Data Processing with Flink and Spark

Author: Nguyen Truong Duong

Email: seedotech@gmail.com

Content

1. Big Data Analysis Overview

Big Data Analysis Overview

Data Sources	Data Integration	Data Storage	Data Analysis	Data Visualization
• Structured Data • Unstructured Data	• Data Collection • Data Cleansing • Data Streaming • Data Processing	• Object Storage • File Storage • Databases	• Machine Learning • Deep Learning	• Reports • Charts • Dashboards

SQL DB

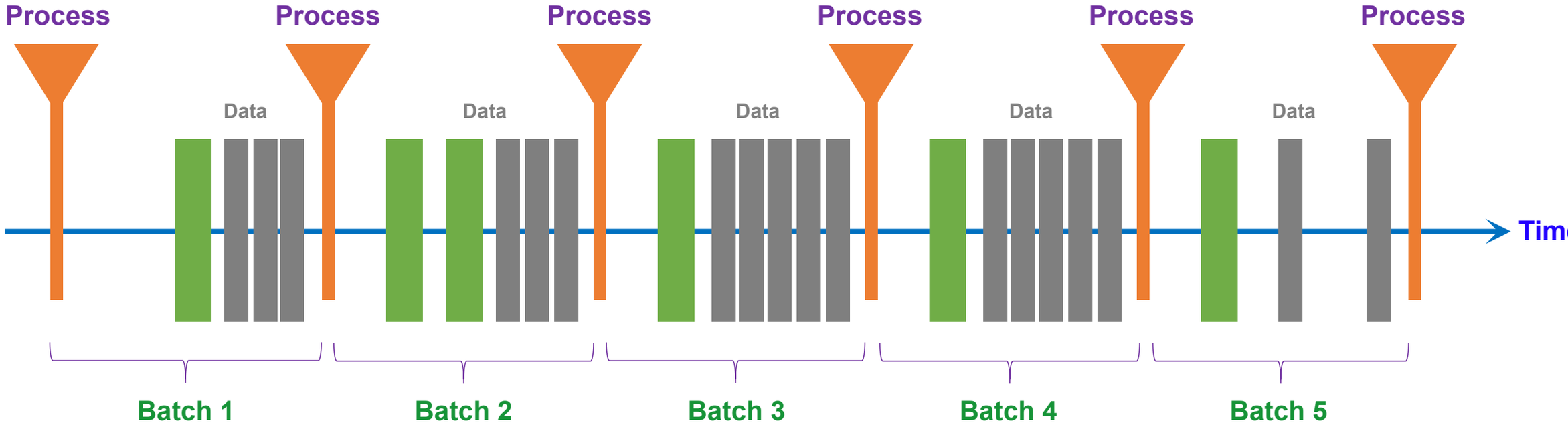
NoSQL DB

2. Batch Processing vs Stream Processing

Batch Processing vs Stream Processing

1. Batch Processing

In **batch** processing, newly arriving data elements are collected into a group. The whole group is then processed at a future time (as a batch, hence the term “batch processing”). Exactly when each group is processed can be determined in a number of ways–for example, it can be based on a scheduled time interval (e.g. every five minutes, process whatever new data has been collected) or on some triggered condition (e.g. process the group as soon as it contains five data elements or as soon as it has more than 1MB of data).

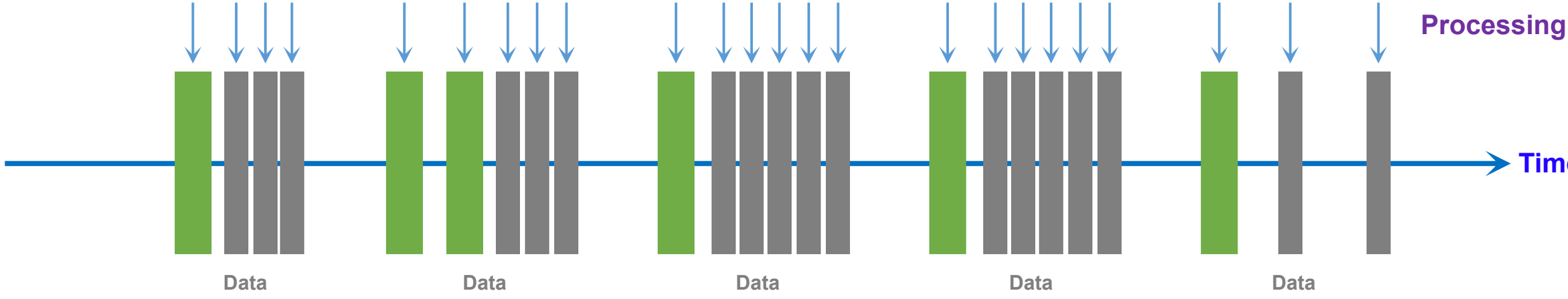


Micro-Batch is frequently used to describe scenarios where batches are small and/or processed at small intervals. Even though processing may happen as often as once every few minutes, data is still processed a batch at a time.

Batch Processing vs Stream Processing

2. Stream Processing

In stream processing, each new piece of data is processed when it arrives. Unlike batch processing, there is no waiting until the next batch processing interval and data is processed as individual pieces rather than being processed a batch at a time.



Use cases:

- Algorithmic Trading, Stock Market Surveillance
- Monitoring a production line
- Intrusion, Surveillance and Fraud Detection (e.g. Uber)
- Predictive Maintenance, (e.g. Machine Learning Techniques for Predictive Maintenance)

3. Apache Flink

Flink Ecosystem

1. Storage / Streaming

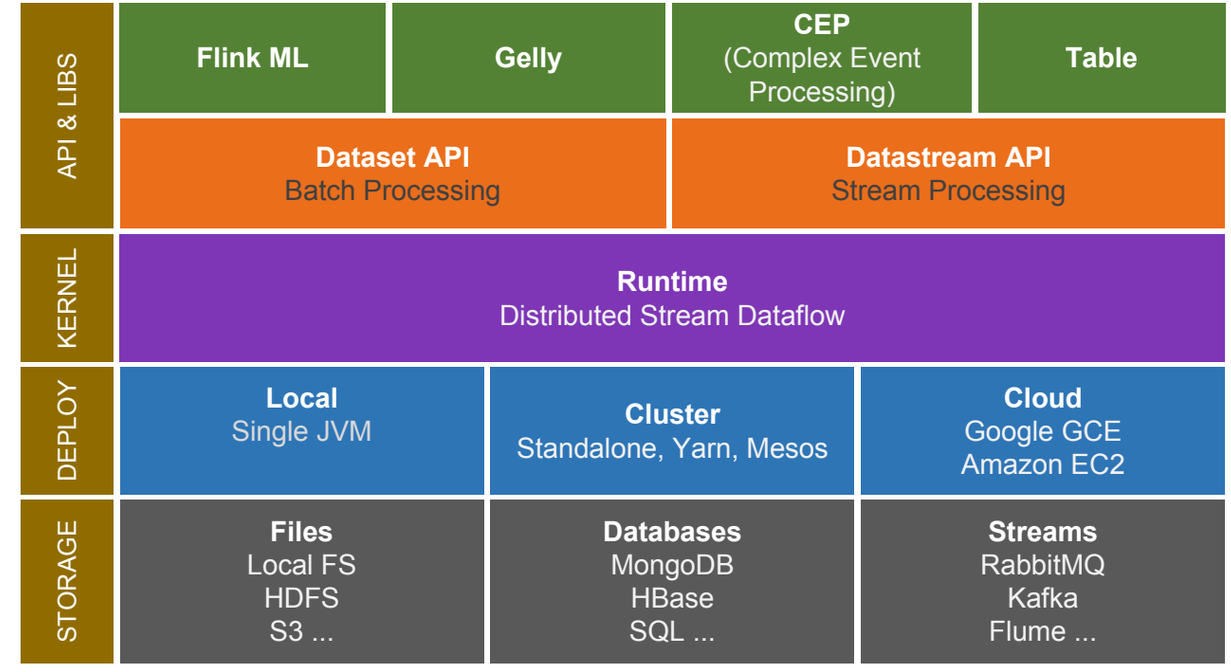
Flink doesn't ship with the storage system; it is just a computation engine. Flink can read, write data from different storage system as well as can consume data from streaming systems. Below is the list of storage/streaming system from which Flink can read write data:

- **HDFS** – Hadoop Distributed File System
- **Local-FS** – Local File System
- **S3** – Simple Storage Service from Amazon
- **HBase** – NoSQL Database in Hadoop ecosystem
- **MongoDB** – NoSQL Database
- **RDBMS** – Any relational database
- **Kafka** – Distributed messaging Queue
- **RabbitMQ** – Messaging Queue
- **Flume** – Data Collection and Aggregation Tool

2. Deploy

Flink can be deployed in following modes:

- **Local mode** – On a single node, in single JVM
- **Cluster** – On a multi-node cluster, with following resource manager.
- **Standalone** – This is the default resource manager which is shipped with Flink.
- **YARN** – This is a very popular resource manager, it is part of Hadoop
- **Mesos** – This is a generalized resource manager.
- **Cloud** – on Amazon or Google cloud



Flink Ecosystem

3. Distributed Streaming Dataflow

It is also called as the kernel of Apache Flink. This is the core layer of flink which provides distributed processing, fault tolerance, reliability, native iterative processing capability,...

4. APIs and Library

- **DataSet API**

  It allows the user to implement operations like map, filter, join, group, etc. on the dataset. It is mainly used for distributed processing.

- **DataStream API**

  It handles a continuous stream of the data. To process live data stream it provides various operations like map, filter, update states, window, aggregate, etc. It can consume the data from the various streaming source and can write the data to different sinks.

- **Table**

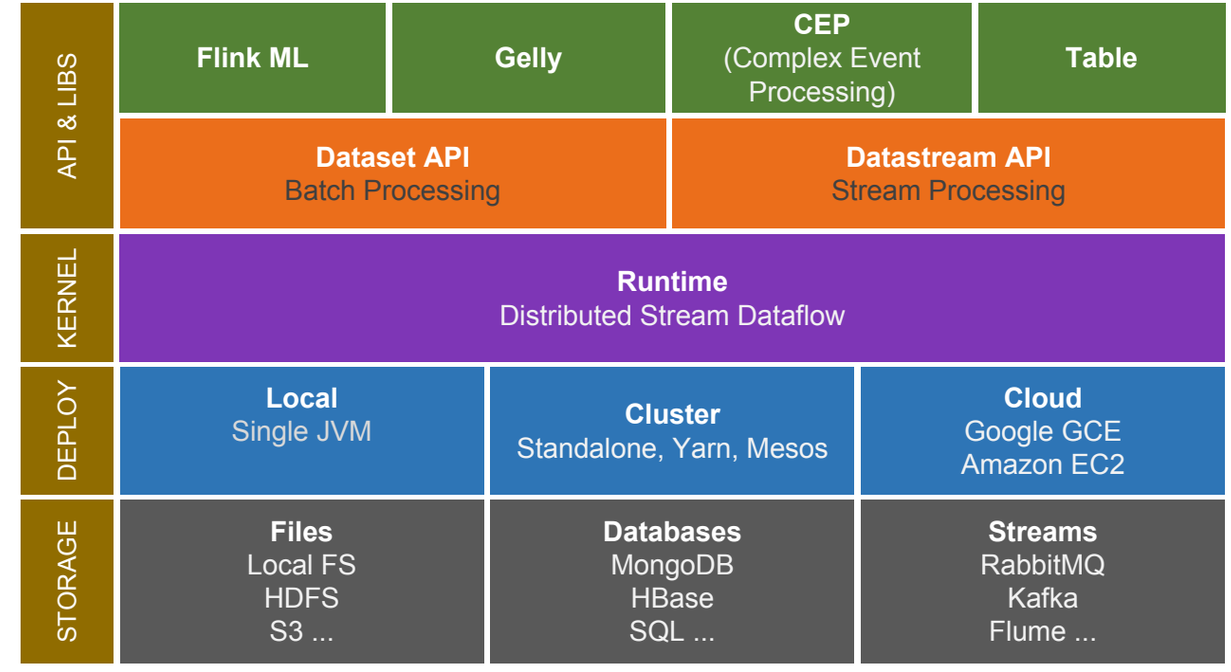
  It enables users to perform ad-hoc analysis using SQL like expression language for relational stream and batch processing. It can be embedded in DataSet and DataStream APIs.

- **Gelly**

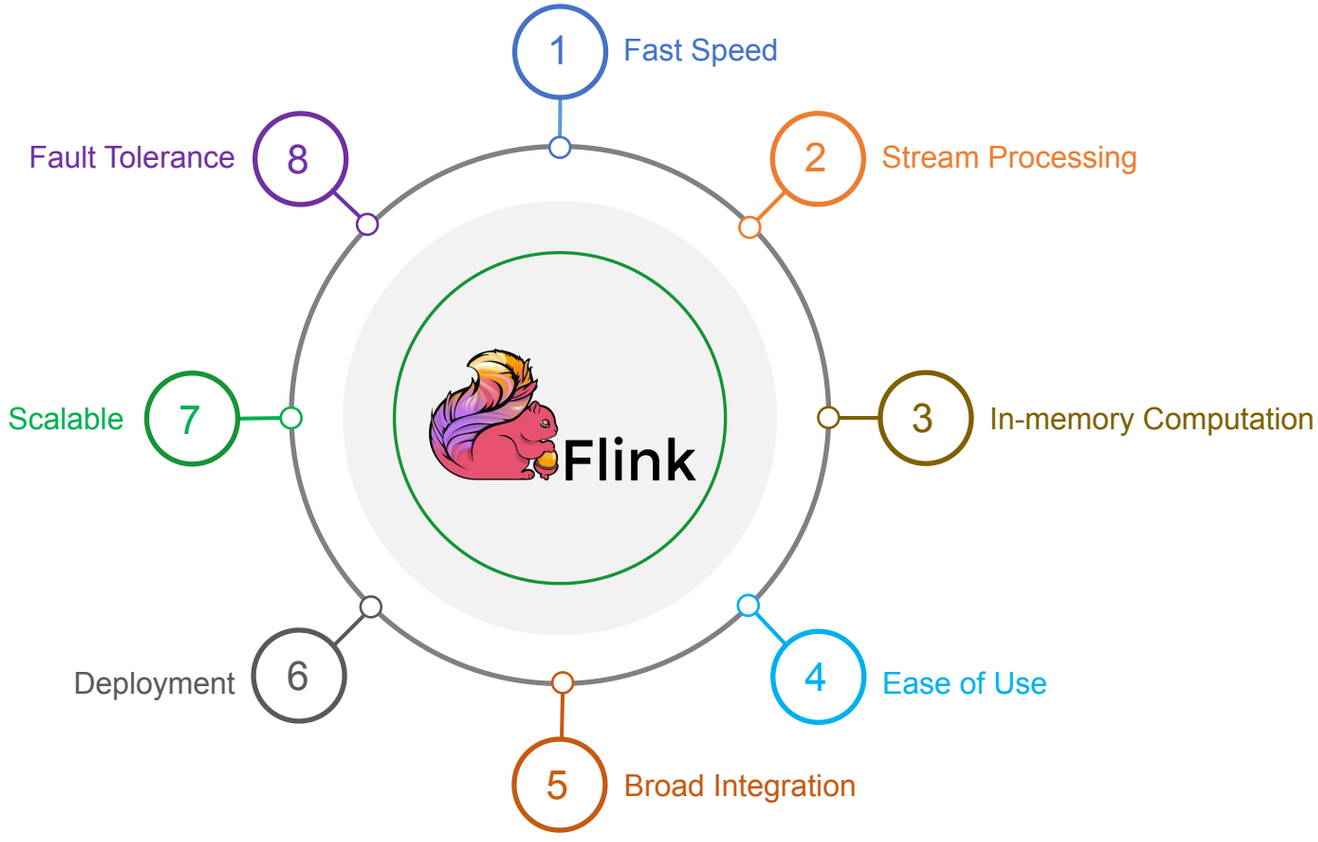
  It is the graph processing engine which allows users to run set of operations to create, transform and process the graph.

- **FlinkML**

  It is the machine learning library which provides intuitive APIs and an efficient algorithm to handle machine learning applications.



Flink Features



1. Fast Speed

Flink processes data at lightning fast speed (hence also called as 4G of Big Data).

2. Stream Processing

Flink is a true stream processing engine.

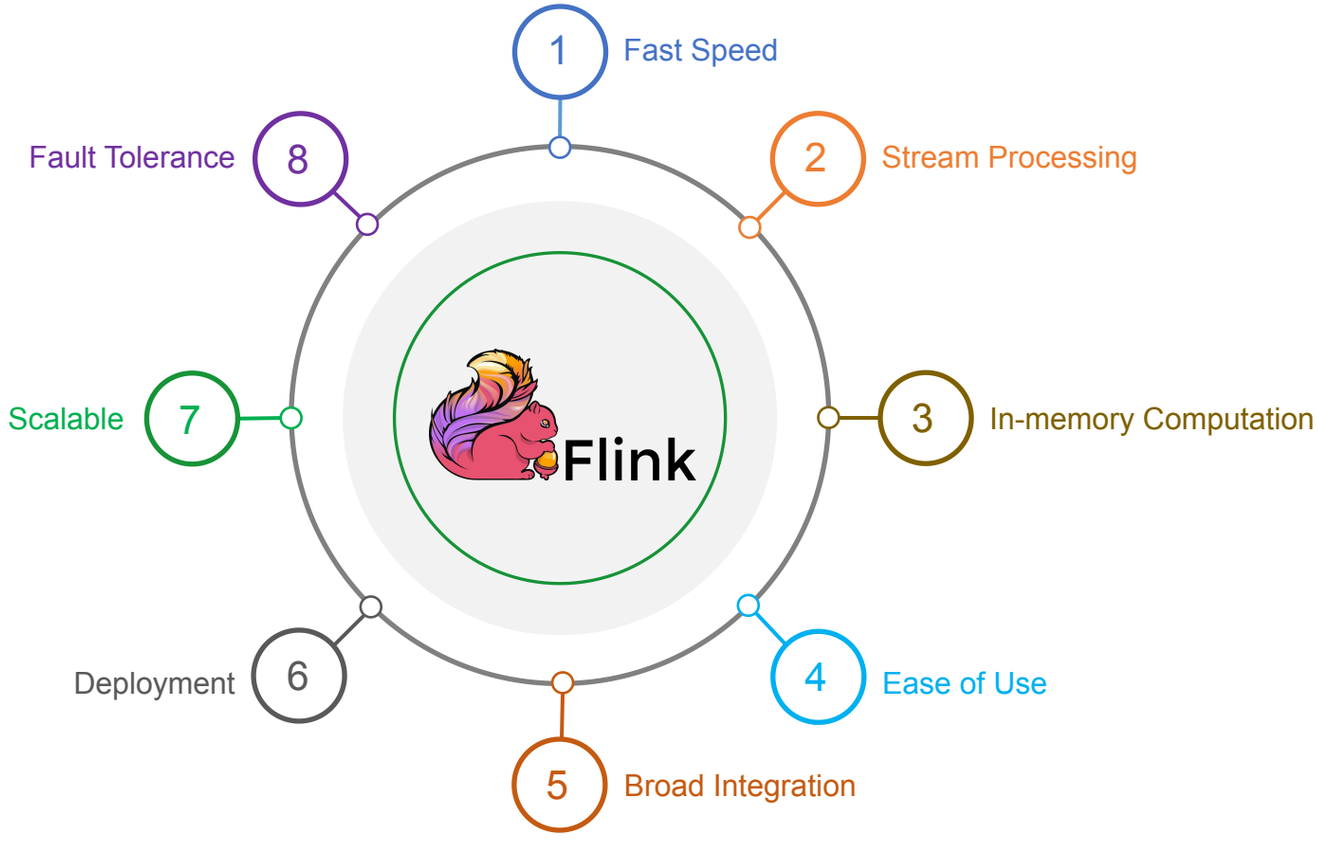
3. In-memory Computation

Data is kept in random access memory(RAM) instead of some slow disk drives and is processed in parallel. It improves the performance by an order of magnitudes by keeping the data in memory.

4. Ease of Use

Flink's APIs are developed in a way to cover all the common operations, so programmers can use it efficiently.

Flink Features



5. Broad integration

Flink can be integrated with the various storage system to process their data, it can be deployed with various resource management tools. It can also be integrated with several BI tools for reporting.

6. Deployment

It can be deployed through Mesos, Hadoop via YARN, or Flink's own cluster manager or cloud (Amazon, Google cloud).

7. Scalable

Flink is highly scalable. With increasing requirements, we can scale the flink cluster.

8. Fault Tolerance

Failure of hardware, node, software or a process doesn’t affect the cluster.

Flink Architecture

Apache Flink is an open source platform for distributed stream and batch data processing. Flink's core is a streaming dataflow engine that provides data distribution, communication, and fault tolerance for distributed computations over data streams. Flink builds batch processing on top of the streaming engine, overlaying native iteration support, managed memory, and program optimization.

1. Program

It is a piece of code, which you run on the **Flink Cluster**.

2. Client

It is responsible for taking code (program) and constructing job dataflow graph, then passing it to **JobManager**. It also retrieves the Job results.

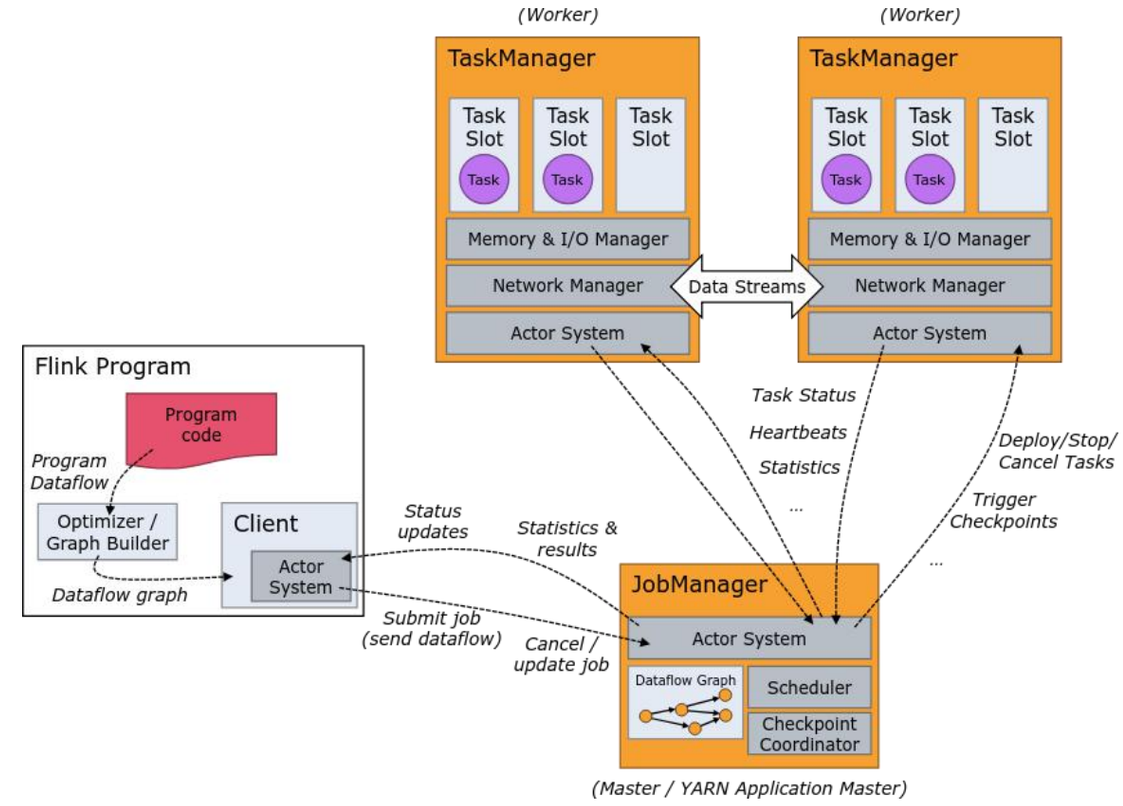
3. JobManager

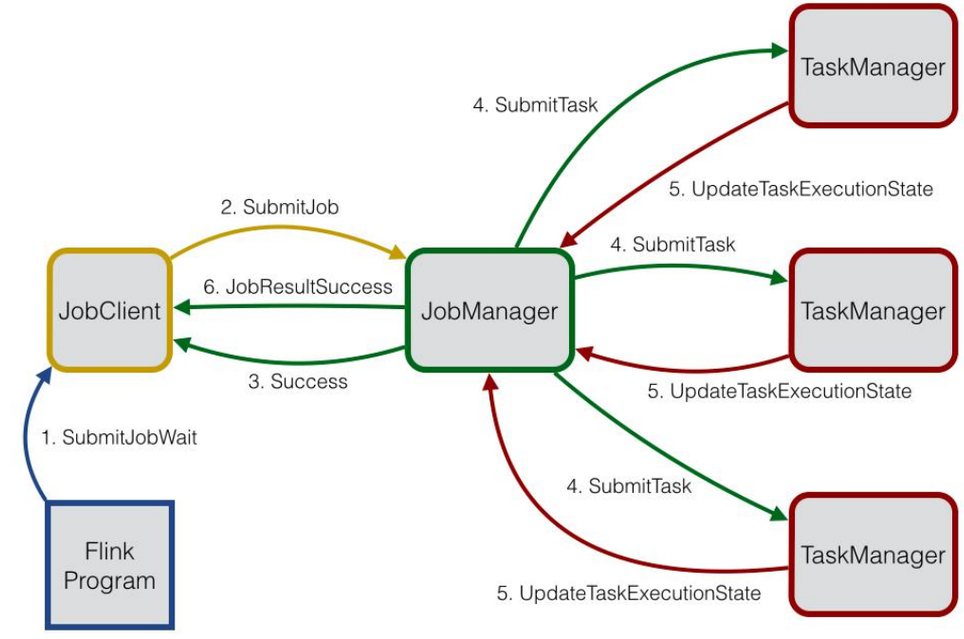
Also called **masters**. They coordinate the distributed execution. They schedule tasks, coordinate checkpoints, coordinate recovery on failures, etc. After receiving the Job **Dataflow Graph** from Client, it is responsible for creating the execution graph. It assigns the job to **TaskManagers** in the cluster and supervises the execution of the job.

There is always at least one **JobManager**. A high-availability setup will have multiple **JobManagers**, one of which one is always the leader, and the others are standby.

4. TaskManager

Also called **workers** or **slaves**. They execute the tasks that have been assigned by **JobManager** (or more specifically, the subtasks) of a dataflow, and buffer and exchange the data streams. All the **TaskManagers** run the tasks in their separate slots in specified parallelism. It is responsible to send the status of the tasks to **JobManager**.





Programs and Dataflows

The basic building blocks of Flink programs are streams and transformations.

Conceptually a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.

When executed, Flink programs are mapped to streaming dataflows, consisting of streams and transformation operators. Each dataflow starts with one or more sources and ends in one or more sinks. The dataflows resemble arbitrary directed acyclic graphs (DAGs). Although special forms of cycles are permitted via iteration constructs, for the most part we will gloss over this for simplicity.

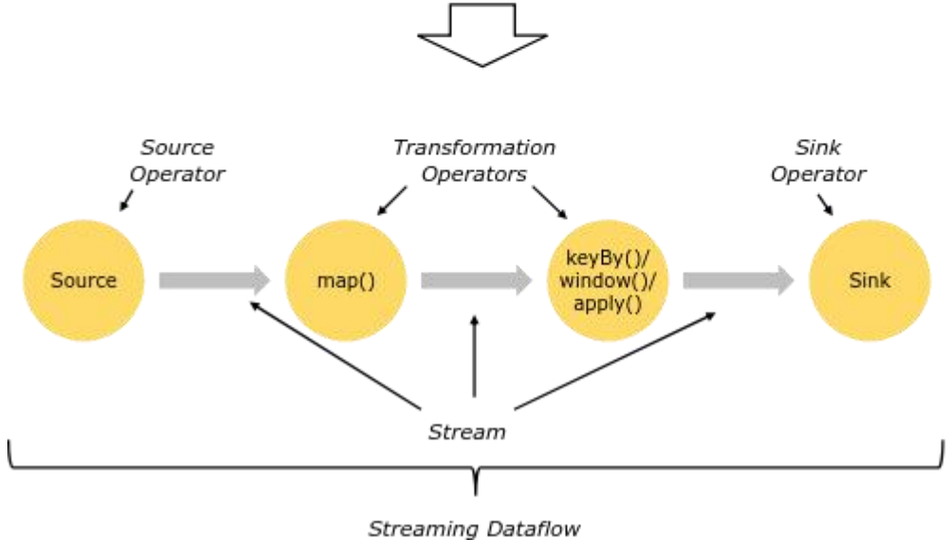
```
DataStream<String> lines = env.addSource(
                           new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

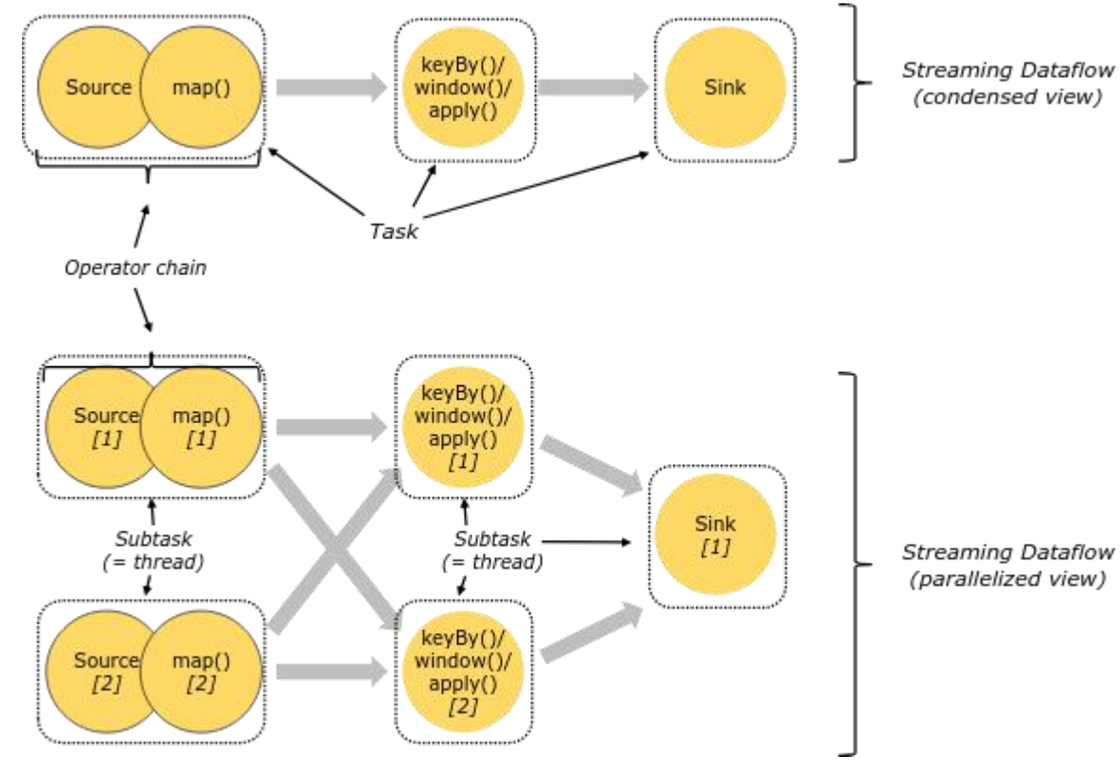
Source

Transformation

Transformation

Sink





Data Source

- Sources are where the program reads its input from.
- A source is attached to the program by using **StreamExecutionEnvironment**.addSource(sourceFunction).
- Flink comes with a number of pre-implemented source functions, but you can always write your own custom sources by implementing the **SourceFunction** for non-parallel sources, or by implementing the **ParallelSourceFunction** interface or extending the **RichParallelSourceFunction** for parallel sources.

There are several predefined stream sources accessible from the **StreamExecutionEnvironment**:

1. File-based:

- **readTextFile(path)** - Reads text files, i.e. files that respect the TextInputFormat specification, line-by-line and returns them as Strings.
- **readFile(fileInputFormat, path)** - Reads (once) files as dictated by the specified file input format.
- **readFile(fileInputFormat, path, watchType, interval, pathFilter, typeInfo)** - This is the method called internally by the two previous ones. It reads files in the path based on the given fileInputFormat. Depending on the provided watchType, this source may periodically monitor (every interval ms) the path for new data (FileProcessingMode.PROCESS_CONTINUOUSLY), or process once the data currently in the path and exit (FileProcessingMode.PROCESS_ONCE). Using the pathFilter, the user can further exclude files from being processed.

2. Socket-based:

- **socketTextStream** - Reads from a socket. Elements can be separated by a delimiter.

Data Source

3. Collection-based:

- **fromCollection(Collection)** - Creates a data stream from the Java Java.util.Collection. All elements in the collection must be of the same type.
- **fromCollection(Iterator, Class)** - Creates a data stream from an iterator. The class specifies the data type of the elements returned by the iterator.
- **fromElements(T ...)** - Creates a data stream from the given sequence of objects. All objects must be of the same type.
- **fromParallelCollection(SplittableIterator, Class)** - Creates a data stream from an iterator, in parallel. The class specifies the data type of the elements returned by the iterator.
- **generateSequence(from, to)** - Generates the sequence of numbers in the given interval, in parallel.

4. Custom:

- **addSource** - Attach a new source function. For example, to read from Apache Kafka you can use addSource(new FlinkKafkaConsumer08<>(...)). See connectors for more details.

Collection Data Sources

Flink provides special data sources which are backed by Java collections to ease testing. Once a program has been tested, the sources and sinks can be easily replaced by sources and sinks that read from / write to external systems.

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

// Create a DataStream from a list of elements
DataStream<Integer> myInts = env.fromElements(1, 2, 3, 4, 5);

// Create a DataStream from any Java collection
List<Tuple2<String, Integer>> data = ...
DataStream<Tuple2<String, Integer>> myTuples = env.fromCollection(data);

// Create a DataStream from an Iterator
Iterator<Long> longIt = ...
DataStream<Long> myLongs = env.fromCollection(longIt, Long.class);
```

Note: *Currently, the collection data source requires that data types and iterators implement Serializable. Furthermore, collection data sources can not be executed in parallel (parallelism = 1).*

DataStream Transformations

https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/index.html

https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch

https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/dataset_transformations.html

DataStream Transformations: Functions

Most transformations require user-defined functions. This section lists different ways of how they can be specified.

1. Implementing an interface

The most basic way is to implement one of the provided interfaces:

```
class MyMapFunction implements MapFunction<String, Integer> {
	public Integer map(String value) {
		return Integer.parseInt(value);
	}
};
data.map(new MyMapFunction());
```

2. Anonymous classes

You can pass a function as an anonymous class:

```
data.map(new MapFunction<String, Integer> () {
	public Integer map(String value) {
		return Integer.parseInt(value);
	}
});
```

3. Java 8 Lambdas

Flink also supports Java 8 Lambdas in the Java API.

```
data.filter(s -> s.startsWith("http://"));
data.reduce((i1, i2) -> i1 + i2);
```

Windows

Windows are at the heart of processing infinite streams. Windows split the stream into “buckets” of finite size, over which we can apply computations. This document focuses on how windowing is performed in Flink and how the programmer can benefit to the maximum from its offered functionality.

The general structure of a windowed Flink program is presented below. The first snippet refers to keyed streams, while the second to non-keyed ones. As one can see, the only difference is the keyBy(...) call for the keyed streams and the window(...) which becomes windowAll(...) for non-keyed streams. This is also going to serve as a roadmap for the rest of the page.

Keyed Windows

```
stream
       .keyBy(...)                <-  keyed versus non-keyed windows
       .window(...)               <-  required: "assigner"
      [.trigger(...)]             <-  optional: "trigger" (else default trigger)
      [.evictor(...)]             <-  optional: "evictor" (else no evictor)
      [.allowedLateness(...)]     <-  optional: "lateness" (else zero)
      [.sideOutputLateData(...)]  <-  optional: "output tag" (else no side output for late data)
       .reduce/aggregate/fold/apply()      <-  required: "function"
      [.getSideOutput(...)]       <-  optional: "output tag"
```

Non-Keyed Windows

```
stream
       .windowAll(...)            <-  required: "assigner"
      [.trigger(...)]             <-  optional: "trigger" (else default trigger)
      [.evictor(...)]             <-  optional: "evictor" (else no evictor)
      [.allowedLateness(...)]     <-  optional: "lateness" (else zero)
      [.sideOutputLateData(...)]  <-  optional: "output tag" (else no side output for late data)
       .reduce/aggregate/fold/apply()      <-  required: "function"
      [.getSideOutput(...)]       <-  optional: "output tag"
```

See more: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html

Data Sink

Data sinks consume **DataStreams** and forward them to files, sockets, external systems, or print them. Flink comes with a variety of built-in output formats that are encapsulated behind operations on the **DataStreams**:

- **writeAsText() / TextOutputFormat** - Writes elements line-wise as **Strings**. The Strings are obtained by calling the toString() method of each element.
- **writeAsCsv(...) / CsvOutputFormat** - Writes tuples as comma-separated value files. Row and field delimiters are configurable. The value for each field comes from the toString() method of the objects.
- **print() / printToErr()** - Prints the toString() value of each element on the standard out / standard error stream. Optionally, a prefix (msg) can be provided which is prepended to the output. This can help to distinguish between different calls to print. If the parallelism is greater than 1, the output will also be prepended with the identifier of the task which produced the output.
- **writeUsingOutputFormat() / FileOutputFormat** - Method and base class for custom file outputs. Supports custom object-to-bytes conversion.
- **writeToSocket** - Writes elements to a socket according to a **SerializationSchema**
- **addSink** - Invokes a custom sink function. Flink comes bundled with connectors to other systems (such as Apache Kafka) that are implemented as sink functions.

Iterations

Iterative streaming programs implement a step function and embed it into an IterativeStream. As a DataStream program may never finish, there is no maximum number of iterations. Instead, you need to specify which part of the stream is fed back to the iteration and which part is forwarded downstream using a split transformation or a filter. Here, we show an example using filters. First, we define an IterativeStream

```
IterativeStream<Integer> iteration = input.iterate();
```

Then, we specify the logic that will be executed inside the loop using a series of transformations (here a simple map transformation)

```
DataStream<Integer> iterationBody = iteration.map(/* this is executed many times */);
```

To close an iteration and define the iteration tail, call the closeWith(feedbackStream) method of the IterativeStream. The DataStream given to the closeWith function will be fed back to the iteration head. A common pattern is to use a filter to separate the part of the stream that is fed back, and the part of the stream which is propagated forward. These filters can, e.g., define the “termination” logic, where an element is allowed to propagate downstream rather than being fed back.

```
iteration.closeWith(iterationBody.filter(/* one part of the stream */));
DataStream<Integer> output = iterationBody.filter(/* some other part of the stream */);
```

See more: *https://ci.apache.org/projects/flink/flink-docs-stable/dev/datastream_api.html*

Data Types

1. Tuples

Tuples are composite types that contain a fixed number of fields with various types. The Java API provides classes from **Tuple1** up to **Tuple25**. Every field of a tuple can be an arbitrary Flink type including further tuples, resulting in nested tuples. **Fields** of a tuple can be accessed directly using the field's name as **tuple.f4**, or using the generic getter method **tuple.getField(int position)**. The field indices start at 0. Note that this stands in contrast to the Scala tuples, but it is more consistent with Java's general indexing.

```
DataStream<Tuple2<String, Integer>> wordCounts = env.fromElements(
    new Tuple2<String, Integer>("hello", 1),
    new Tuple2<String, Integer>("world", 2));

wordCounts.map(new MapFunction<Tuple2<String, Integer>, Integer>() {
    @Override
    public Integer map(Tuple2<String, Integer> value) throws Exception {
        return value.f1;
    }
});

wordCounts.keyBy(0); // also valid .keyBy("f0")
```

Data Types

2. POJOs

Java and Scala classes are treated by Flink as a special POJO data type if they fulfill the following requirements:

- The class must be public.
- It must have a public constructor without arguments (default constructor).
- All fields are either public or must be accessible through getter and setter functions. For a field called foo the getter and setter methods must be named getFoo() and setFoo().
- The type of a field must be supported by Flink. At the moment, Flink uses Avro to serialize arbitrary objects (such as Date).

The following example shows a simple POJO with two public fields.

```
public class WordWithCount {
    public String word;
    public int count;

    public WordWithCount() {}

    public WordWithCount(String word, int count) {
        this.word = word;
        this.count = count;
    }
}

DataStream<WordWithCount> wordCounts = env.fromElements(
    new WordWithCount("hello", 1),
    new WordWithCount("world", 2));

wordCounts.keyBy("word"); // key by field expression "word"
```

Controlling Latency

By default, elements are not transferred on the network one-by-one (which would cause unnecessary network traffic) but are buffered. The size of the buffers (which are actually transferred between machines) can be set in the Flink config files. While this method is good for optimizing throughput, it can cause latency issues when the incoming stream is not fast enough. To control throughput and latency, you can use env.setBufferTimeout(timeoutMillis) on the execution environment (or on individual operators) to set a maximum wait time for the buffers to fill up. After this time, the buffers are sent automatically even if they are not full. The default value for this timeout is 100 ms.

```
LocalStreamEnvironment env = StreamExecutionEnvironment.createLocalEnvironment();
env.setBufferTimeout(timeoutMillis);

env.generateSequence(1,10).map(new MyMapper()).setBufferTimeout(timeoutMillis);
```

To maximize throughput, set setBufferTimeout(-1) which will remove the timeout and buffers will only be flushed when they are full. To minimize latency, set the timeout to a value close to 0 (for example 5 or 10 ms). A buffer timeout of 0 should be avoided, because it can cause severe performance degradation.

Lazy Evaluation

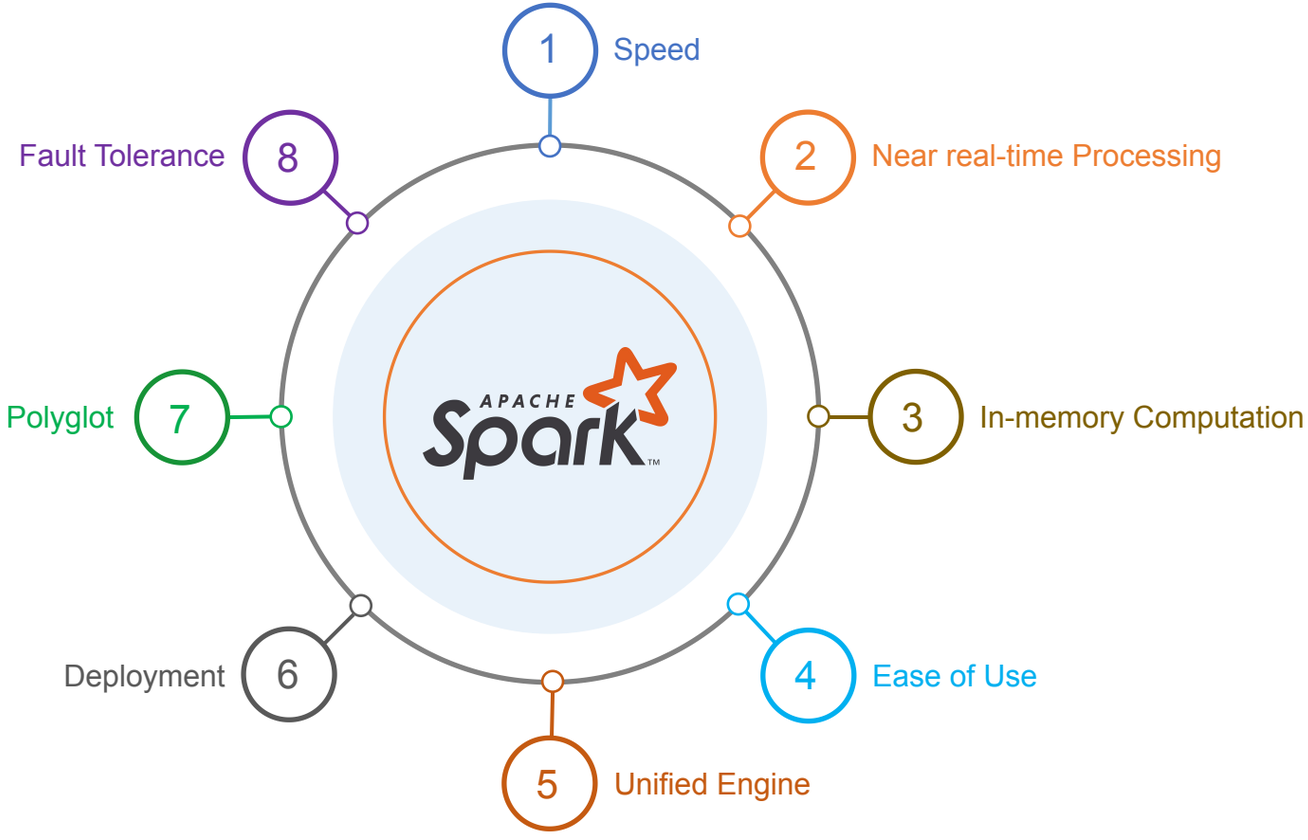
All Flink programs are executed lazily: When the program's main method is executed, the data loading and transformations do not happen directly. Rather, each operation is created and added to the program's plan. The operations are actually executed when the execution is explicitly triggered by an execute() call on the execution environment. Whether the program is executed locally or on a cluster depends on the type of execution environment.

The lazy evaluation lets you construct sophisticated programs that Flink executes as one holistically planned unit.

4. Apache Spark

Spark Features

Apache Spark is an open source cluster computing framework for real-time data processing. The main feature of Apache Spark is its in-memory cluster computing that increases the processing speed of an application. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. It is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries, and streaming.



1. Speed

Spark runs up to 100 times faster than Hadoop MapReduce for large-scale data processing. It is also able to achieve this speed through controlled partitioning. Source

2. Near real-time Processing

It offers near real-time computation & low latency because of in-memory computation.

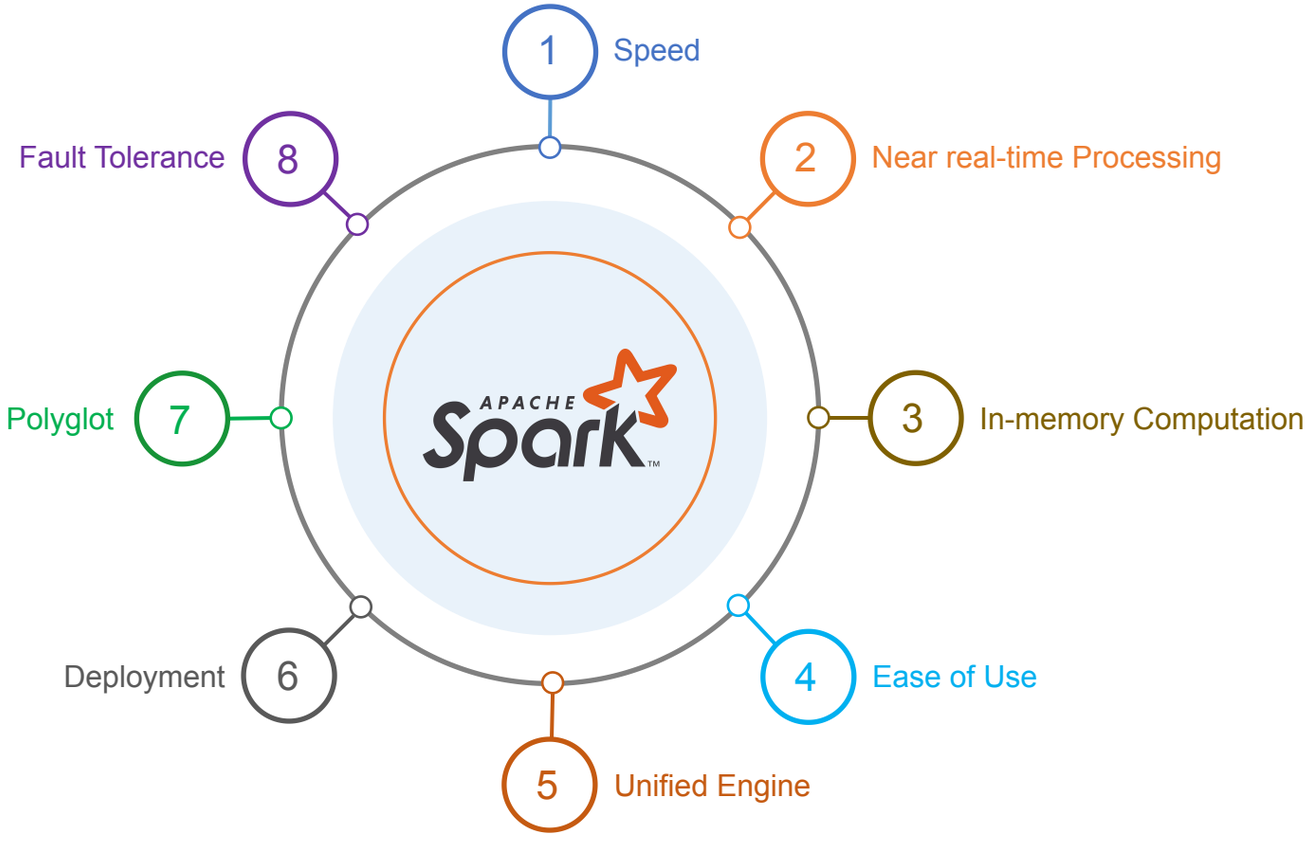
3. In-memory Computation

Data is kept in random access memory(RAM) instead of some slow disk drives and is processed in parallel. It improves the performance by an order of magnitudes by keeping the data in memory.

4. Ease of Use

Spark has easy-to-use APIs for operating on large datasets. This includes a collection of over 100 operators for transforming data and familiar data frame APIs for manipulating semi-structured data.

Spark Features



5. Unified Engine

Spark comes packaged with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing. These standard libraries increase developer productivity and can be seamlessly combined to create complex workflows.

6. Deployment

It can be deployed through Mesos, Hadoop via YARN, or Spark's own cluster manager.

7. Polyglot

Spark provides high-level APIs in Java, Scala, Python, and R. Spark code can be written in any of these four languages. It also provides a shell in Scala and Python.

8. Fault Tolerance

Upon the failure of worker node, using lineage of operations we can re-compute the lost partition of RDD from the original one. Thus, we can easily recover the lost data.

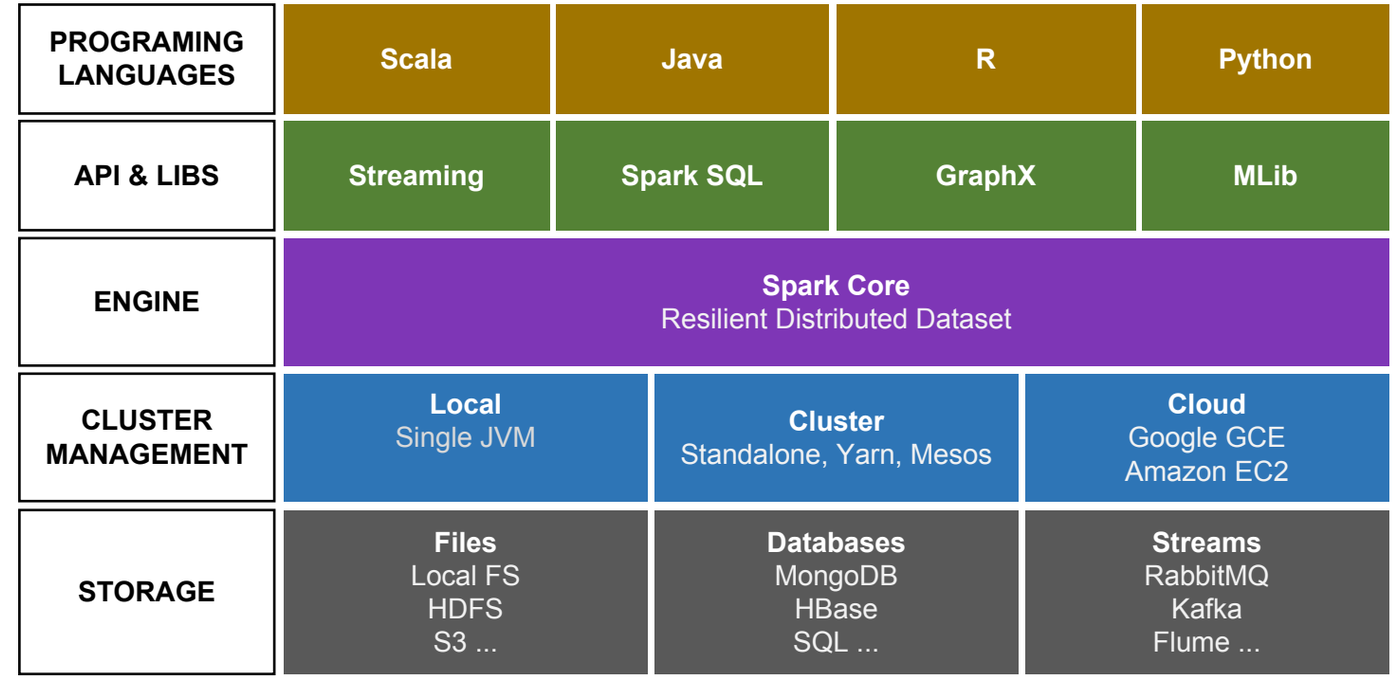
Spark Ecosystem

1. Spark Core

Spark Core is the base engine for large-scale parallel and distributed data processing. It is responsible for memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster & interacting with storage systems.

2. Spark Streaming

Spark Streaming is the component of Spark which is used to process real-time streaming data. Thus, it is a useful addition to the core Spark API. It enables high-throughput and fault-tolerant stream processing of live data streams.



3. Spark SQL

Spark SQL is a new module in Spark which integrates relational processing with Spark's functional programming API. It supports querying data either via SQL or via the Hive Query Language. For those of you familiar with RDBMS, Spark SQL will be an easy transition from your earlier tools where you can extend the boundaries of traditional relational data processing.

4. GraphX

GraphX is the Spark API for graphs and graph-parallel computation. Thus, it extends the Spark RDD with a Resilient Distributed Property Graph. At a high-level, GraphX extends the Spark RDD abstraction by introducing the Resilient Distributed Property Graph (a directed multigraph with properties attached to each vertex and edge).

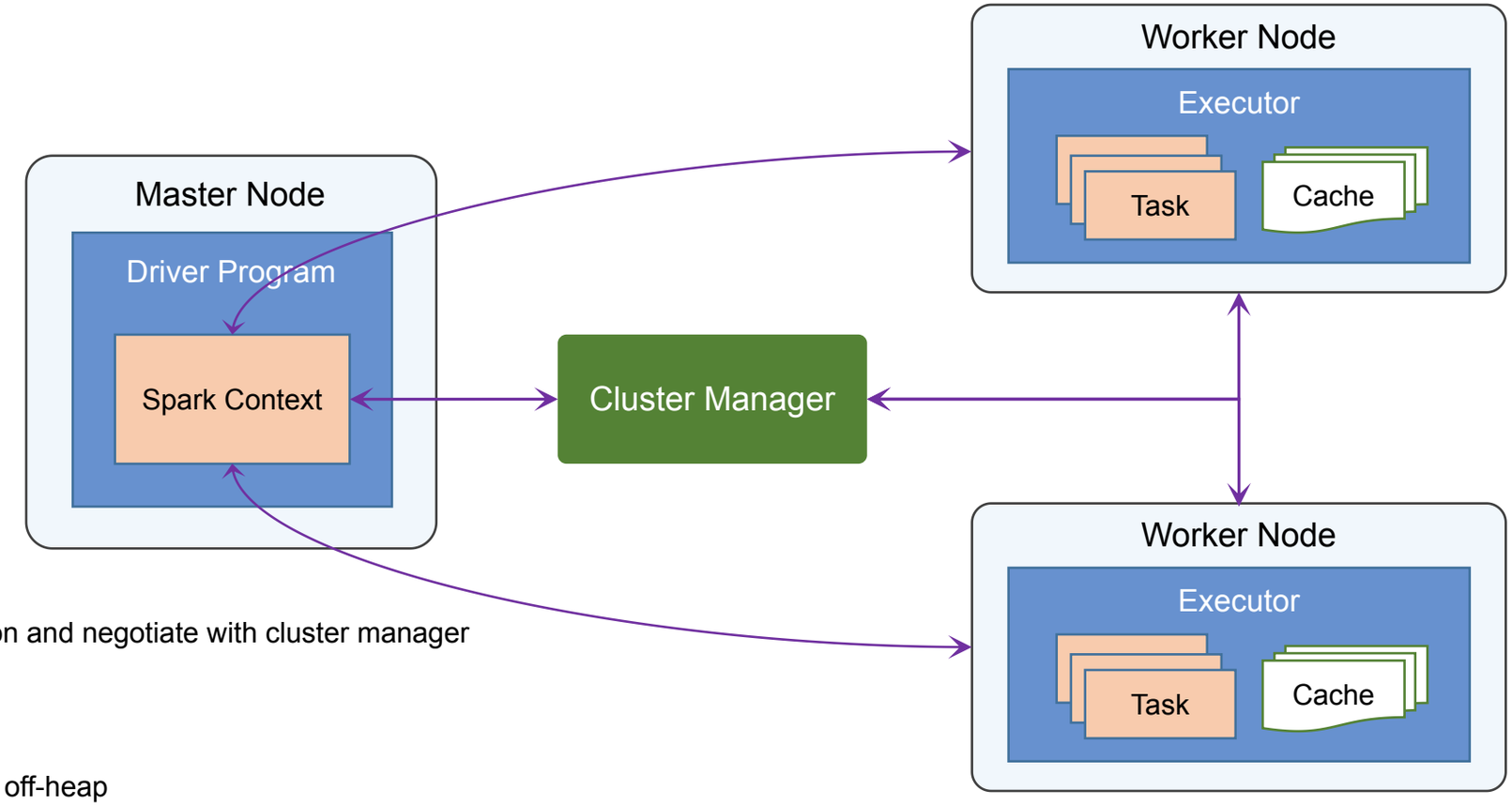
5. MLlib (Machine Learning)

MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

Glossary

Term	Meaning
Application	User program built on Spark. Consists of a driver program and executors on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext.
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN).
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster.
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor.
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called stages that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

Spark Architecture



1. Spark Driver

- Separate process to execute user applications
- Creates SparkContext to schedule jobs execution and negotiate with cluster manager

2. Executors

- Run tasks scheduled by driver
- Store computation results in memory, on disk or off-heap
- Interact with storage systems

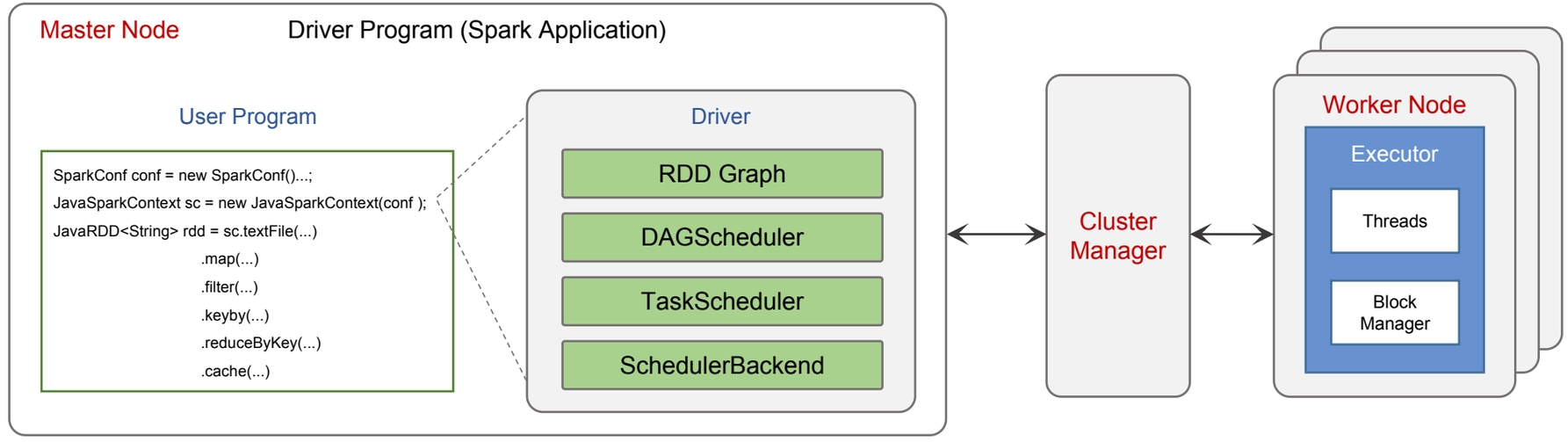
3. Cluster Manager

- Mesos
- YARN
- Spark Standalone

4. SparkContext

- Represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster

Spark Architecture



5. DAGScheduler

- Computes a DAG of stages for each job and submits them to TaskScheduler
- Determines preferred locations for tasks (based on cache status or shuffle files locations) and finds minimum schedule to run the jobs

6. TaskScheduler

- Responsible for sending tasks to the cluster, running them, retrying if there are failures, and mitigating stragglers

7. SchedulerBackend

- Backend interface for scheduling systems that allows plugging in different implementations(Mesos, YARN, Standalone, local)

8. BlockManager

- Provides interfaces for putting and retrieving blocks both locally and remotely into various stores (memory, disk, and off-heap)

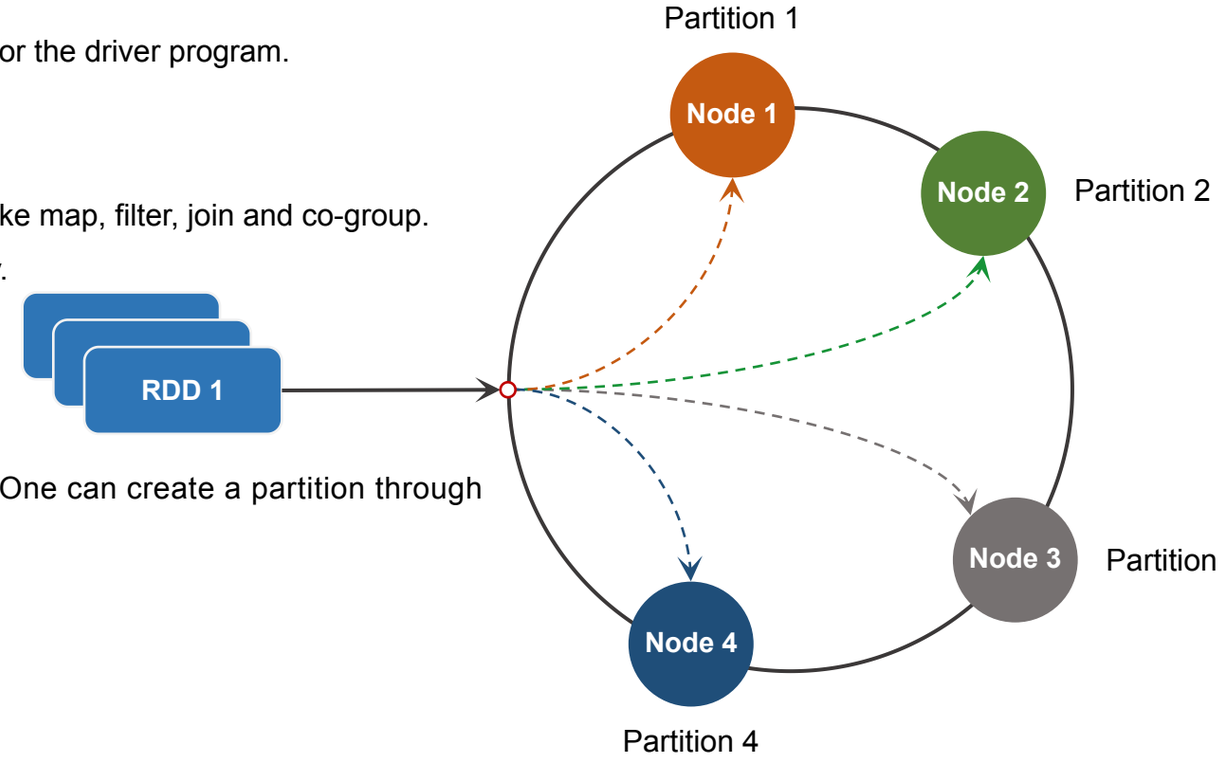
Resilient Distributed Dataset (RDD)

RDDs are the building blocks of any Spark application. **RDDs** Stands for:

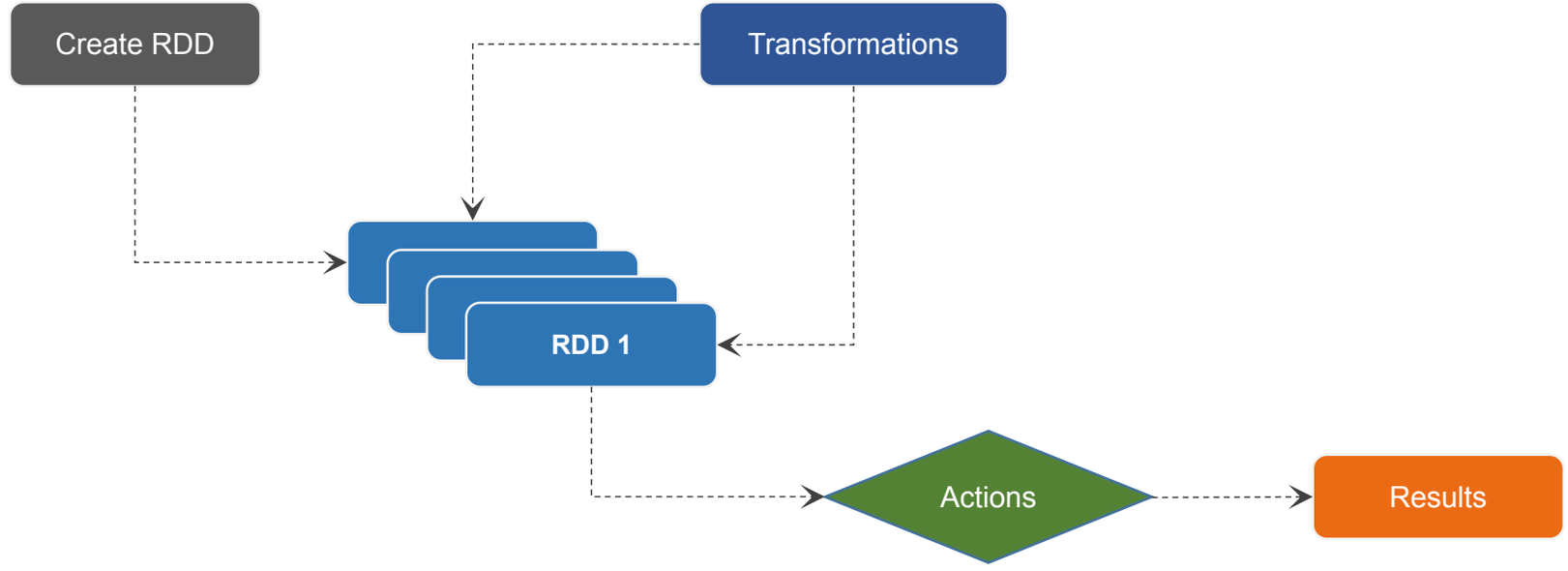
- **Resilient**: Fault tolerant and is capable of rebuilding data on failure
- **Distributed**: Distributed data among the multiple nodes in a cluster
- **Dataset**: Collection of partitioned data with values, e.g. tuples or other objects

RDD Features

1. **In-memory Computation**
   - RDDs stores intermediate data/results in RAM instead of disk.
2. **Lazy Evaluations**
   - All transformations are lazy, right away it will not compute the results.
   - Apache Spark computes transformations when an action requires a result for the driver program.
3. **Immutability**
   - Once RDD created it cannot be changed because of read only abstraction.
   - RDD can be transformed one form to another RDD using transformations like map, filter, join and co-group.
   - Immutable nature of RDD Spark helps to maintain level of high consistency.
4. **Partitioned**
   - Partitions are basic units of parallelism in Apache Spark
   - RDDs are collection of partitions.
   - Each partition is one logical division of data which is mutable in nature. One can create a partition through some transformations on existing partitions.
   - Partitions of an RDD are distributed across all the nodes in a network.
5. **Parallel**
6. **Persistence**
   - In RDD persistence nature does fast computations.
   - Here users have the option of selecting RDD for reuse and they can also select storage either disk or in-memory to store data.
7. **Fault Tolerance**
   - Spark RRD has the capability to operate and recover loss after a failure occurs.
   - It rebuild lost data on failure using lineage, each RDD remembers how it was created from other datasets to recreate itself.



RDD Workflow



There are two ways to create RDDs − parallelizing an existing collection in your driver program, or by referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, etc.

RDD Operations

RDDs support two types of operations:

- **Transformations**: are the functions that take an RDD as an input and produce one or more RDDs as an output.
- **Actions**: which returns final result to the driver program after running RDD computations on the dataset.

E.g:

map is a transformation that passes each dataset element through a function and returns a new RDD representing the results.

reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset).

1. Transformations

- Apply user function to every element in a partition (or to the whole partition).
- Apply aggregation function to the whole dataset (groupBy, sortBy).
- Introduce dependencies between RDDs to form DAG.
- Provide functionality for repartitioning (repartition, partitionBy).

2. Actions

- Trigger job execution.
- Used to materialize computation results.

Directed Acyclic Graph (DAG)

Direct: Means which is directly connected from one node to another. This creates a sequence i.e. each node is in linkage from earlier to later in the appropriate sequence.

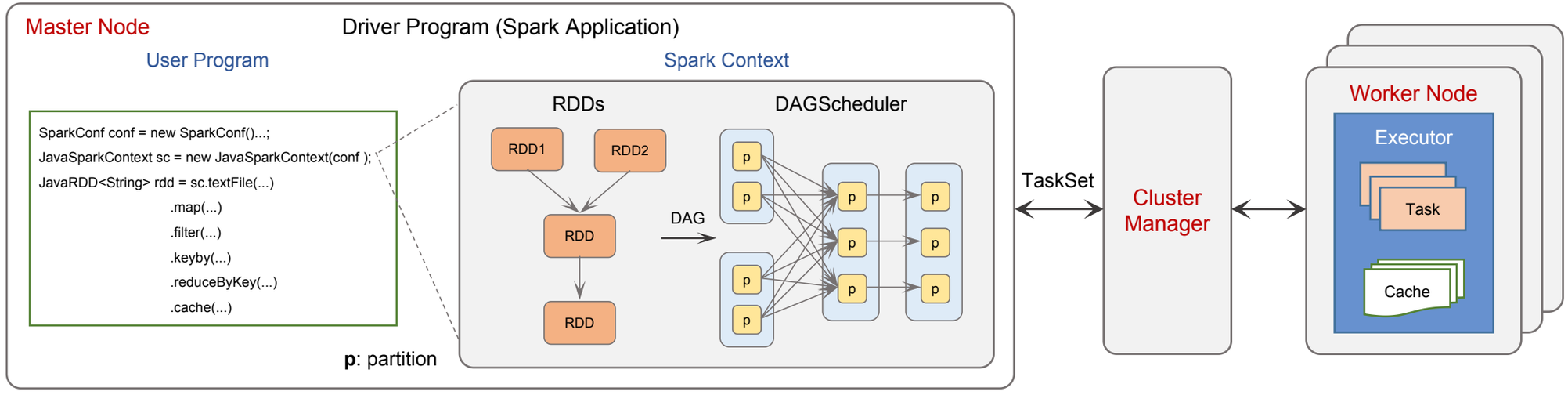
Acyclic: Defines that there is no cycle or loop available. Once a transformation takes place it cannot returns to its earlier position.

Graph: From graph theory, it is a combination of vertices and edges. Those pattern of connections together in a sequence is the graph.

Directed Acyclic Graph is an arrangement of edges and vertices. In this graph, vertices indicate RDDs and edges refer to the operations applied on the RDD. According to its name, it flows in one direction from earlier to later in the sequence. When we call an action, the created DAG is submitted to DAG Scheduler. That further divides the graph into the stages of the jobs.

- The DAG scheduler divides operators into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together. For e.g. Many map operators can be scheduled in a single stage. The final result of a DAG scheduler is a set of stages.
- The Stages are passed on to the Task Scheduler.The task scheduler launches tasks via cluster manager (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies of the stages.
- The Worker executes the tasks on the Slave.

Internal Job Execution In Spark



Build operator **DAG** and split graph into stages of task and submit each stage as ready to **TaskScheduler**. **TaskScheduler** then launches tasks via **Cluster Manager**.

STEP 1: The client submits spark user application code. When an application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically directed acyclic graph called DAG. At this stage, it also performs optimizations such as pipelining transformations.

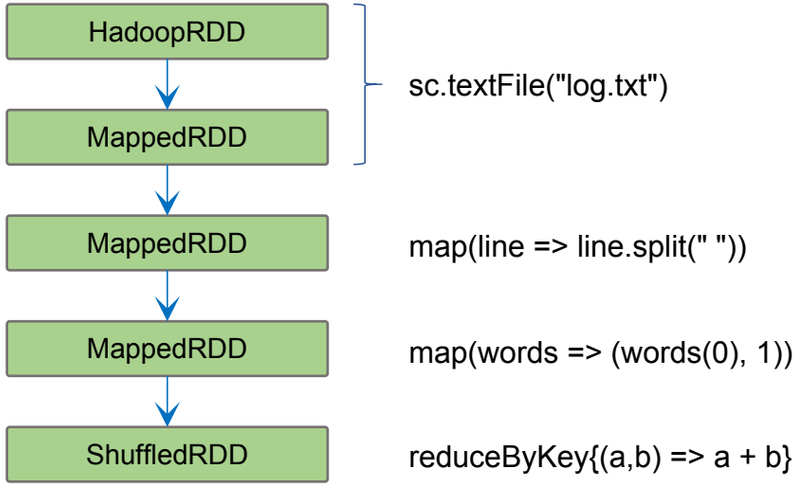
STEP 2: After that, it converts the logical graph called DAG into physical execution plan with many stages. After converting into a physical execution plan, it creates physical execution units called tasks under each stage. Then the tasks are bundled and sent to the cluster.

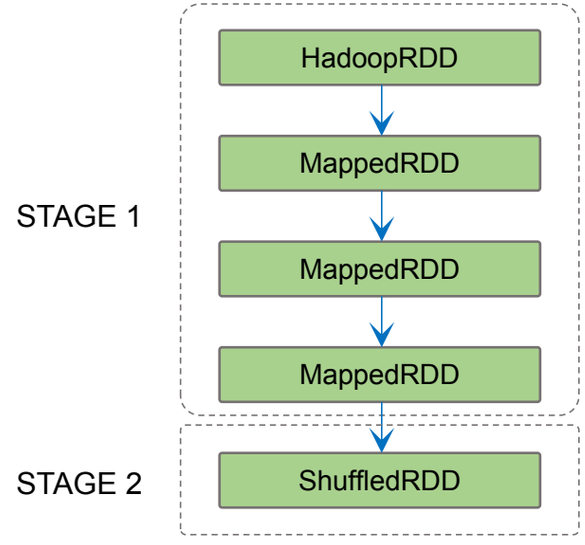
STEP 3: Now the driver talks to the cluster manager and negotiates the resources. Cluster manager launches executors in worker nodes on behalf of the driver. At this point, the driver will send the tasks to the executors based on data placement. When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing the task.

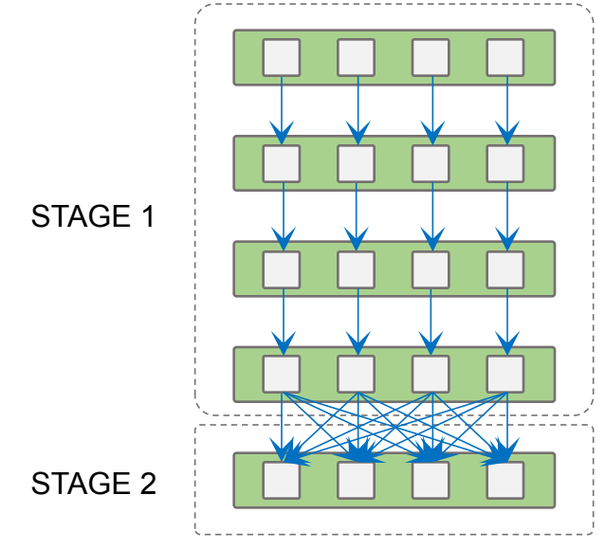
STEP 4: During the course of execution of tasks, driver program will monitor the set of executors that runs. Driver node also schedules future tasks based on data placement.

Job Execution In Spark Example

```
val input = sc.textFile("log.txt")
val splitedLines = input.map(line => line.split(" "))
                        .map(words => (words(0), 1))
                        .reduceByKey{(a,b) => a + b}
```







5. Data Processing Technologies Comparison

Apache Flink vs Spark vs Hadoop

	Apache Hadoop	Apache Spark	Apache Flink
Year of Origin	2005	2009	2009
Place of Origin	MapReduce (Google) Hadoop (Yahoo)	University of California, Berkeley	Technical University of Berlin
Data Processing Engine	Batch	Micro-Batch	Stream
Processing Speed	Slower than Spark and Flink	100x Faster than Hadoop	Faster than spark
Data Transfer	Batch	Micro-Batch	Pipelined and Batch
Programming Languages	Java, C, C++, Ruby, Groovy, Perl, Python	Java, Scala, Python and R	Java and Scala
Programming Model	MapReduce	Resilient Distributed Datasets (RDD)	Cyclic Dataflows
Memory Management	Disk Based	JVM Managed	Active Managed
Latency	Low	Medium	Low
Throughput	Medium	High	High
Optimization	Manual	Manual	Automatic
API	Low-level	High-level	High-level
Streaming Support	NA	Spark Streaming	Flink Streaming
SQL Support	Hive, Impala	SparkSQL	Table API and SQL
Graph Support	NA	GraphX	Gelly
Machine Learning Support	NA	SparkML	FlinkML

References

Source Code Example:

https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/index.html

https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch

https://ci.apache.org/projects/flink/flink-docs-stable/dev/batch/dataset_transformations.html

https://cwiki.apache.org/confluence/display/FLINK/Data+exchange+between+tasks

https://www.infoq.com/articles/machine-learning-techniques-predictive-maintenance

https://www.edureka.co/blog/spark-architecture

https://data-flair.training/blogs/spark-tutorial

https://data-flair.training/blogs/spark-in-memory-computing

https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97