finish and then return a `JobExecutionResult`

Once you have a DataStream containing your final results, you can write it to an outside system by creating a sink. These are just some example methods for creating a sink:

```
writeAsText(String path);

print();
```

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class WindowWordCount {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Tuple2<String, Integer>> dataStream = env
                .socketTextStream("localhost", 9999)
                .flatMap(new Splitter())
                .keyBy(value -> value.f0)
                .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
                .sum(1);
        dataStream.print();

        env.execute("Window WordCount");
    }

    public static class Splitter implements FlatMapFunction<String, Tuple2<String,
Integer>> {
        @Override
        public void flatMap(String sentence, Collector<Tuple2<String, Integer>>
out) throws Exception {
            for (String word: sentence.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }

}
```

Reference

https://nightlies.apache.org/flink/flink-docs-release-1.1/apis/batch/examples.html#word-count