One Bank
One UniCredit

Big Data and Apache Spark

Framework Introduction

Constantin Andrei Popa

Data & Analytics
Innovative E2E Applications

21.06.2017

Welcome to UniCredit

Content

- Introduction
- Spark Install
- Spark Modules
- Spark Terminology
- Data Models
- Deployment
- Monitoring Jobs
- Common mistakes

What is Apache Spark?

Introduction

- original author Andrei Zaharia at the University of California
- project later donated to and maintained by Apache
- open source general cluster-computing framework
- better performance compared to Hadoop's MapReduce framework
- written in Scala with support for Scala, Java, Python, R

https://spark.apache.org/
https://github.com/apache/spark

Introduction – Apache Hadoop's MapReduce Model

Input
Welcome to Hadoop
Class Hadoop is
good Hadoop is bad

Input Splits
Welcome to Hadoop
Class Hadoop is
good Hadoop is
bad

Mapping
Welcome ,1
to , 1
Hadoop , 1
Class , 1
Hadoop , 1
is , 1
good , 1
Hadoop , 1
is , 1
bad , 1

Shuffling
bad , 1
Class,1
good,1
Hadoop , 1
Hadoop , 1
Hadoop , 1
is , 1
is , 1
to , 1
Welcome ,1

Reducer
bad , 1
Class,1
good,1
Hadoop , 3
is , 2
to , 1
Welcome ,1

Final Output
bad 1
Class 1
good 1
Hadoop 3
is 2
to 1
Welcome 1

image credits guru99.com

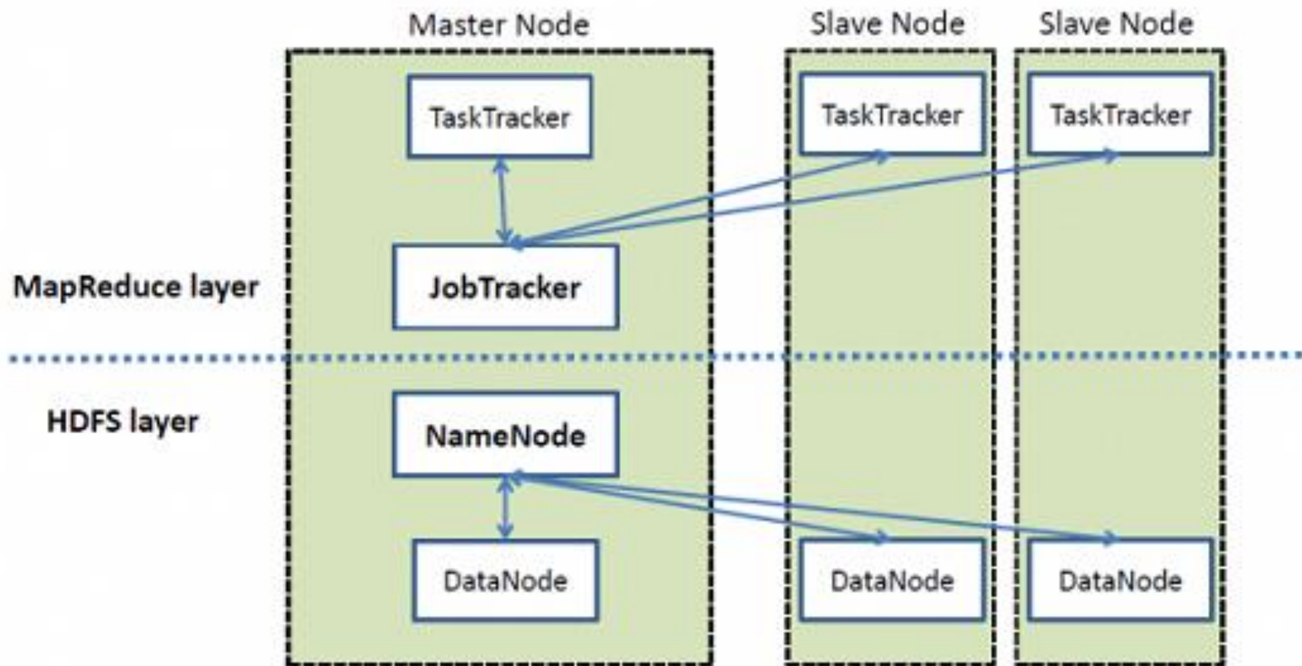
Introduction – Hadoop's MapReduce Model vs Spark

Input Data on Disk → t1 → RDD1 (in memory) → t2 → RDD2 (in disk) → t3 → RDD3 (in memory) → t4 → Output Data on Disk

Input Data on Disk → MR1 → Tuples (on Disk) → MR2 → Tuples (on Disk) → MR3 → Tuples (on Disk) → MR4 → Output Data on Disk

dattamsha.com

Introduction – Apache Hadoop Architecture



Introduction – Apache Hadoop Architecture

HDFS Architecture

Metadata ops

Namenode

Metadata (Name, replicas, …):
/home/foo/data, 3, …

Client

Block ops

Read

Datanodes

Datanodes

Replication

Blocks

Rack 1

Write

Rack 2

Client

Introduction – Apache Hadoop's MapReduce Model

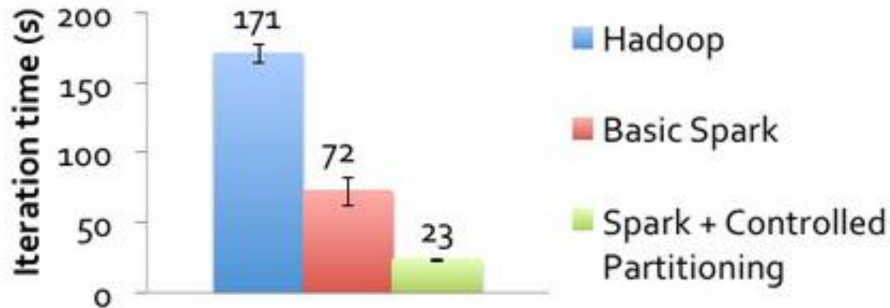
Advantages

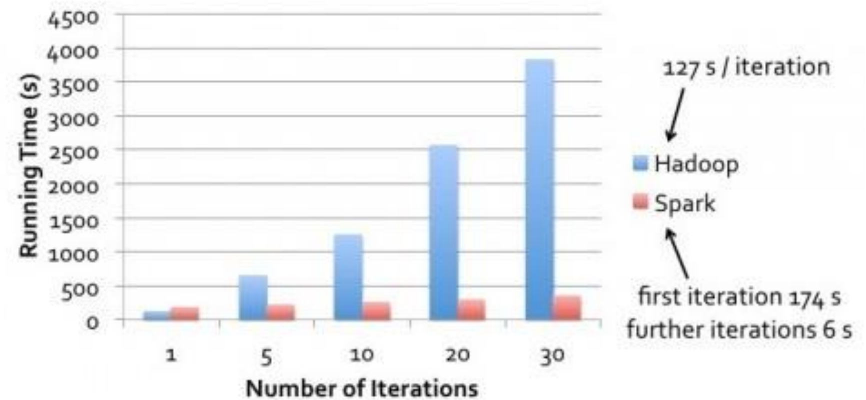
- simple model of programming
- scalable
- cost-effectiveness

Disadvantages

- simple model of programming – is not always easy to implement solutions as MapReduce
- jobs run in isolation
- result are not computed in real time
- usually more than one MapReduce jobs run in a sequence – writing intermediary steps to disk

Introduction – Apache Spark vs Apache Hadoop MapReduce





https://www.edureka.co/blog/apache-spark-vs-hadoop-mapreduce

Install

Spark Install

- Java 1.7 or higher
- Scala 2.10 or higher
- Scala Build Tool (SBT)
- download Spark from https://spark.apache.org/downloads.html
- check installation by opening spark-shell from spark_home/bin/spark-shell

- install Intellij Idea + Scala Plugin + Sbt Plugin
- set in build.sbt Spark dependencies

Modules

Apache Spark - Modules

- Spark Core
- Spark SQL
- Spark Streaming
- MLlib
- GraphX

Apache Spark - Modules

Spark Core Module

- provides task dispatching, scheduling and IO
- main abstraction RDD

Spark Streaming Module

- data is processed in mini-batches
- latency due to the mini batch duration

Spark SQL Module

- component on top of Spark Core
- main abstraction DataFrames
- support for structured and semi-structured data

GraphX Module

- distributed graph processing framework on top of Spark
- based on RDDs - not suited for update
- MapReduce style API

Apache Spark - Modules

Mllib

- distributed machine learning algorithms over Spark Core:
  - summary statistics. correlations. stratified sampling. …
  - linear models (SVMs, logistic regression, linear regression) decision trees. naive Bayes.
  - alternating least squares (ALS)
  - k-means.
  - singular value decomposition (SVD) principal component analysis (PCA)
  - stochastic gradient descent. limited-memory BFGS (L-BFGS)

Apache Spark - Terminology

- Driver program
- Cluster Manager
- Deploy Mode
- Worker Node
- Executor
- Task
- Job
- Stage
- SparkContext

Apache Spark – Application Flow

Spark driver

Job Job

Executor | Executor | Executor
Task Task | Task Task | Task Task

Apache Spark – Application Flow

Client Node
Driver JVM
Spark Context
Controls *spark.driver.memory* RAM

Worker Node 1
YARN Node Manager
HDFS Datanode
Node Memory Pool
(yarn.nodemanager.resource.memory-mb)

Executor JVM #1
Task #1
Requires *spark.task.cpus*
Task #2
Requires *spark.task.cpus*
…
Controls *spark.executor.cores CPU cores and spark.executor.memory RAM*

Executor JVM #2
Task #1
Requires *spark.task.cpus*
Task #2
Requires *spark.task.cpus*
…
Controls *spark.executor.cores CPU cores and spark.executor.memory RAM*

…

…

Worker Node N
YARN Node Manager
HDFS Datanode
Node Memory Pool
(yarn.nodemanager.resource.memory-mb)

Executor JVM #M-1
Task #1
Requires *spark.task.cpus*
Task #2
Requires *spark.task.cpus*
…
Controls *spark.executor.cores CPU cores and spark.executor.memory RAM*

Executor JVM #M
Task #1
Requires *spark.task.cpus*
Task #2
Requires *spark.task.cpus*
…
Controls *spark.executor.cores CPU cores and spark.executor.memory RAM*

…

Data Models

Apache Spark – Data Models

- RDDs
- DataFrame
- Dataset

Apache Spark – Resilient Distributed Dataset (RDD)

- basic abstraction of Spark Core
- immutable
- is a reference to an internal parallel collection or external data set such as HDFS files, Cassandra, Hbase
- they are considered resilient because in case of failure they can be re-computated

Types of operations

- **transformations**
- **actions**

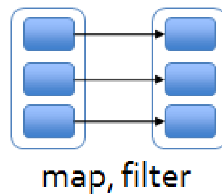
Apache Spark - Transformations

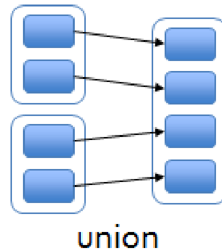
Transformations are lazy operations that create a new data set.

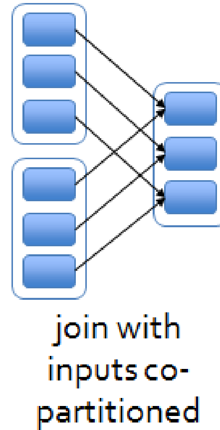
Narrow transformation - does not require shuffle of data across partitions.

Wide transformation - requires the data to be shuffled, for example records that need to be matched due to a join operation.

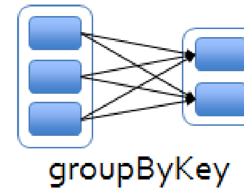
“Narrow” deps:

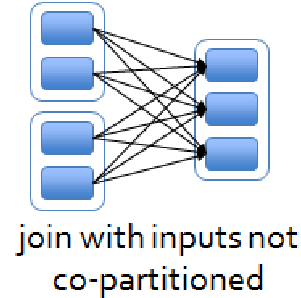






“Wide” (shuffle) deps:





Apache Spark – Transformations

map(func)	Return a new distributed dataset formed by passing each element of the source through a function func.
filter(func)	Return a new dataset formed by selecting those elements of the source on which funcreturns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so funcshould return a Seq rather than a single item).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
distinct([numTasks]))	Return a new dataset that contains the distinct elements of the source dataset.

Apache Spark – Transformations

groupByKey([numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks.
reduceByKey(func, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

Apache Spark - Actions

- return a value to the driver
- each action call forces the computation of an RDD.
- re-computations can be avoided when using persist.
- types of persist:

  MEMORY_ONLY
  MEMORY_AND_DISK
  MEMORY_ONLY_SER
  MEMORY_AND_DISK_SER
  DISK_ONLY
  MEMORY_ONLY_2

Apache Spark – Actions

reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(n)	Return an array with the first n elements of the dataset.
takeSample(withReplacement,num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.

Apache Spark – Actions

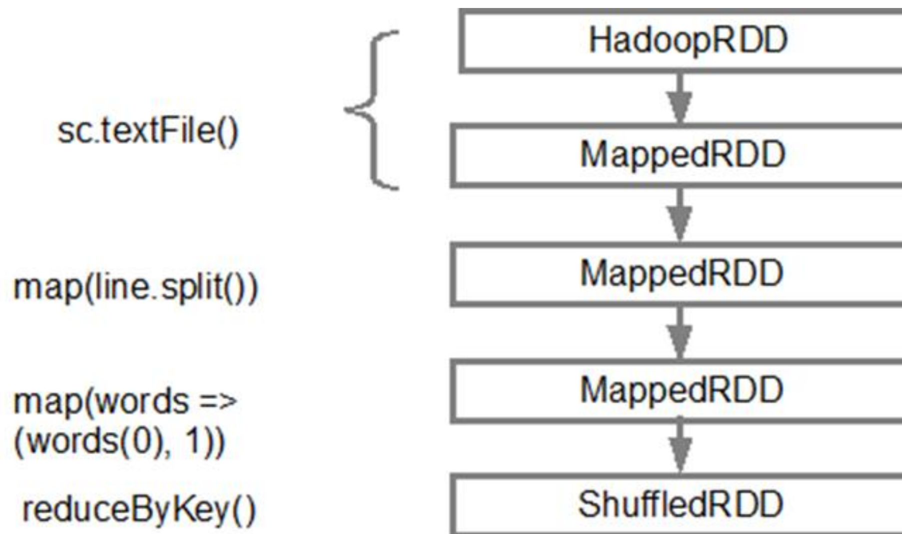
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(path) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).

Apache Spark - DAG

- Transformations and Actions define an application's **Direct Acyclic Graph (DAG).**
- using the DAG a physical execution plan is defined:
  - DAG Scheduler splits the DAG into multiple stages (stages are based on transformations, narrow transf. are piped together);
  - DAG Scheduler submits the stages to the Task Scheduler.

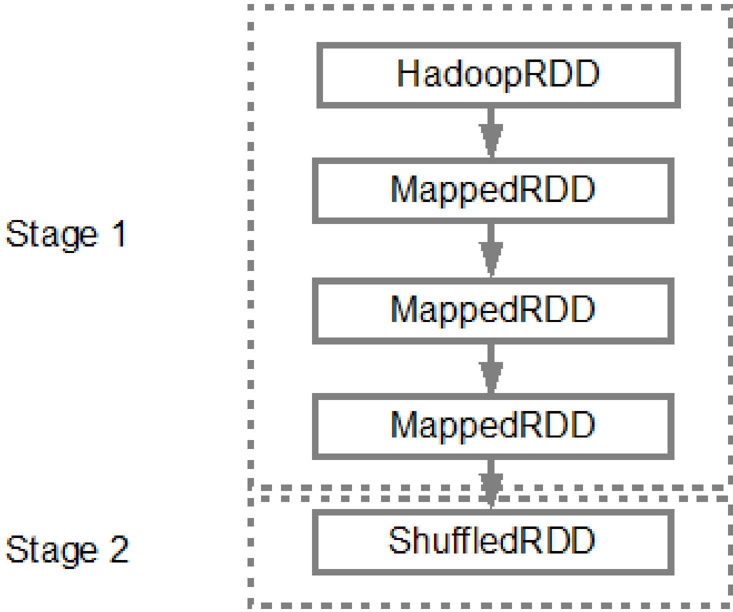
Apache Spark – DAG Example

Sequence of Transformations and Actions



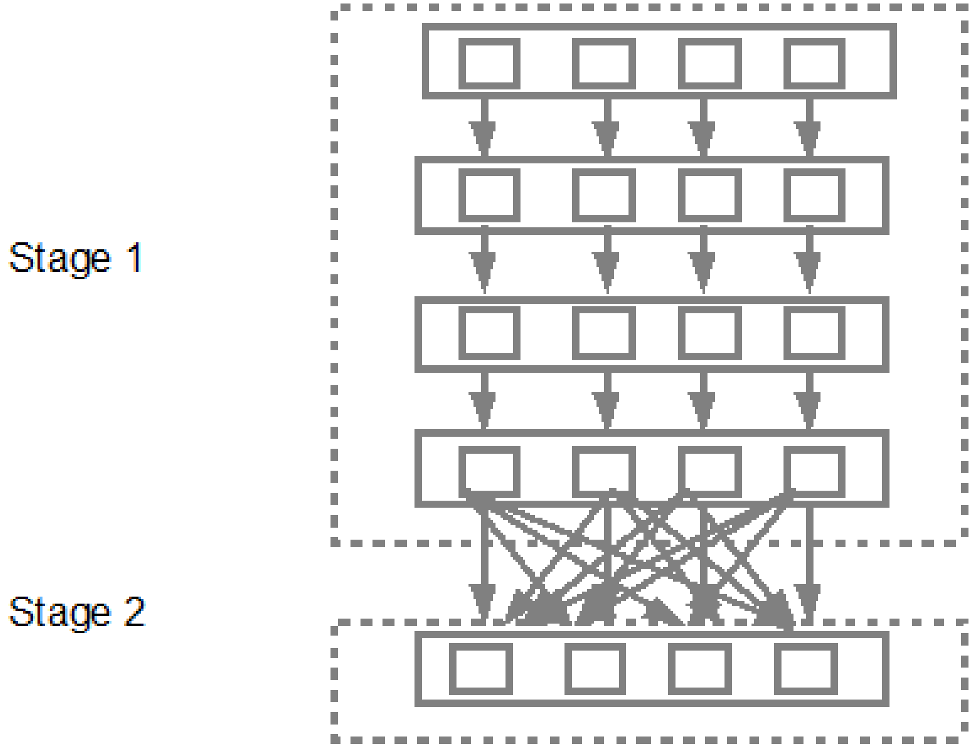
Apache Spark – DAG Example

Sequence of Stages



Apache Spark – DAG Example

Sequence of Stages/Tasks



Apache Spark – DataFrame, Datasets

- **Dataset**
  - distributed collection of data
  - strong typed
  - uses SQL Engine
  - use Encoder for optimizing filtering, sorting and hashing without de-serializing the object

- **DataFrame**
  - is a Dataset with named columns, Dataset[Rows]
  - equivalent of a relational database table
  - not strongly typed

- Dataset and DataFrame were introduced In Spark 1.6
  - DataFrame API as stable
  - Dataset API as experimental

- Spark 2.X – Dataset API became stable

Apache Spark – RDD vs Dataframe

- Dataframe
  - uses **Catalyst** optimizer on logical plan by pushing filtering and aggregations
  - uses **Tungsten** optimizer on physical plan by optimizing memory usage

- RDD
  - blackbox of data
  - plan cannot be optimized

Apache Spark – Catalyst

- Spark SQL query optimizer
- used to take the query plan and transform it into an execution plan
- transformations on RDD builds an a execution DAG
- transformations on Dataframe/Datasets Optimizations builds an optimal execution Tree
- **PushPredicateThroughJoin**:
    - If you first make a join between 2 dataframes and then filter the result using
    - rules that includes only one of them, the catalyst will change the plan and
    - will first filter the dataframe and after that will make the join
- **ColumnPruning**
    - attempts to eliminate the reading of unneeded columns from the query plan
- **CombineFilters**
    - if you make filter and then you filter again the result the catalyst will make
    - firstFilter AND secondFilter in 1 step
- **SimplifyFilters**
    - If the filter condition always is true, the filter is removed
    - If the filter always is false, replace input with empty relation

Apache Spark – Catalyst

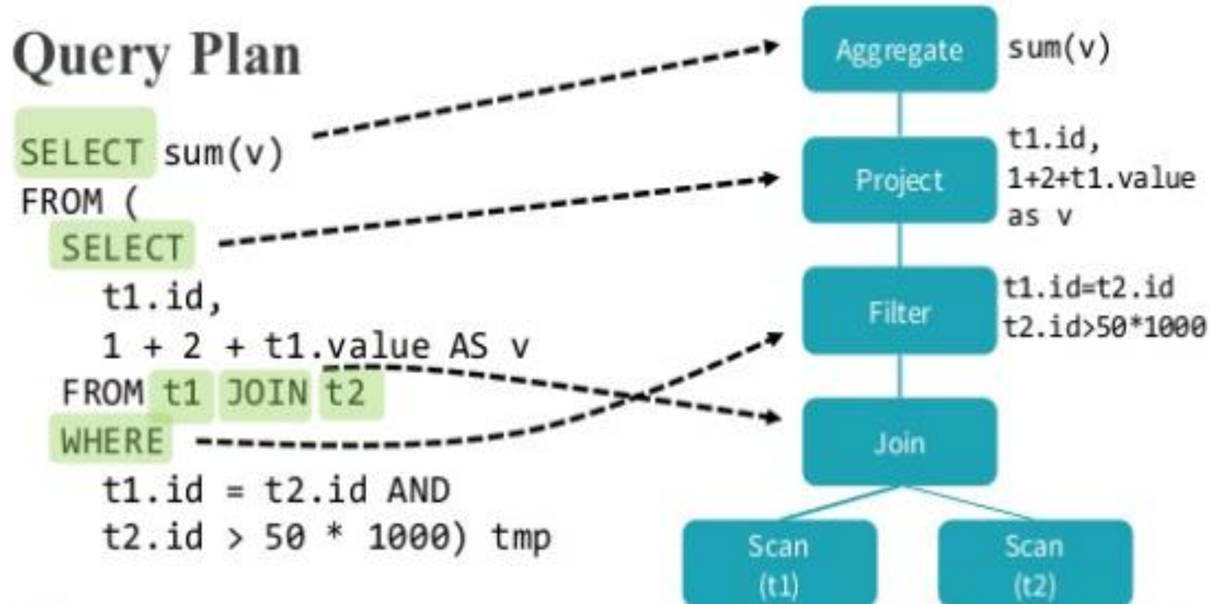


image credit: databricks

Deployment

Apache Spark - Deployment

- Standalone Deploy Mode
  - each node is defined in the Spark Configuration file

- Cloud deployment
  - Amazon EC2

- Hadoop Yarn

- Local Deployment
  is also available

Submit job via spark-submit command

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

47

Monitoring Jobs

Apache Spark – Monitoring Jobs Example

Spark Jobs (?)

Total Uptime: 12 min
Scheduling Mode: FIFO
Completed Jobs: 2

▸ Event Timeline

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <ipython-input-6-4615ba263c05>:1	2015/09/29 10:00:32	4 s	2/2	40/40
0	runJob at PythonRDD.scala:366	2015/09/29 10:00:27	4 s	1/1	1/1

Apache Spark – Monitoring Jobs Example

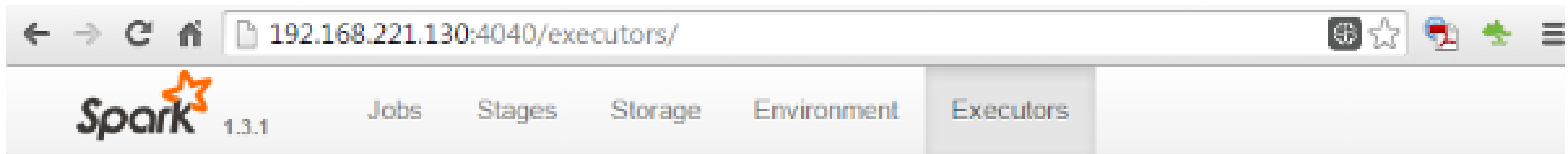
Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

Completed Stages (2)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	collect at <console>:26+details	2015/06/17 07:43:19	1.0 s	2/2			73.6 KB	
0	map at <console>:23+details	2015/06/17 07:43:17	2 s	2/2	209.8 KB			73.6 KB

Apache Spark – Monitoring Jobs Example

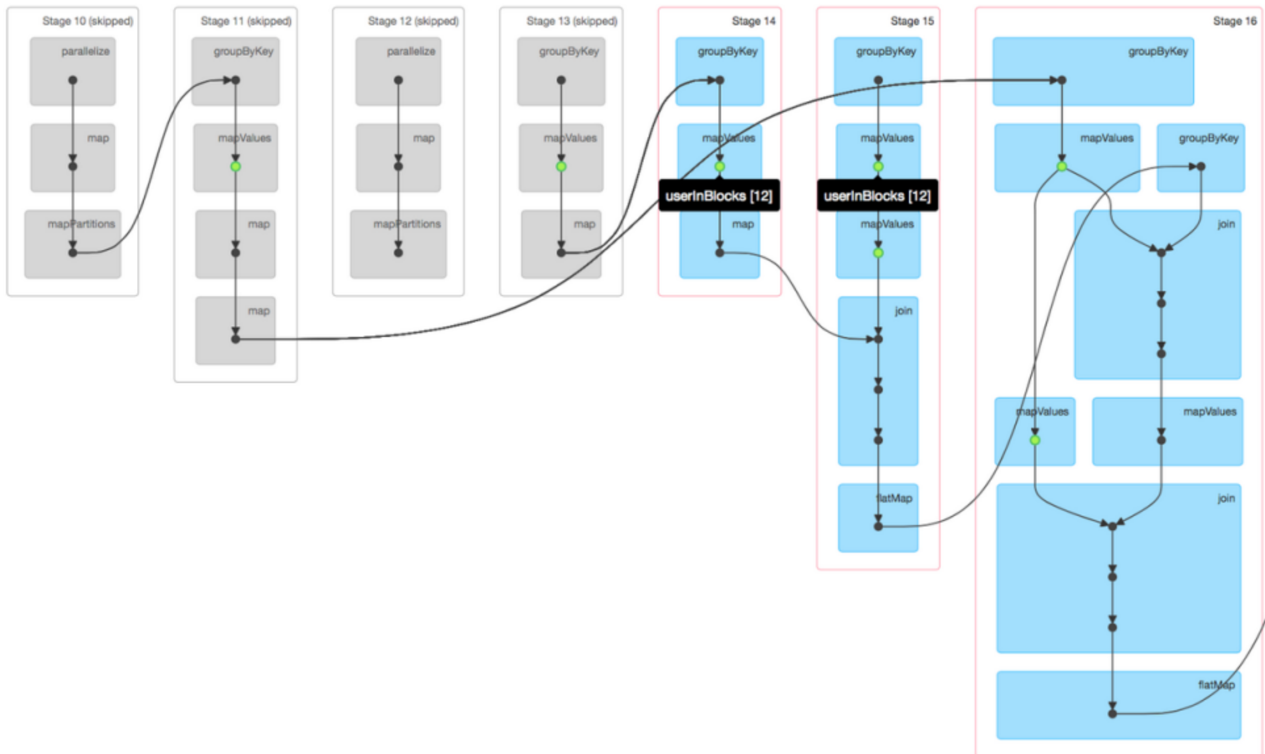


Executors (1)

Memory: 448.3 KB Used (246.0 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Thread Dump
<driver>	localhost:56501	2	448.3 KB / 246.0 MB	0.0 B	0	0	4	4	4.6 s	209.8 KB	0.0 B	73.6 KB	Thread Dump

Apache Spark – Monitoring Jobs Example



Common mistakes

Apache Spark - Mistakes

- Resource allocation and level of parallelization not explored/configured properly
- Intermediary data sets are not partitioned correctly – shuffle size problem
- Skew and Carthesian
- Try to avoid shuffles, use reduceByKey instead of groupByKey
- Use tree reduce instead of reduce to transfer load to the executors instead of the driver