Apache Flink

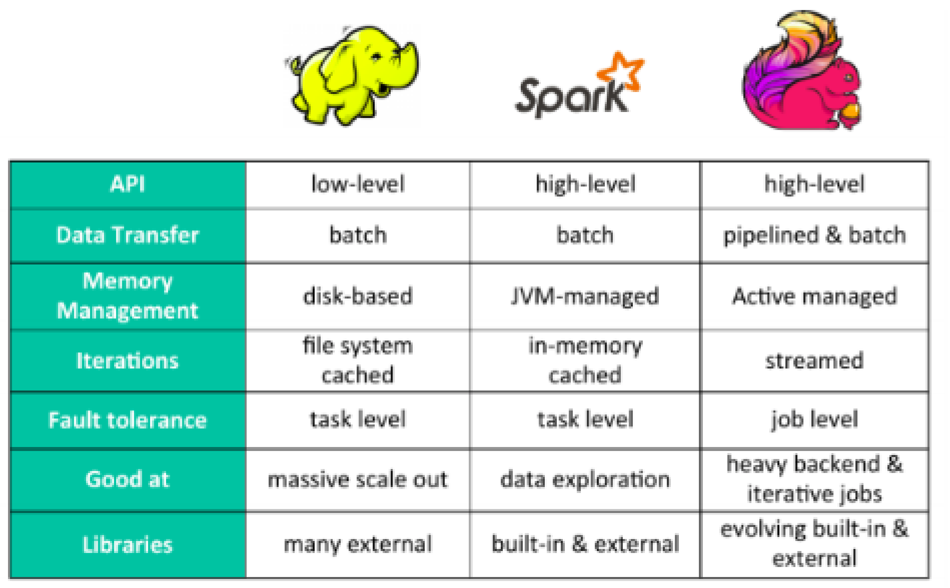
Apache Flink® is an open-source stream processing framework for distributed, high-performing, always-available, and accurate data streaming applications. It:

- provides results that are **accurate**, even in the case of out-of-order or late-arriving data
- is **stateful** and **fault-tolerant** thanks to *distributed snapshots*:
    - *at least once* : all operators see all events
    - *exactly once* : ensure that operators do not perform duplicate updates to their state
- performs at **large scale**, running on thousands of nodes with very good **throughput** and **latency** characteristics

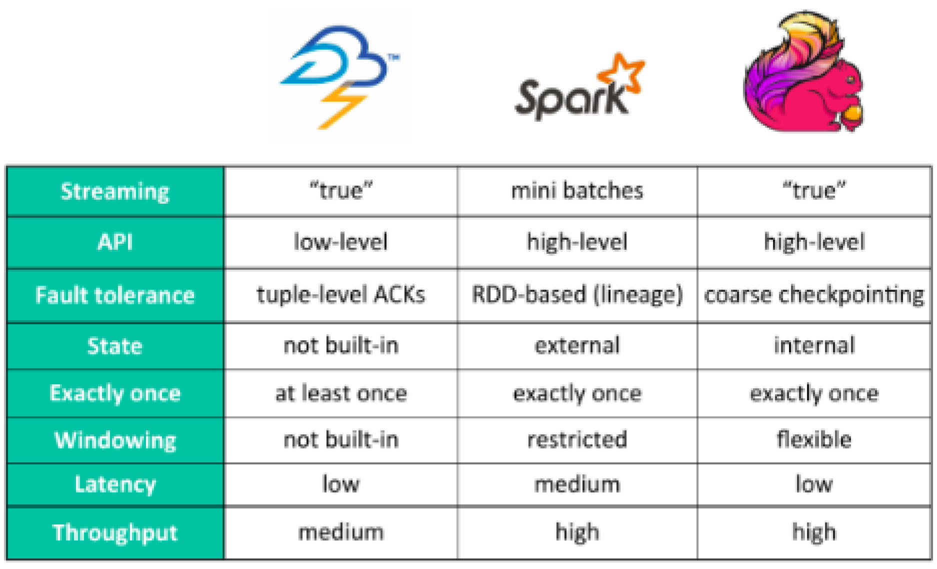
Flink is born to allow for **real-time streaming** computation, such as in tweets analysis, sentiment analysis and predictions. It supports stream processing and windowing with **event time semantics**, which means that computation can be done on streams when events arrive, as well as **flexible windowing** based on time, count, or session.

Moreover, it is very fast, as there is no need to write to disk, and the code is very easy to write.

When compared with other BigData architectures (respectively HadoopMR and Spark for batch processing, Storm and Spark Streaming for stream processing):

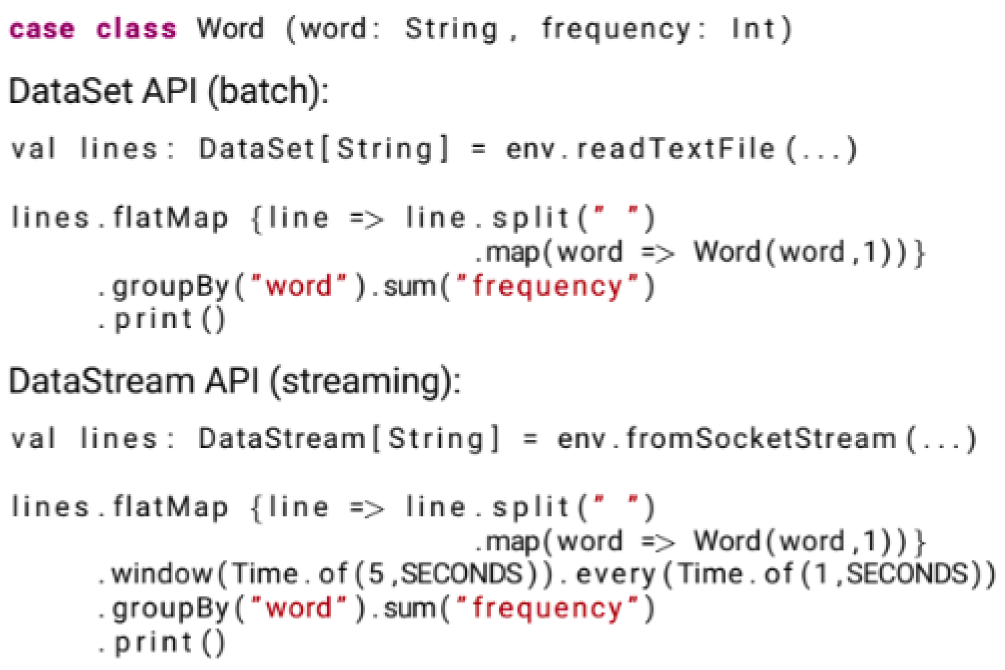


		Spark	
API	low-level	high-level	high-level
Data Transfer	batch	batch	pipelined & batch
Memory Management	disk-based	JVM-managed	Active managed
Iterations	file system cached	in-memory cached	streamed
Fault tolerance	task level	task level	job level
Good at	massive scale out	data exploration	heavy backend & iterative jobs
Libraries	many external	built-in & external	evolving built-in & external



		Spark	
Streaming	"true"	mini batches	"true"
API	low-level	high-level	high-level
Fault tolerance	tuple-level ACKs	RDD-based (lineage)	coarse checkpointing
State	not built-in	external	internal
Exactly once	at least once	exactly once	exactly once
Windowing	not built-in	restricted	flexible
Latency	low	medium	low
Throughput	medium	high	high

Using the Flink API for Scala, the *wordcount* problem becomes:



```
case class Word (word: String, frequency: Int)
```

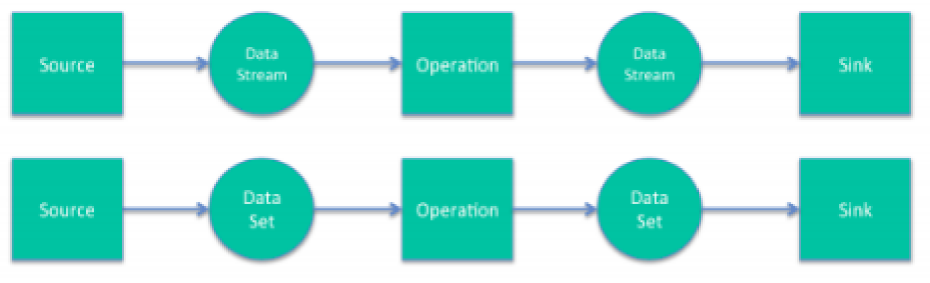
DataSet API (batch):

```
val lines: DataSet[String] = env.readTextFile(...)

lines.flatMap {line => line.split(" ")
                        .map(word => Word(word,1))}
     .groupBy("word").sum("frequency")
     .print()
```

DataStream API (streaming):

```
val lines: DataStream[String] = env.fromSocketStream(...)

lines.flatMap {line => line.split(" ")
                        .map(word => Word(word,1))}
     .window(Time.of(5,SECONDS)).every(Time.of(1,SECONDS))
     .groupBy("word").sum("frequency")
     .print()
```

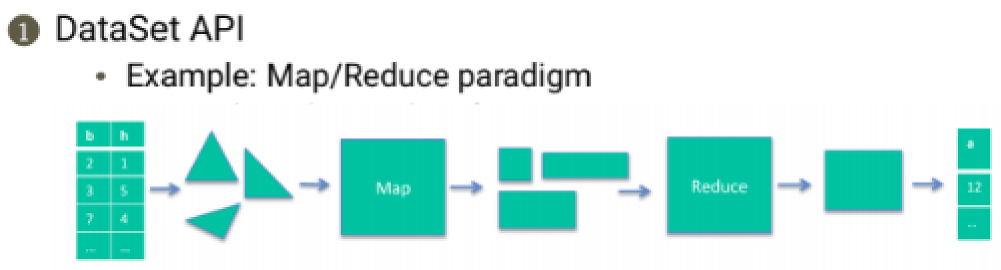
Apache Flink API



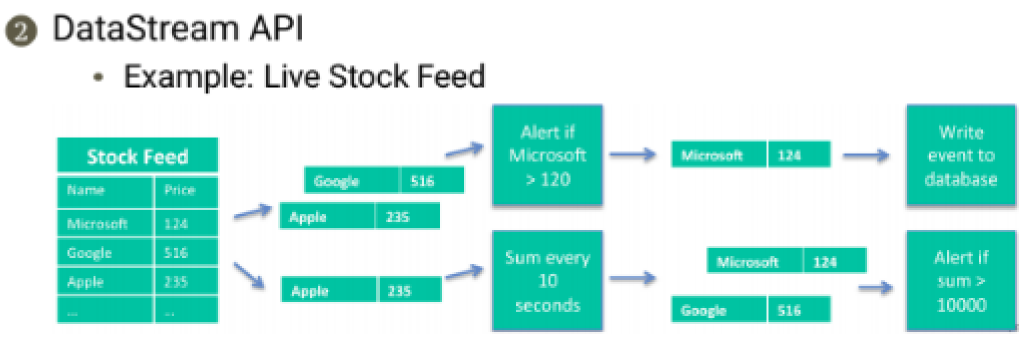
Any Flink program is composed of three phases: the loading of **source data**, which can be treated either as batch (*DataSet API* for finite set of data) or as stream (*DataStream API* for unbound set of data). Sources can be:

- collection-based
  - fromCollection
  - fromElements
- File-based
  - TextInputFormat
  - CsvInputFormat
- Other
  - SocketInputFormat
  - KafkaInputFormat
  - Databases

Then, **operations** are applied on the data set. Operations are defined within the *app topology*. In case of the *DataSet API* follows a standard MapReduce paradigm: input data is partitioned in batches, mapped via a Mapper function, then Reduced to the desired output format.



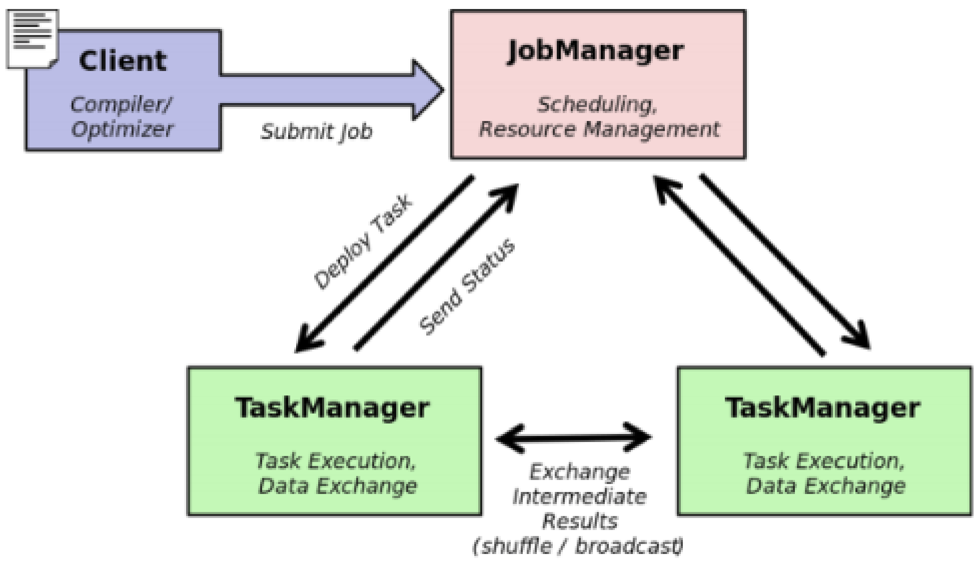
With the *DataStream API* instead, the windowed stream is moved along the previously defined topology, where each element of the topology can act as a "trigger" or "control" for a certain event, or be a normal operation to be applied on the incoming data. *DataStream API* allows for great concurrency of operations, by reading the same input data simultaneously.



Finally, the data is given in output through **data sinks**, which can be:

- File-based
  - TextOutputFormat
  - CsvOutputFormat
  - PrintOutput
- Others
  - SocketOutputFormat
  - KafkaOutputFormat
  - Databases

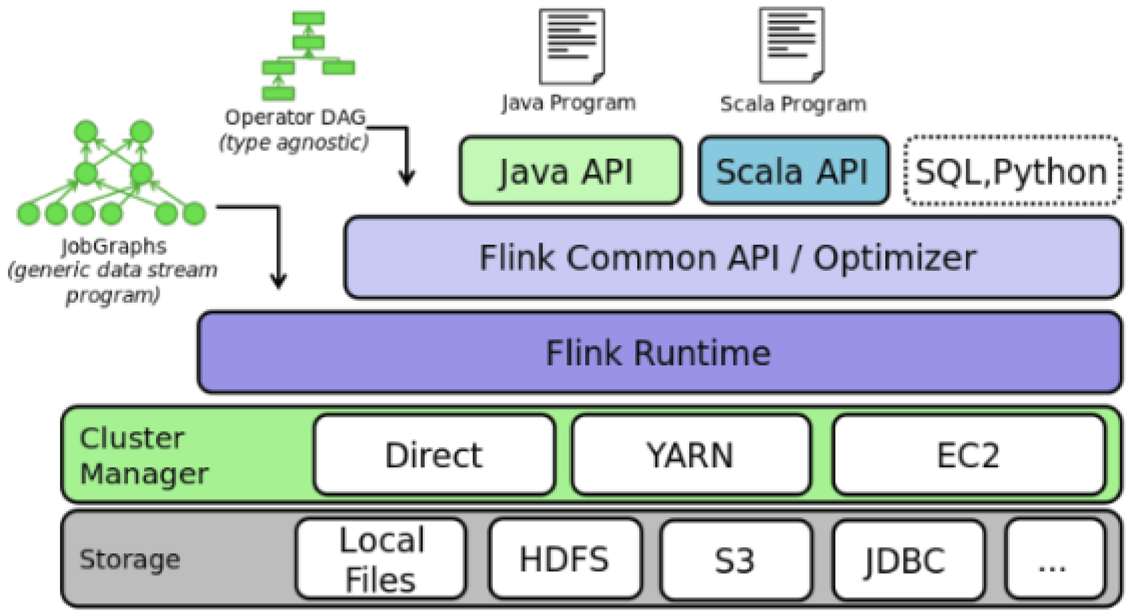
Apache Flink Architecture


Client
Compiler/ Optimizer
Submit Job
JobManager
Scheduling, Resource Management
Deploy Task
Send Status
TaskManager
Task Execution, Data Exchange
Exchange Intermediate Results (shuffle / broadcast)
TaskManager
Task Execution, Data Exchange

The Flink runtime consists of two types of processes:

- The **JobManagers** (also called masters) coordinate the distributed execution. They schedule tasks, coordinate checkpoints, coordinate recovery on failures, etc. There is always at least one Job Manager. A high-availability setup will have multiple JobManagers, one of which one is always the leader, and the others are standby.
- The **TaskManagers** (also called workers) execute the tasks (or more specifically, the subtasks) of a dataflow, and buffer and exchange the data streams. Each parallel instance of an operation runs in a separate **task slot**. The scheduler may run several tasks from different operators in one task slot. There must always be at least one *TaskManager*.
- The **client** is not exactly part of the runtime and program execution, but is used to prepare and send a dataflow (job graph) to the *JobManager*. After that, the client can disconnect, or stay connected to receive progress reports.

Flink Component Stack



- API layer: implements multiple APIs that create operator DAGs for their programs. Each API needs to provide utilities (serializers, comparators) that describe the interaction between its data types and the runtime.
- Optimizer and common api layer: takes programs in the form of operator DAGs. The operators are specific (e.g., Map, Join, Filter, Reduce, ...), but are data type agnostic.
- Runtime layer: receives a program in the form of a JobGraph. A JobGraph is a generic parallel data flow with arbitrary tasks that consume and produce data streams.

Flink Algorithms

The basic skeleton of a Flink program is composed of five steps:

1. Obtain an *ExecutionEnvironment*/*StreamExecutionEnvironment*

    - [StreamExecutionEnvironment.]getExecutionEnvironment()
    - [StreamExecutionEnvironment.]createLocalEnvironment()
    - [StreamExecutionEnvironment.]createRemoteEnvironment()
2. Load/create the initial data from the data sources (specified above)
3. Specify transformations on this data
    - Map - one-to-one transformation
    - FlatMap - one-to-many transformation
    - Filter - evaluate each element returning boolean for filter satisfaction
    - Reduce - combine a group of elements by combining two at a time
    - ReduceGroup - combine a group of elements into one or more elements
    - Aggregate - aggregates a group of values into a single value, may be applied on a full data set, or on a grouped data set
    - CoGroup
    - Cross
    - Union
    - Rebalance
    - partitionByHash()
    - sortPartition()
    - first(n)
    - for *DataStream API*, use *Window operators*
4. Specify where to put the results of your computations
5. Trigger the program execution