Cloud Computing et informatique distribuée
Cours : Hadoop

Francesca Bugiotti

CentraleSupélec

May 14, 2020

Objectives

- List the main components of Hadoop ecosystem and their function
- Understand the usage of HDFS and MapReduce in Hadoop
- Explain Spark architecture
- Summarize how Spark manages and executes code on Clusters

Plan

Hadoop [2]

Apache Hadoop is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware

Characteristics

Hadoop components provide scalability to store and process large volumes of data on commodity hardware

Most of Hadoop components provide the possibility to recover from failures

It handles different data types

Hadoop

History

- 2004 - Google published a paper about their in-house processing framework called MapReduce [1]
- 2005 - Yahoo released an open-source implementation based on this framework called Hadoop

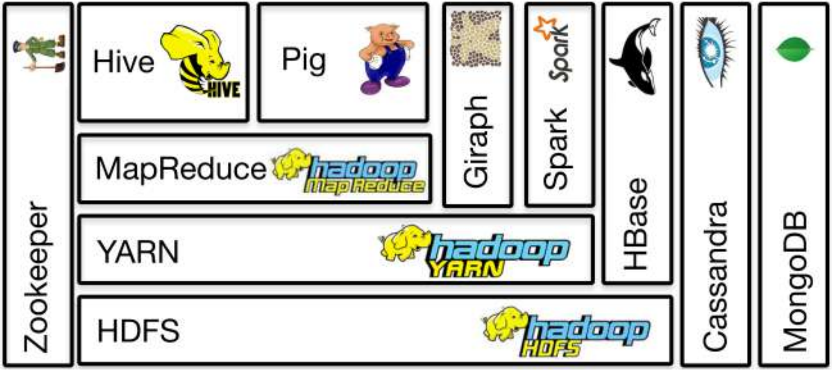
Now: more than 100 open-source projects on Hadoop

Hadoop

Many projects and components but on the core part we can identify:

- Hadoop Distributed File System (HDFS)
- Hadoop YARN
- Hadoop MapReduce

Hadoop


Zookeeper
Hive
HIVE
Pig
MapReduce
hadoop
MapReduce
YARN
hadoop
YARN
HDFS
hadoop
HDFS
Giraph
Spark
Spark
HBase
Cassandra
MongoDB

Hadoop

- HDFS: distributed file system on commodity hardware, scalable and reliable storage
- YARN: flexible scheduling and resource management over HDFS storage
- MapReduce: programming model for parallel computing
- Pig: augments data modeling of MapReduce with flow modeling
- Hive: augments data modeling of MapReduce with relational Algebra
- Giraph: processes of large scale graphs
- Storm, Spark, and Flink real time and in memory processing of Big Data
- NoSQL projects like Cassandra (created by Facebook) that also uses HBase
- Zookeeper: centralized management system for synchronization, configuration, and high availability

References I

Jeffrey Dean and Sanjay Ghemawat.
Mapreduce: Simplified data processing on large clusters.
In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150, 2004.

Tom White.
Hadoop: The Definitive Guide.
O'Reilly Media, Inc., 1st edition, 2009.

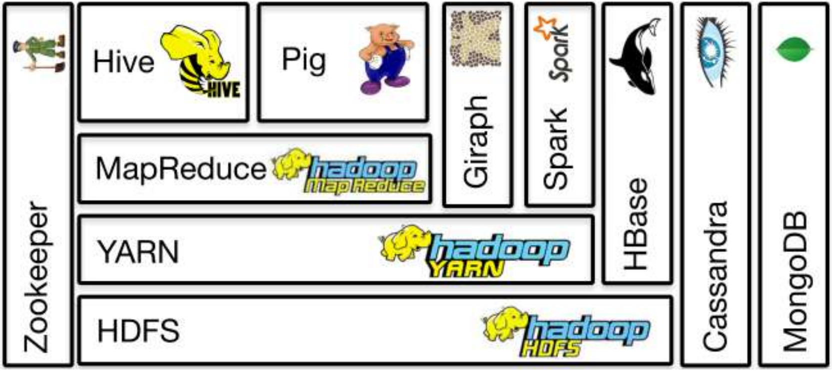
Cloud Computing et informatique distribuée
Cours : HDFS

Francesca Bugiotti

CentraleSupélec

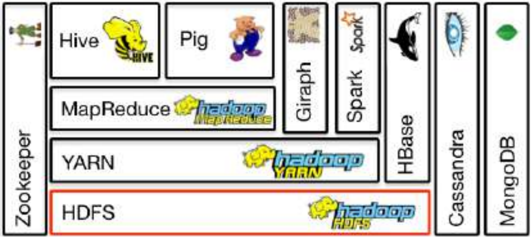
May 14, 2020

Hadoop


Zookeeper
Hive
HIVE
Pig
MapReduce
hadoop
Map Reduce
Giraph
Spark
Spark
HBase
Cassandra
MongoDB
YARN
hadoop
YARN
HDFS
hadoop
HDFS

Plan

HDFS



Hadoop Distributed File System

A storage layer for Big Data

- Scalability
- Reliability

HDFS

Some numbers

- Hortonworks: 200 petabytes
- Single cluster of 4.500 servers
- A billion of files and blocks

You do not run out of space:

- You add more nodes to increase the space

Scalability:

- Partition and splitting
  - Parallel access

HDFS

More numbers

- Gigabytes to terabyte
- Chunk size (the size of each piece) 64 megabytes

Fault tolerance

- Replica of file blocks
- By default 3 copies (configurable) -> replication factor

Many data types:

- Input file/output file format
- Flexibility:
  - Geospatial data can be read as vectors or rasters

HDFS

Two components

Name Node:

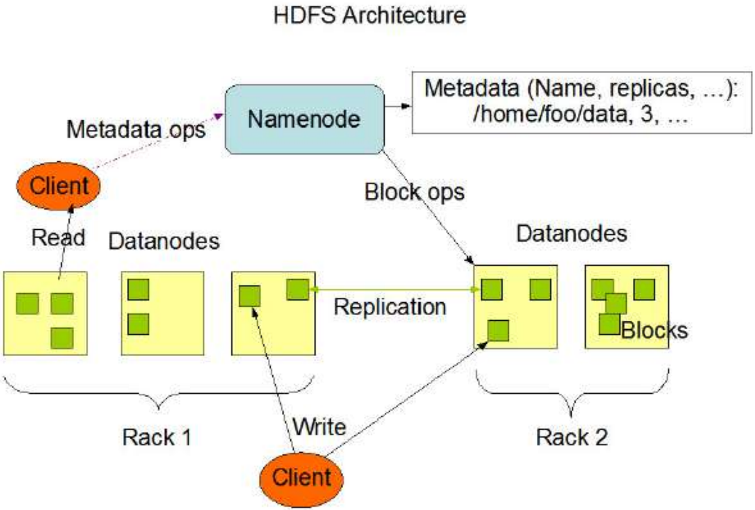
- Responsible for metadata
  - Name, location in the directory hierarchy, etc.
- Decides where to store data
- There is one in each cluster

Data Nodes:

- Block storage
- Does the block creation, deletion, and replication
- There is one in each node

Master/slave relationship

HDFS [?]


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

HDFS in practice

- Copy `myfile.txt` into HDFS:

```
hadoop fs -copyFromLocal myfile.txt
```

- Verify the existing files in HDFS:

```
hadoop fs -ls
```

- Delete a file:

```
hadoop fs -rm myfile.txt
```

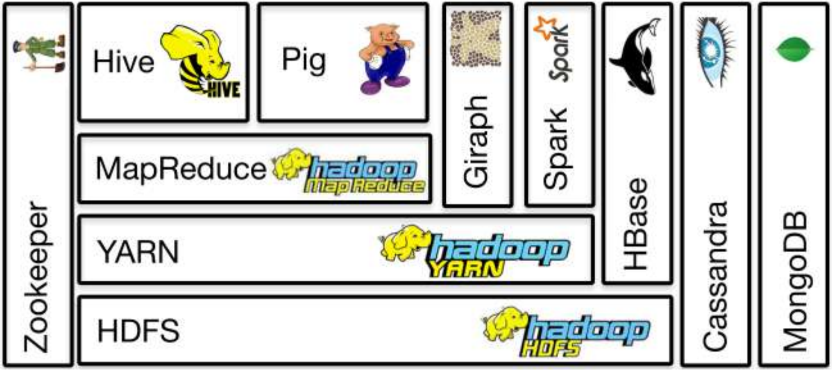
Cloud Computing et informatique distribuée
Cours : Yarn

Francesca Bugiotti

CentraleSupélec

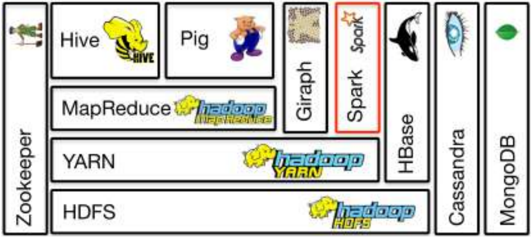
May 14, 2020

Hadoop


Hive
HIVE
Pig
Giraph
Spark
Spark
HBase
Cassandra
MongoDB
Zookeeper
MapReduce
hadoop
Map Reduce
YARN
hadoop
YARN
HDFS
hadoop
HDFS

Plan

YARN [1]



Hadoop Resource Manager

Interacts with applications and schedules resources for their usage

- Multiple applications
- Beyond MapReduce
- Beyond the data parallel programming model

YARN

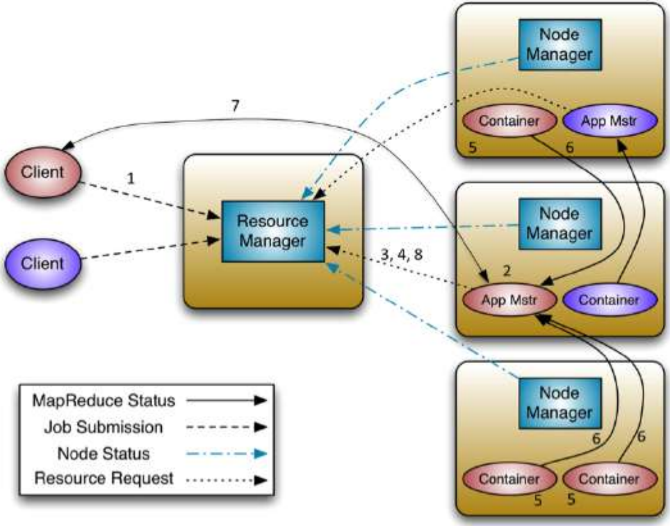
Added in Hadoop Ecosystem in a second moment

- Allows the support for not MapReduce applications
- Enables graph analytics
- Straming data

Main components:

- Resource Manager
- Node Manager
- Container

YARN


Node Manager
Container
App Mstr
5
6
7
Client
1
Resource Manager
Client
Node Manager
3, 4, 8
2
App Mstr
Container
Node Manager
6
6
Container
Container
5
5
MapReduce Status
Job Submission
Node Status
Resource Request

YARN

- The resource manager controls all the resources, and decides which one gets what
- Node manager operates at machine level and is in charge of a single machine

Application level:

- Each application gets an application master
  - Resources negotiation with the Resource Manager
  - Interaction with Node Manager to get the tasks completed
- Container
  - A collection of CPU memory disk network and other resources within the compute note

YARN

- Increases CPU utilization
- 1000 machines to their 2500 machines cluster
- Twice the number of jobs run before

Many distributed applications over the same Hadoop cluster

References I

Hadoop HDFS.
http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
Accessed 2016.

Cloud Computing et informatique distribuée
Cours : Hadoop-MapReduce

Francesca Bugiotti

CentraleSupélec

May 14, 2020

Plan

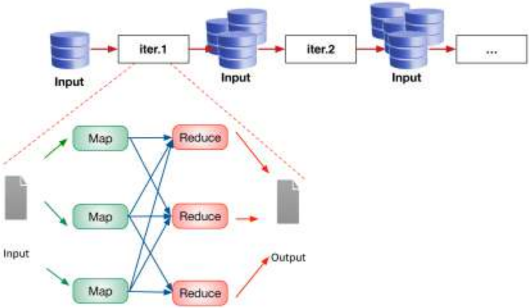
Hadoop MapReduce

With MapReduce you must create `Map` and `Reduce` tasks and run them

- MapReduce job usually splits the input data-set into independent chunks
  - the process works in parallel manner
- The framework sorts and shuffles the outputs of the maps
- The output of the map is given in input to the reduce tasks
- Both the input and the output of the job are stored in a file-system

The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

MapReduce



Input
iter.1
Input
iter.2
Input
...
Map
Map
Map
Reduce
Reduce
Reduce
Input
Output

MapReduce problems

- Programming model
  - Hard to implement everything as a MapReduce program
    - Iterative algorithms
    - Machine Learning, Graphs & Network Analysis
    - Interactive Data Mining
    - R, Excel-like computations, ad hoc reporting, etc.
  - Multiple MapReduce steps for simple operations
  - Lack of control structures and data types

MapReduce problems

- Efficiency
  - High communication cost
  - Frequent writing of output to disk
  - Limited exploitation of main memory
- Real-time processing
  - A MapReduce job requires to scan the entire input
  - Stream processing and random access impossible
- Programming language
  - Native support for Java only

Solutions?

Leverage to memory:

- Replace disks with SSD (Solid State Drive)
- Load data into memory

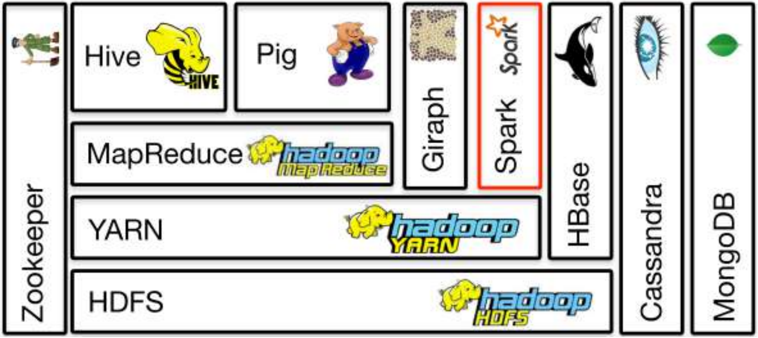
Cloud Computing et informatique distribuée
Cours : Hadoop-Spark

Francesca Bugiotti

CentraleSupélec

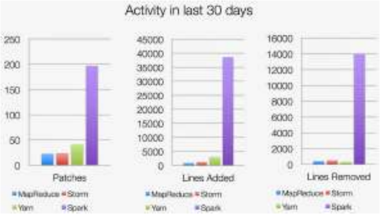
May 14, 2020

Plan


Zookeeper
Hive
HIVE
Pig
MapReduce
hadoop
Map Reduce
Giraph
Spark
Spark
HBase
Cassandra
MongoDB
YARN
hadoop
YARN
HDFS
hadoop
HDFS

History

- Spark [1, 2] project started in 2009
- Developed originally at UC Berkeley's AMPLab
- Open sourced in 2010
- Shark started summer 2011, alpha April 2012
- It is now the most popular project for big data analysis [3]



Spark

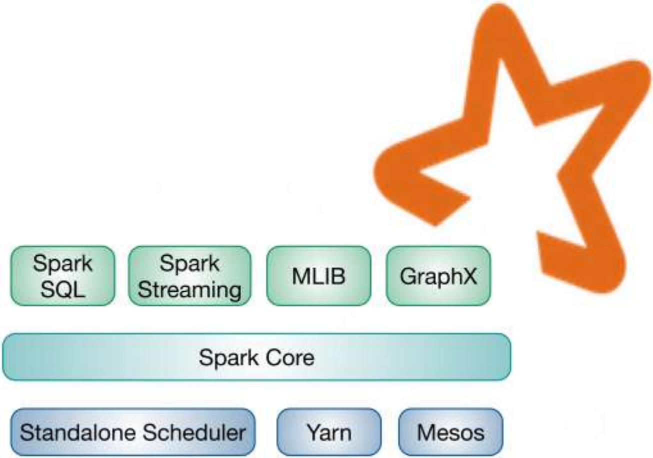
- Separate, fast, MapReduce-like engine
  - In-memory data storage for very fast iterative queries
  - General execution graphs and powerful optimizations
  - Up to 40x faster than Hadoop
- Compatible with Hadoop's storage APIs
  - Can run on top of an Hadoop cluster
  - Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc.
- Not a modified version of Hadoop

Spark

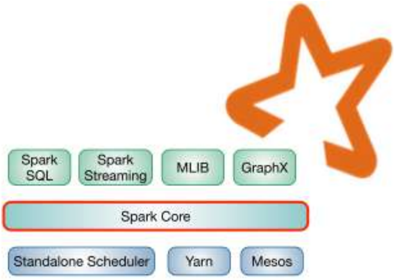
MapReduce greatly simplified big data analysis, but as soon as it got popular, users wanted more:

- Complex, multi-stage applications
  - Iterative graph algorithms and machine learning
  - RDD (Resilient Distributed Datasets) as unit of manipulation
- Efficiency
- Interactive ad-hoc queries
- Both multi-stage and interactive apps require faster data sharing across parallel jobs

Spark Stack

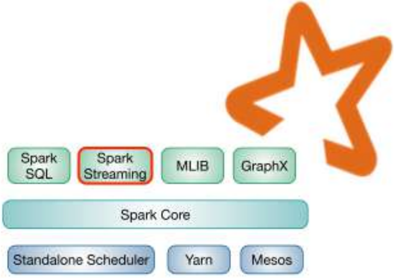

Spark SQL
Spark Streaming
MLIB
GraphX
Spark Core
Standalone Scheduler
Yarn
Mesos

Spark Core



- Memory management
- Distributed scheduling
- Fault tolerance
- API for distributed datasets

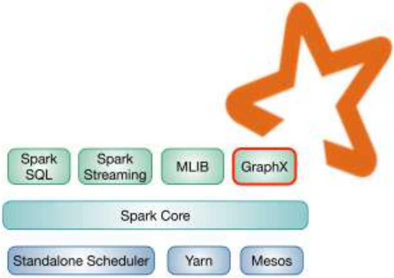
Spark Streaming



It implements an incremental stream processing using a model called “discretized streams”

- Input is split in small batches
- Regular combinations with states stored into RDD (Resilient Distributed Datasets)

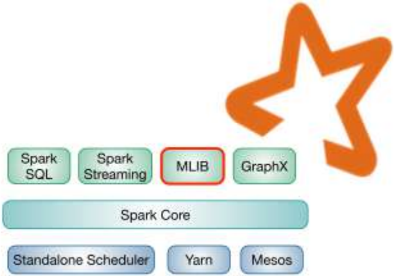
Spark GraphX



A graph computation interface that treats each graph as a directed multigraph with properties attached to each vertex and edge

- Variety of graph algorithms
- Flexible, fault tolerant, intuitive

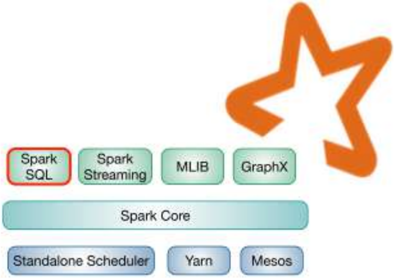
MLib



Machine learning library

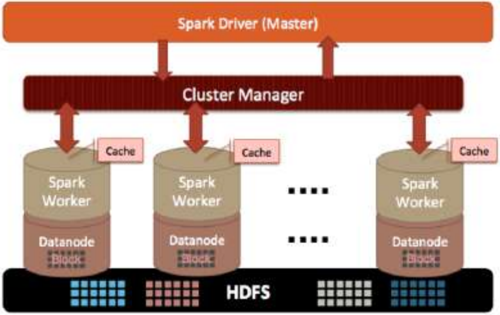
- More than 50 algorithms implemented

Spark SQL

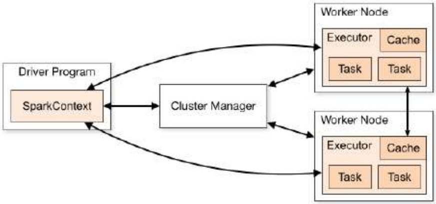


SQL queries on Spark with techniques similar to analytical databases

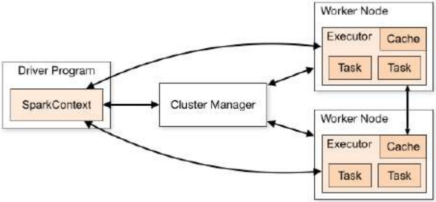
- Column storage
- Cost-based optimization
- Code generation for query execution


Spark Driver (Master)
Cluster Manager
Cache
Spark Worker
Datanode
Cache
Spark Worker
Datanode
Cache
Spark Worker
Datanode
HDFS

Cluster Architecture [4]

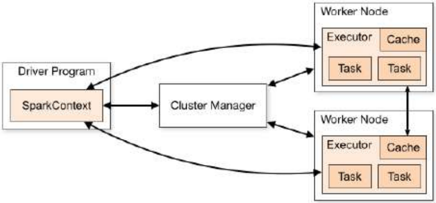


Cluster Architecture [4]



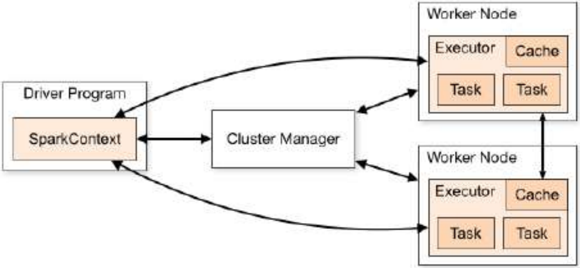
- Applications run as independent sets of processes on a cluster
- Applications are coordinated by the SparkContext object in the main program (the driver program)
- Each application gets its own executor processes
- The execution process stays up for the duration of the whole application and runs tasks in multiple threads

Cluster Architecture [4]



- To run on a cluster, the SparkContext can connect to several types of cluster managers, which allocate resources across applications
- Once connected, Spark acquires executors on nodes in the cluster, which run computations and store data
- Next, it sends the application code to the executors
- Finally, SparkContext sends tasks to the executors to run

Cluster Architecture [4]



Two main components:

- A driver program
- A set of worker nodes

Driver Program

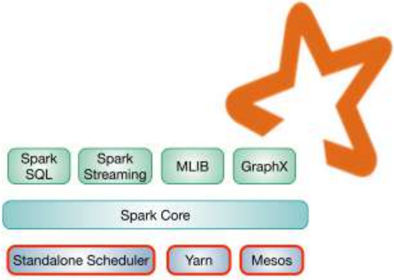
Where the application starts:

- Distributes RDDs on a computational cluster
- Creates a connection to a Spark cluster through a SparkContext object
- The default SparkContext in the Spark shell is an object (called SC for SparkContext)
- Manages a potentially large number of nodes called worker nodes

Worker Node

- Keeps a running Java virtual machine, called the executor
  - The core for application execution
  - Executors can execute tasks related to mapping stages or reducing stages or other Spark specific pipelines
- In a real Big Data scenario, we have many worker nodes running tasks internally
- Automatic provisioning and restarting of these nodes handled by the cluster manager

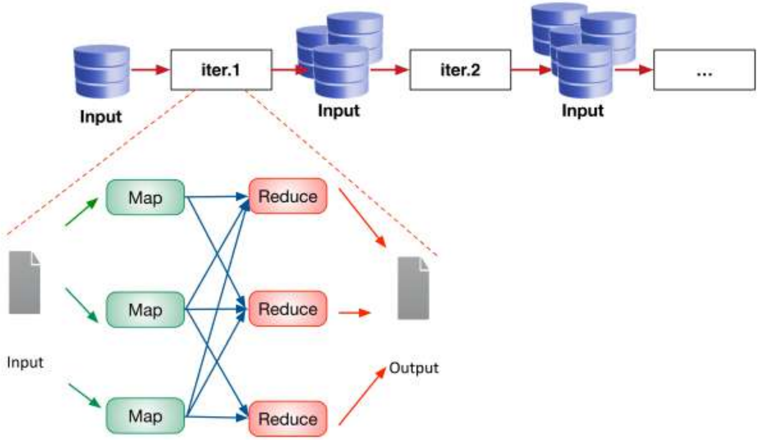
Worker Node



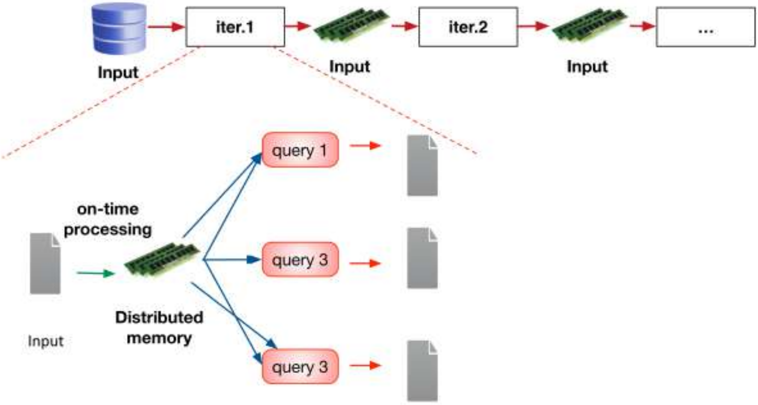
Three interfaces for cluster management:

- Spark's standalone cluster manager
- The Apache Mesos
- Hadoop YARN

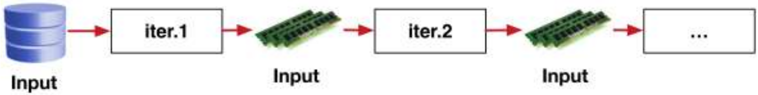
Data in MapReduce



Data in Spark



Data in Spark



- Extract a working set
- Cache the working set
- Query the working set repeatedly

RDD: Resilient Distributed Dataset

- Immutable Data Structure
- In-memory
- Fault tolerant
- Parallel data structure
- Controlled partitioning
- Manipulated using a rich set of operators

RDD: Resilient Distributed Dataset

- Distributed collection of objects that can be cached in memory across cluster nodes
- Manipulated through various parallel operators
- Automatically rebuilt on failure
- Can persist in memory, on disk, or both
- Can be partitioned to control parallel processing
- Interface:
  - Clean language-integrated API for Scala, Python, and Java
  - Can be used interactively from Scala console

RDD: Resilient Distributed Dataset

Currently two types of RDDs

- Parallelized collections: created by executing operators on an existing programming collection
  - Developer can specify the number of slices to cut the dataset into
  - Ideally 2-3 slices per CPU
- Hadoop Datasets: created from any file stored on HDFS or other storage systems supported by Hadoop (S3, Hbase, etc.)
  - These are created using SparkContext's textFile operator
  - Default number of slices in this case is 1 slice per file block

Operators over RDD

Programmer can perform three types of operations:

1. Transformations
2. Actions
3. Persistence

Operators over RDD

1 Transformations:

- Create a new dataset from an existing one
- Lazy in nature
- They are executed only when some action is performed

Example

- `Map(func)`
- `Filter(func)`
- `Distinct()`

Transformations

- Transformations lazy operations on a RDD
- They return RDD objects or collections of RDD
- Are not executed immediately, but only after an action has been executed

Two kind of transformations:

- Wide transformations
- Narrow transformations

Narrow Transformations

- They are the result of `map`, `filter` and such that is from the data from a single partition only
  - i.e. it is self-sustained
- An output RDD has partitions with records that originate from a single partition in the parent RDD
- Spark groups narrow transformations as a stage

Wide Transformations

- They are the result of `groupByKey` and `reduceByKey`
- The data required to compute the records in a single partition may reside in many partitions of the parent RDD
- All of the tuples with the same key must end up in the same partition, processed by the same task
- Spark must execute RDD shuffle, which transfers data across cluster and results in a new stage with a new set of partitions

Transformations

- `filter`
- `map`
- `flatMap`
- `sample`
- `union`
- `intersection`
- `distinct`
- `groupByKey`
- `reduceByKey`
- `sortByKey`
- `join`
- `cogroup`
- `cartesian`

Operators over RDD

2 Actions:

- Returns a value to the driver program
- Exports data to a storage system after performing a computation

Example

- `Count()`
- `Reduce(funct)`
- `Collect`
- `Take()`

Actions

- Actions are operations that return values
  - i.e. any RDD operation that returns a value of any type but an RDD is an action
    - e.g., `saveAs`, `collect`, `take`, `reduce`, etc.
- Actions are synchronous: they trigger execution of RDD transformations to return values
- Until no action is fired, the data to be processed is not even accessed
- Only actions can materialize the entire process with real data
- Cause data to be returned to driver or saved to output
- Cause data retrieval and execution of all transformations on RDDs

Actions

- `reduce`
- `collect`
- `count`
- `first`
- `take`
- `takeSample`
- `takeOrdered`
- `saveAsTextFile`
- `saveAsSequenceFile`
- `foreach`

Operators over RDD

3 Persistence:

- For caching datasets in-memory for future operations
- Option to store on disk or RAM or mixed (Storage Level)

Example

- `Persist()`
- `Cache()`

Persistence

By default, each transformed RDD is recomputed each time you run an action on it, unless you specify the RDD to be cached in memory

- RDD can be persisted on disks as well
- Caching is the key tool for iterative algorithms
- Using `persist()`, one can specify the Storage Level for persisting an RDD
- `cache()` is just a short hand for default storage level, which is MEMORY_ONLY

Persistence

Storage Level for persist():

- MEMORY_ONLY
- MEMORY_AND_DISK
- DISK_ONLY
- MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.

Which Storage level is best?

- Try to keep in-memory as much as possible
- Try not to spill to disc unless your computed datasets are memory expensive
- Use replication only if you want fault tolerance

How does Spark work?

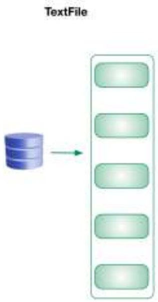
- User applications create RDDs, transform them, and run actions
- This results in a DAG (Directed Acyclic Graph) of operators
- DAG is compiled into stages
- Each stage is executed as a series of Tasks (one Task for each Partition)
- Actions drive the execution

Example in Python

```
text_file = sc.textFile(''hdfs://...'')
counts = text_file.flatMap(lambda line: line.split(" "))
                  .map(lambda word: (word, 1))
                  .reduceByKey(lambda a, b: a + b)
output = counts.collect()
output.saveAsTextFile(''hdfs://...'')
```

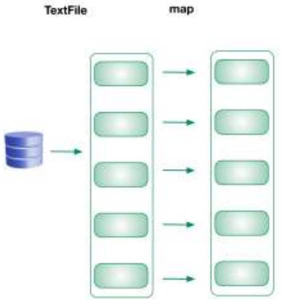
Example in Python

```
text_file = sc.textFile(''hdfs://...'')
```



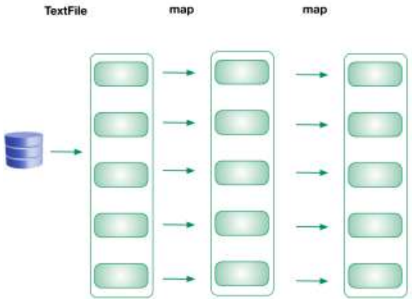
Example in Python

```
text_file = sc.text_file = sc.textFile(''hdfs://...'')
counts = text_file.flatMap(lambda line: line.split(" "))
```



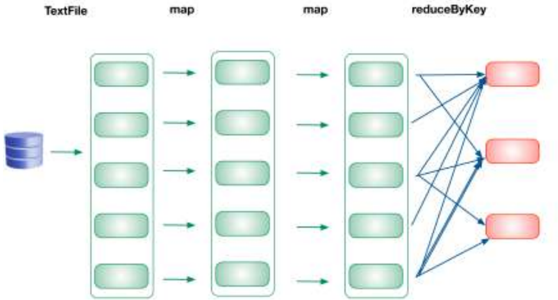
Example in Python

```
text_file = sc.textFile(''hdfs://...'')
counts = text_file.flatMap(lambda line: line.split(" "))
                  .map(lambda word: (word, 1))
```



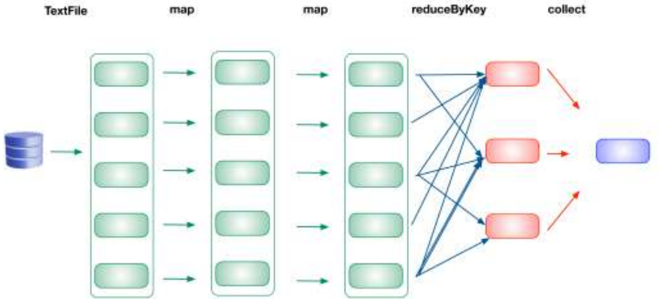
Example in Python

```
text_file = sc.textFile(''hdfs://...'')
counts = text_file.flatMap(lambda line: line.split(" "))
                  .map(lambda word: (word, 1))
                  .reduceByKey(lambda a, b: a + b)
```



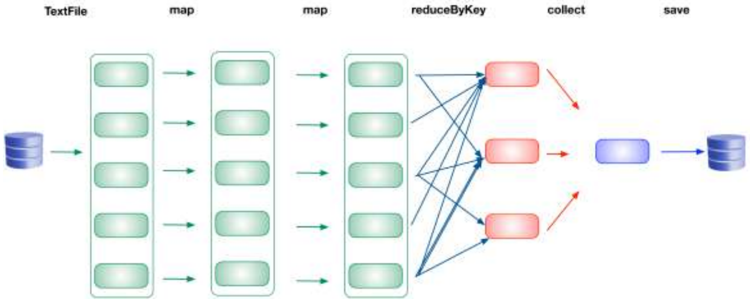
Example in Python

```
text_file = sc.textFile(''hdfs://...'')
counts = text_file.flatMap(lambda line: line.split(" "))
                  .map(lambda word: (word, 1))
                  .reduceByKey(lambda a, b: a + b)
output = counts.collect()
```



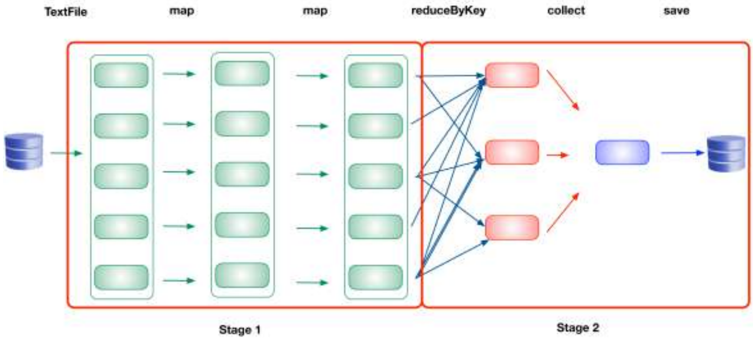
Example in Python

```
text_file = sc.textFile(''hdfs://...'')
counts = text_file.flatMap(lambda line: line.split(" "))
                  .map(lambda word: (word, 1))
                  .reduceByKey(lambda a, b: a + b)
output = counts.collect()
output.saveAsTextFile(''hdfs://...'')
```



Example in Python

2 Stages:



Stages

Stages are sequences of RDDs, that don't have a Shuffle in between

- Spark:
  - Creates a task for each Partition in the new RDD
  - Schedules and assigns tasks to slaves
  - All this happens internally

Summary

- Task: The fundamental unit of execution in Spark
- Stage: Set of Tasks that run in parallel
- DAG: Logical Graph of RDD operations
- RDD: Parallel dataset with partitions

References I

- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber.
  Bigtable: A distributed storage system for structured data (awarded best paper!).
  In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218, 2006.

- Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia.
  Learning Spark: Lightning-Fast Big Data Analytics.
  O'Reilly Media, Inc., 1st edition, 2015.

References II

A growing number of applications are being built with Spark.
https://www.oreilly.com/.
Accessed 2016.

Spark Hadoop.
http://spark.apache.org/docs/latest/cluster-overview.html.
Accessed 2016.