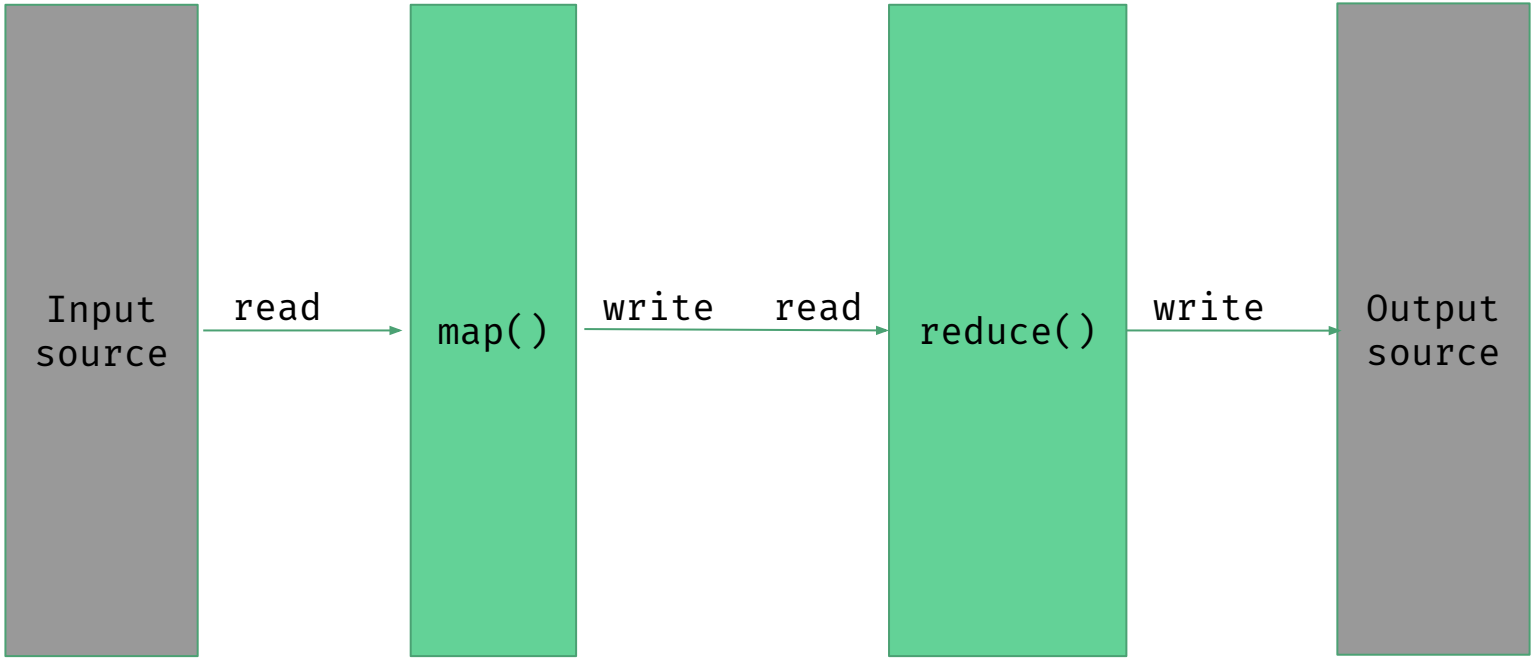
CS4830

Lab 3 - 11 Feb 2019

MapReduce


Input source
read
map()
write
read
reduce()
write
Output source

MapReduce model

- To reuse data, you need to store data to an external stable storage system, eg., HDFS
- Only applies to data whose computation requires a mapping and then a reduction
- Uses disk I/O, data replication and serialization
- Not built for interactive data analysis

Where does Spark fit in?

- *Iterative* computation
- Also supports MapReduce style
- *Interactive* data mining
- Load several datasets in memory and run ad-hoc queries across them
- Due to the support of iterative computation, can also run ML and graph processing algorithms

What is Spark?

- Cluster computing framework
- Allows implicit data parallelism and fault-tolerance
- Created at the UC Berkeley AMPLab and later donated to Apache Software Foundation
- Spark has two ***major*** abstractions, namely: ***RDDs*** and ***shared variables***
- RDDs or Resilient Distributed Datasets form the foundation of Spark
- Shared variables are a solution to reason about mutating global state or sharing data across nodes in a cluster

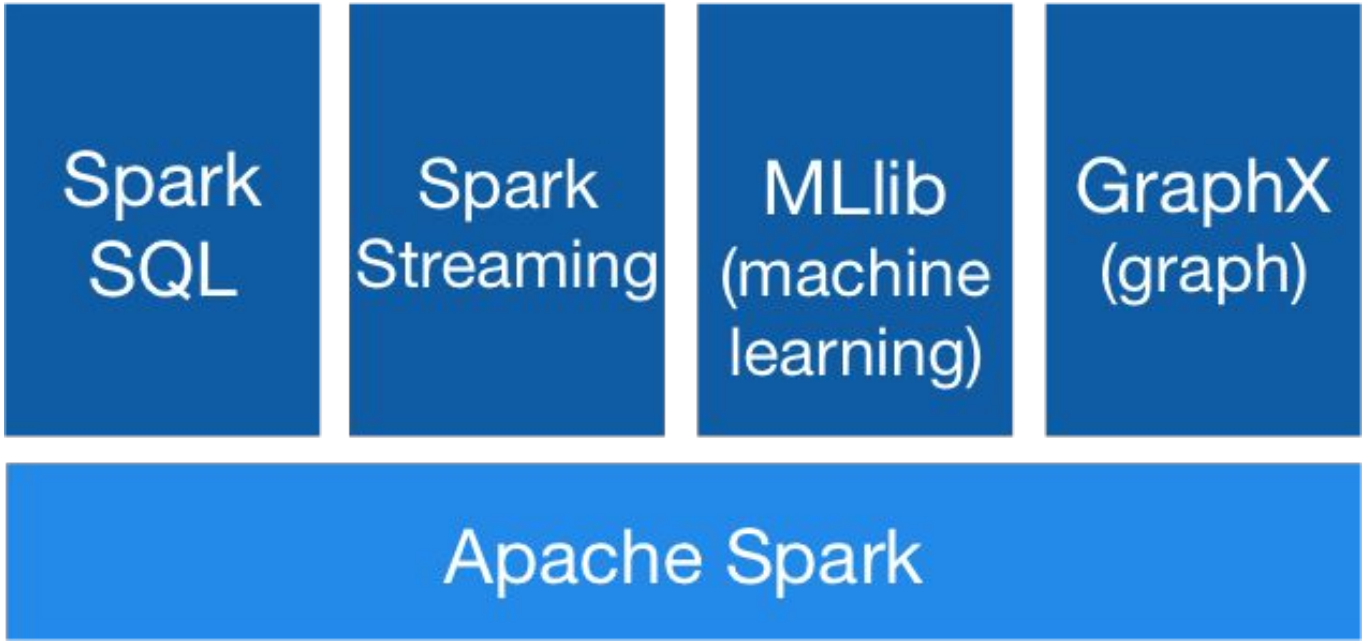
Resilient Distributed Datasets

- Developed from the motivation of being able to leverage distributed memory
- Systems like MapReduce take advantage of distributed storage and not memory
- Fault-tolerant abstraction for in-memory cluster computing
- Formally defined as a read-only, partitioned collection of records
- Can be created either by: ***loading data from stable storage*** or ***performing operations on other RDDs***
- Has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage
- Users can control: ***persistence*** and ***partitioning***

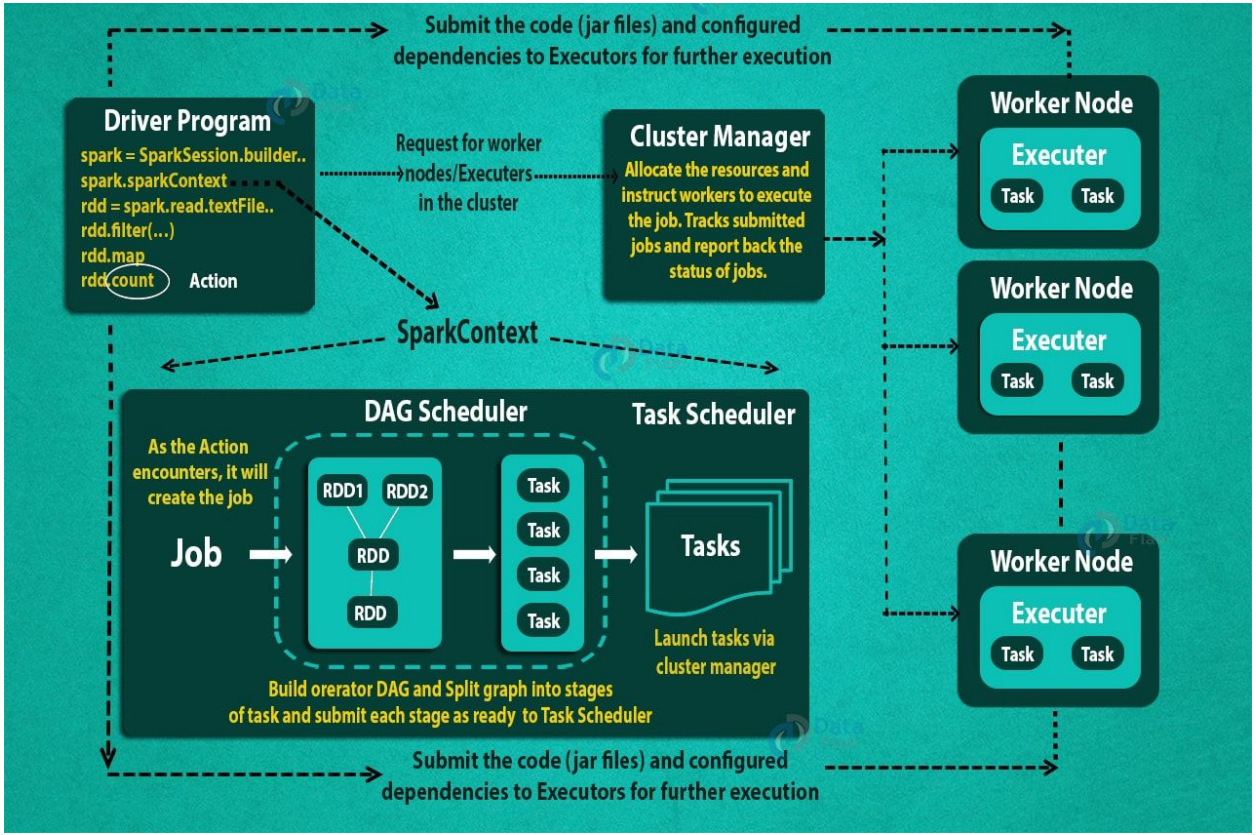
Shared variables

- Are of two types: ***Broadcast variables*** and ***Accumulators***
- Broadcast variable allows programmer to keep a read-only version of a data cached in every worker node
- For example, it doesn’t make sense to have a large dataset being shipped with every task for a computation
- Accumulators are variables which are “added” to through an associative and commutative operation
- For example, if you wanted to keep a counter and access it in a parallel fashion, you would use an accumulator

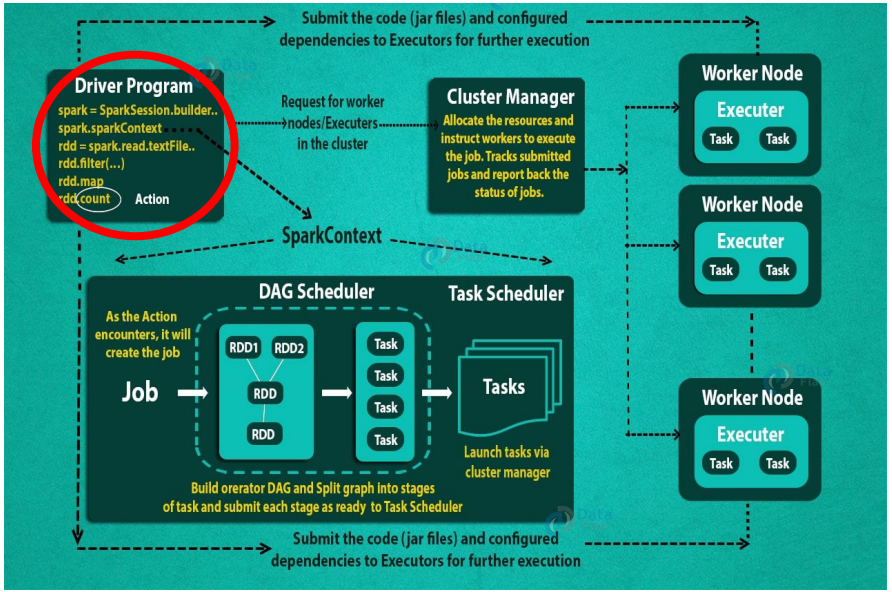
Spark Libraries



How it works?

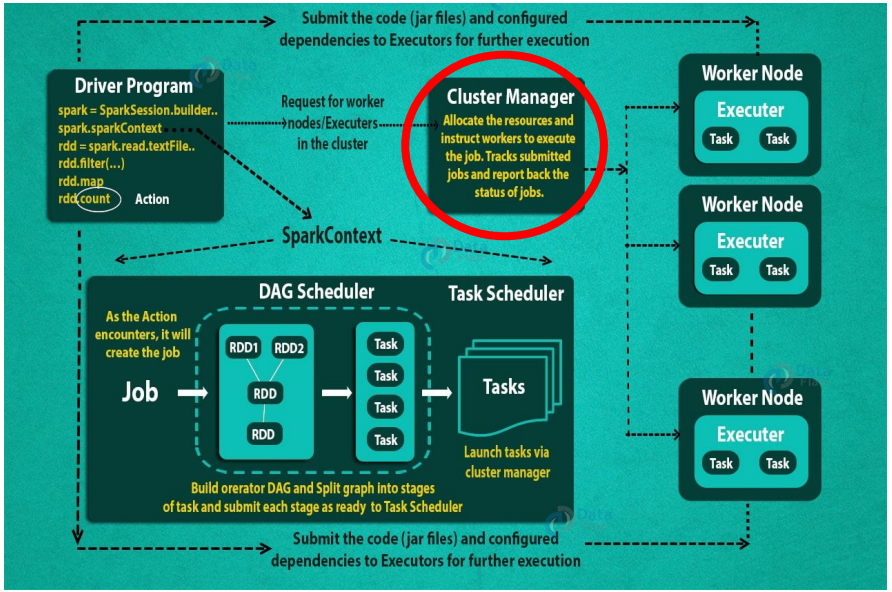

Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to Executors for further execution

Driver



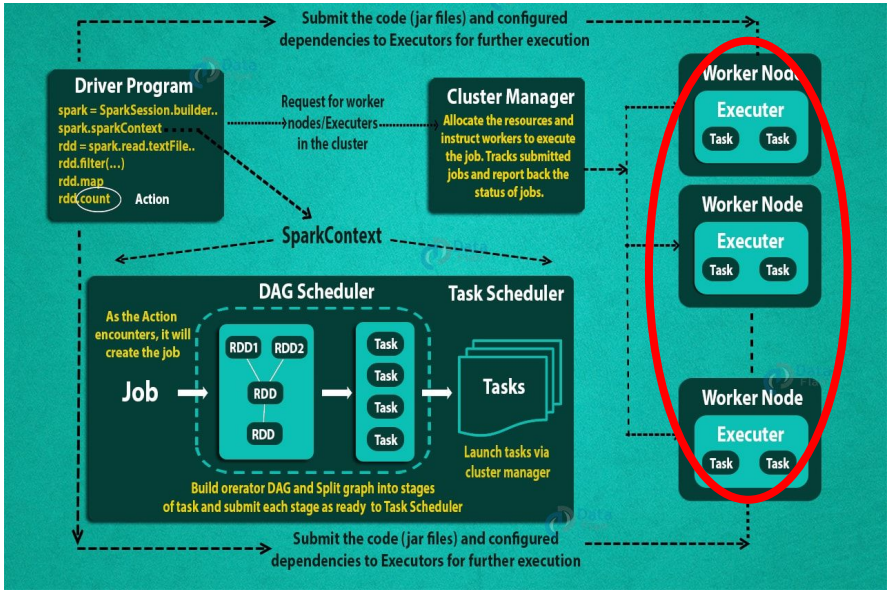
- It is the central coordinator in a Spark application.
- Creates RDDs, performs transformation and action, and also creates **SparkContext**.
- Communicate with a potentially large number of **executers**.
- Splits the application into **tasks** and **schedules** them to run on the **executer**.

Cluster Manager



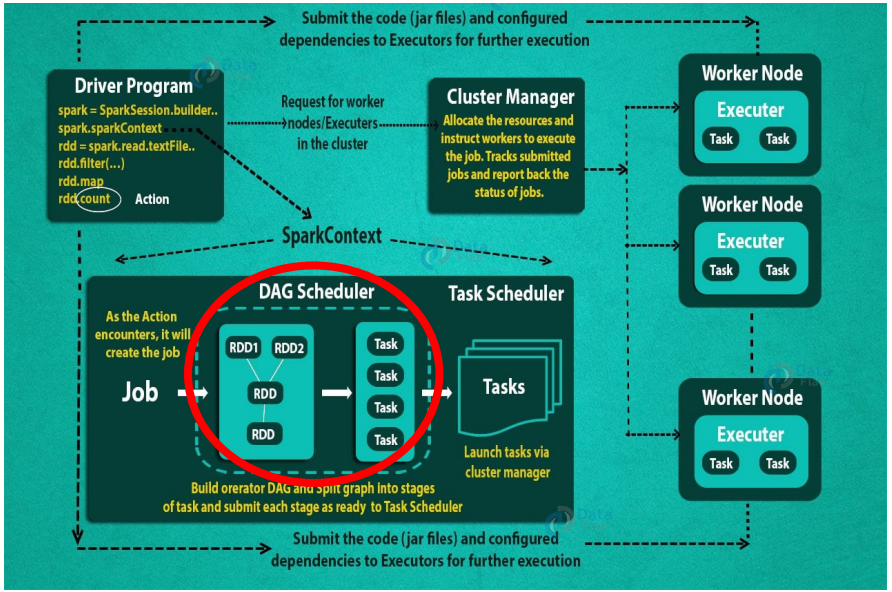
- Default: **Standalone Cluster Manager**. Alternatives: **Hadoop Yarn**, **Apache Mesos** etc.
- **Driver** program asks for the resources to the **cluster manager** that are needed to launch **executers**.
- Resources are allocated based on the workload.

Executer



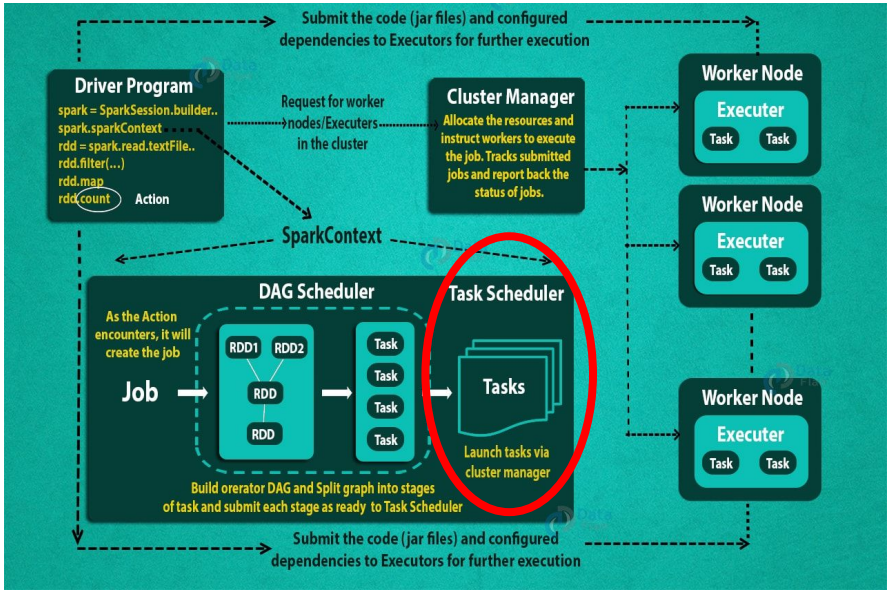
- Runs the **task** that makes up the application and returns the result to the **driver**.
- Provide in-memory storage for RDDs.
- With the help of **Heartbeat Sender Thread**, it sends metrics and heartbeats.
- Launched once in the beginning and run for the entire lifetime of an application.

DAG Scheduler



- **Job:** Parallel computation consisting of multiple **tasks** that get spawned in response to **actions**.
- **Directed Acyclic Graphs** (DAG): **Vertices** = RDD, **Edges** = Operations applied on RDDs
- DAGs are created implicitly. **Input** ⟶ **Transformation** ⟶ **Actions**
- DAG Scheduler: Splits the DAG into the **stages** of **tasks** (physical execution plan).

Task Scheduler



- **Task:** Unit of work sent to the **executer**.
- **Stage:** Each **stage** has some **task**, one task per partition. Classified as computational boundaries.
- The same **task** is done over different partitions of RDD.
- Task Scheduler: Distributes **tasks** among **workers**.

Anatomy of Spark Program

```
val paragraphs = sc.textFile("/path/to/file")
paragraphs.take(1)
val wordCount = paragraphs
    .flatMap(line => line.split(" "))
    .map(word => (word.stripSuffix(".").stripSuffix(","), 1))
    .reduceByKey((a, b) => a + b)
wordCount.collect()
```

- Code is in Scala (Spark is written in Scala)
- However there are APIs to Python, Java, R
- Reads as RDD
- org.apache.spark.rdd.RDD[String]

Data in Spark

- Spark Implementations of how data is stored
- Dataframe, dataset merged into Dataset<T> from Spark-2.0.

RDD

- Distributed collection of elements
- Immutable
- Low level operations

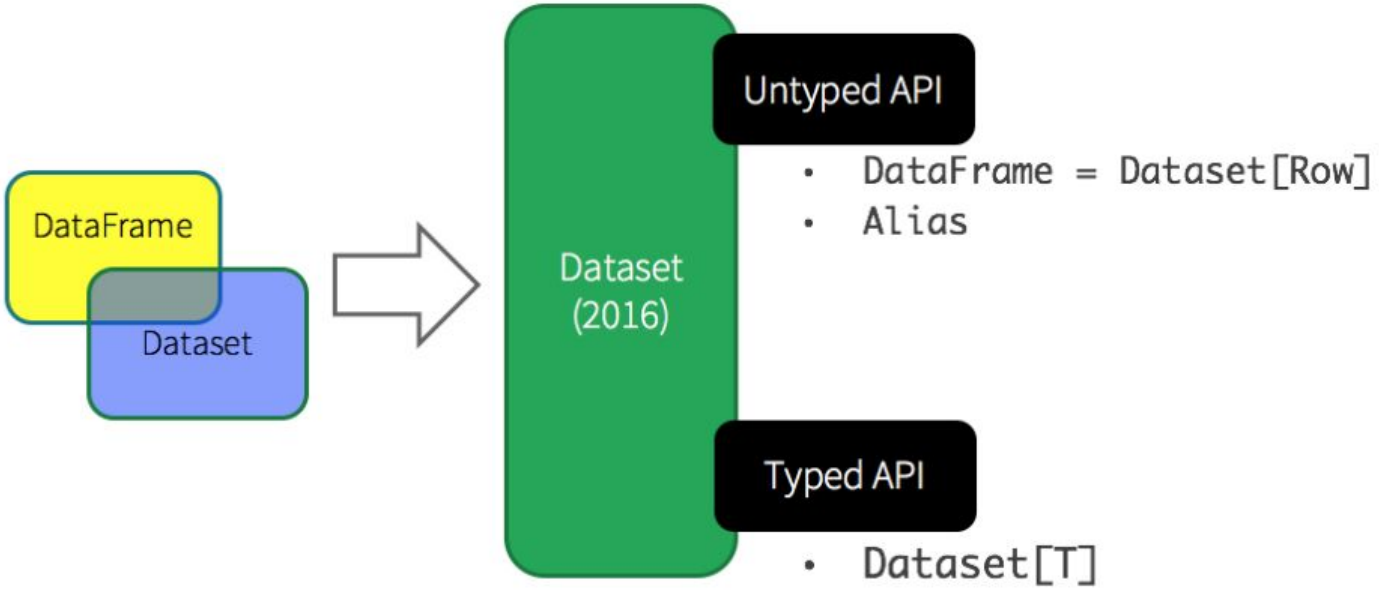
DataFrame

- Collection of elements organized as table
- Like Pandas/R DataFrame
- Not strongly typed

Dataset

- Organized as table
- Not present in R/Python
- Strongly typed
- In Spark 2.0

Unified Apache Spark 2.0 API



databricks

Anatomy of Spark Program

```
val paragraphs = sc.textFile("/path/to/file")
paragraphs.take(1)
val wordCount = paragraphs
    .flatMap(line => line.split(" "))
    .map(word => (word.stripSuffix(".").stripSuffix(","), 1))
    .reduceByKey((a, b) => a + b)
wordCount.collect()
```

- The variable “sc” is the SparkContext
- Every Zeppelin notebook starts with a SparkContext
- Establishes connection to Spark environment
- Setup internal variables

Anatomy of Spark Program

```
val paragraphs = sc.textFile("/path/to/file")
paragraphs.take(1)
val wordCount = paragraphs
    .flatMap(line => line.split(" "))
    .map(word => (word.stripSuffix(".").stripSuffix(","), 1))
    .reduceByKey((a, b) => a + b)
wordCount.collect()
```

- RDD has two operation
  - Transformations
  - Actions
- Actions include
  - Take
  - Count
  - FIrst
  - Collect

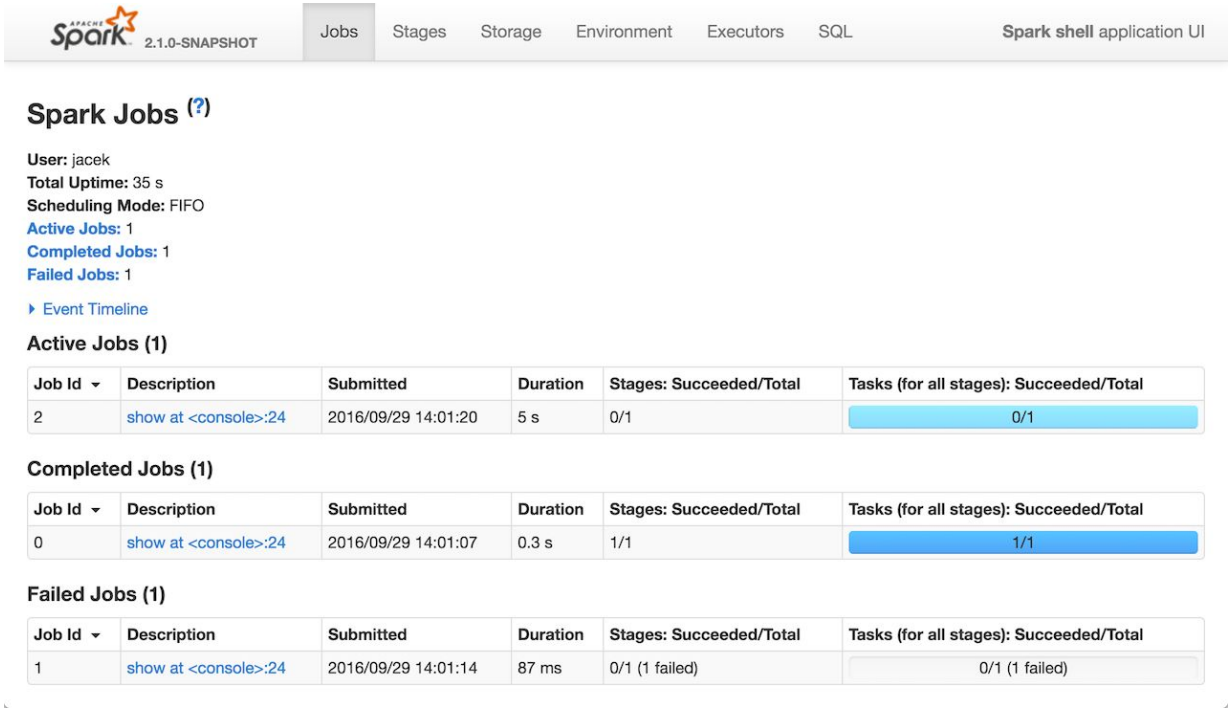
Anatomy of Spark Program

```
val paragraphs = sc.textFile("/path/to/file")
paragraphs.take(1)
val wordCount = paragraphs
   .flatMap(line => line.split(" "))
   .map(word => (word.stripSuffix(".").stripSuffix(","), 1))
   .reduceByKey((a, b) => a + b)
wordCount.collect()
```

- A set of transformations
- FlatMap - Maps every element to one or more element
- Map - Word -> (Word, 1)
- Reduce By Key - Groups keys and sums value

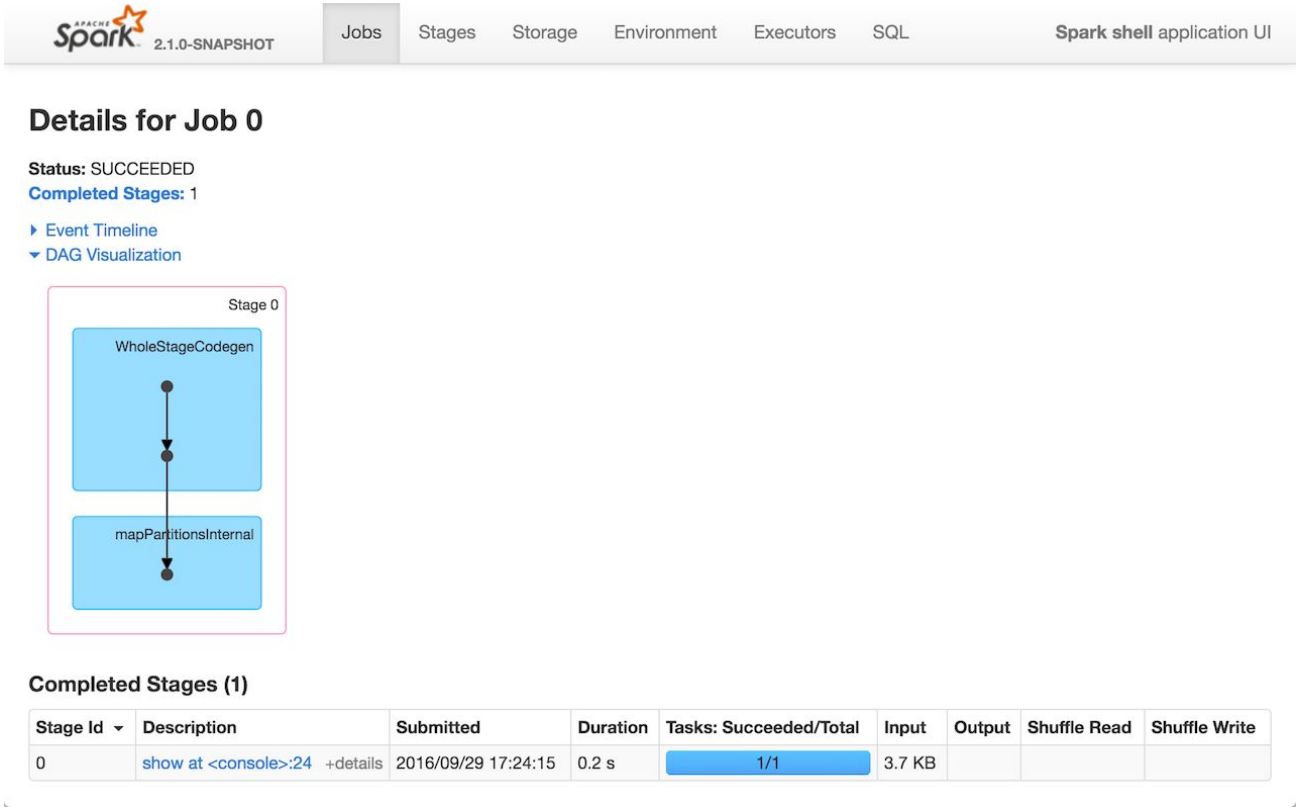
Spark - DAG Model

- Check IP:18081 to view spark UI
- View
  - active jobs
  - Tasks
  - Executor nodes



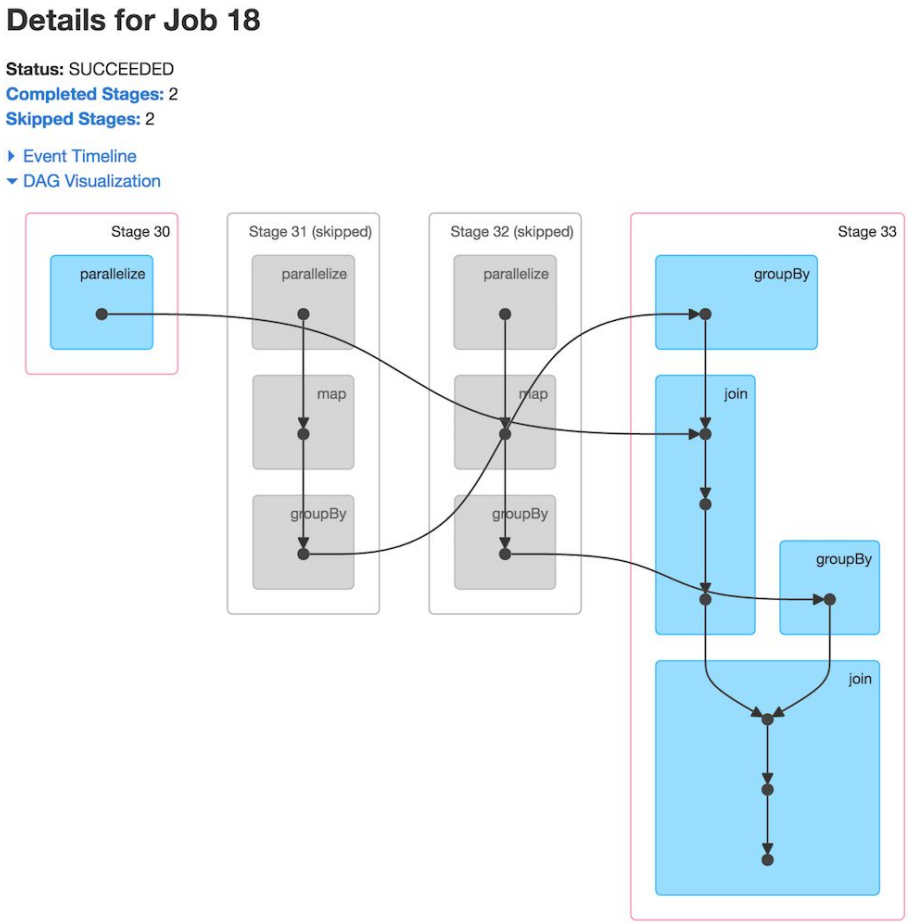
Spark - UI

- Check IP:18081 to view spark UI
- View
  - active jobs
  - Tasks
  - Executor nodes



Spark - DAG Model

- Word-count graph is simple
- in practice, they can be complicated



Anatomy of Spark Program

```
val paragraphs = sc.textFile("/path/to/file")
paragraphs.take(1)
val wordCount = paragraphs
   .flatMap(line => line.split(" "))
   .map(word => (word.stripSuffix(".").stripSuffix(","), 1))
   .reduceByKey((a, b) => a + b)
wordCount.collect()
```

- This is internally optimized in Spark (difference normal python code)
- 50 lines of code in MapReduce is 1 line in Spark

References

- https://spark.apache.org/docs/2.1.1/programming-guide.html
- https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-webui-jobs.html
- https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-architecture.html

Additional Material

- Internals of spark: https://jaceklaskowski.gitbooks.io/mastering-apache-spark/
- https://spark.apache.org/docs/2.1.0/quick-start.html
- https://cs.stanford.edu/~matei/papers/2012/nsdi_spark.pdf