بسم الله الرحمن الرحیم

تکنولوژی کامپیوتر

جلسه‌ی بیست و ششم

کار کردن با هدوپ – شروع اسپارک

جلسه گذشته

Batch Processing

- High Throughput Vs. Low Latency

- The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware

Assumptions and Goals

- Hardware Failure
  - *Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.*

Assumptions and Goals

- Streaming Data Access
  - *Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access. POSIX imposes many hard requirements that are not needed for applications that are targeted for HDFS. POSIX semantics in a few key areas has been traded to increase data throughput rates.*

Assumptions and Goals

- Large Data Sets
  - *Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.*

Assumptions and Goals

- Simple Coherency Model: write-once-read-many
  - *HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed except for appends and truncates. Appending the content to the end of the files is supported but cannot be updated at arbitrary point. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model.*

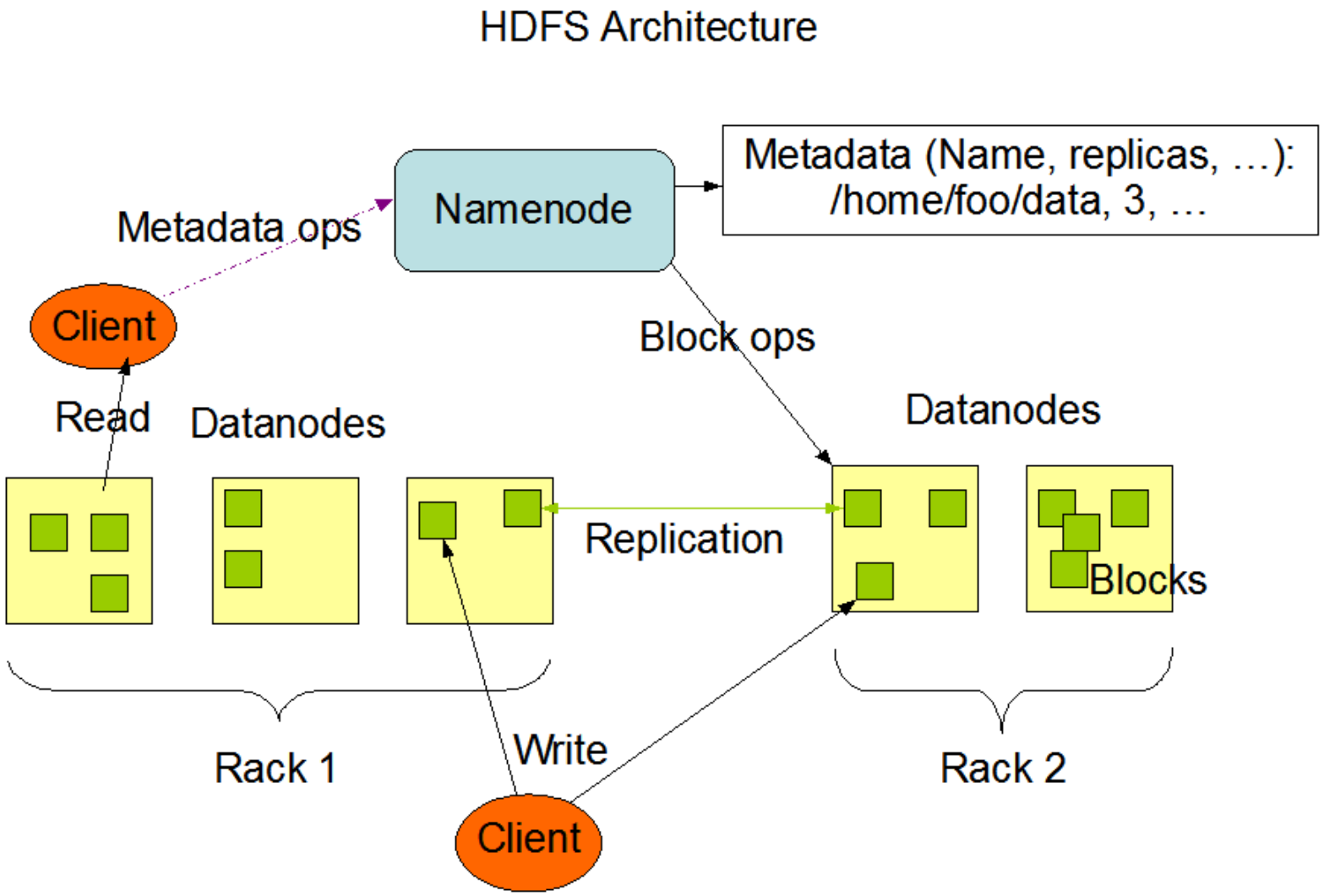
Assumptions and Goals

- Moving Computation is Cheaper than Moving Data
  - *A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.*

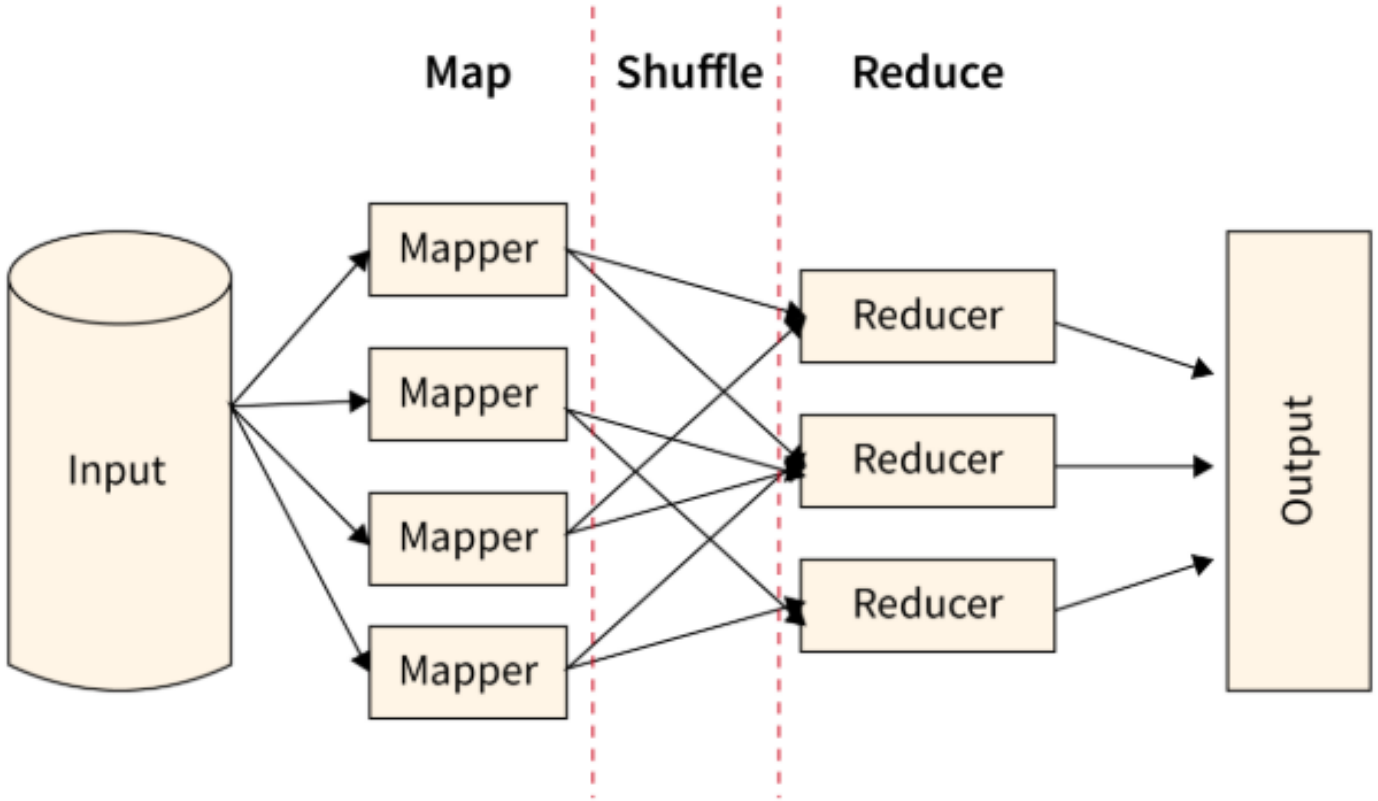
Assumptions and Goals

- Portability Across Heterogeneous Hardware and Software Platforms
  - *HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications.*

HDFS Components



MAP REDUCE



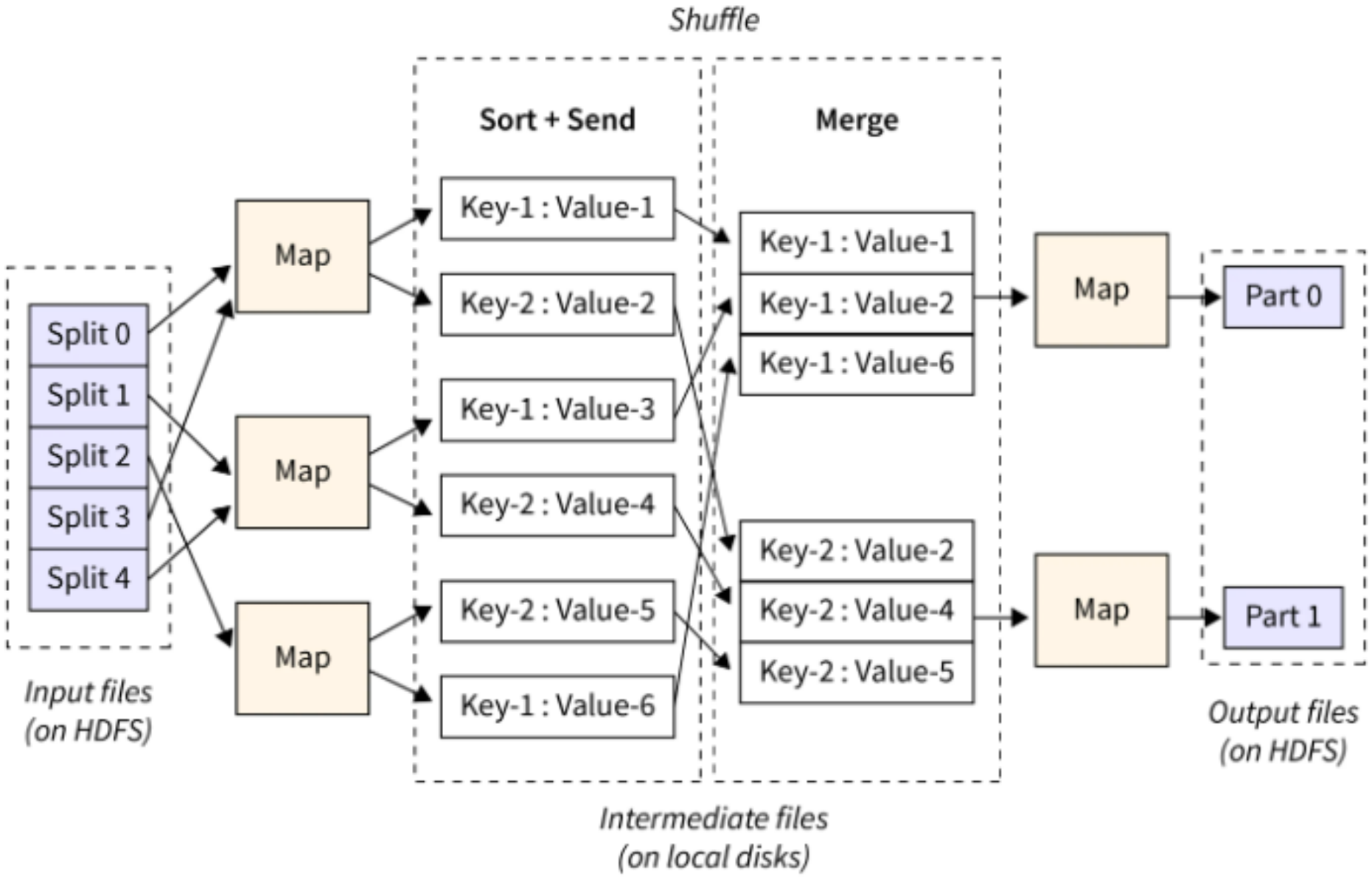
SCALER Topics

Map Stage

- Input data is divided into smaller chunks or blocks.
- Several worker nodes work in parallel to process each chunk independently.
- A "Map" function is applied to each data chunk, generating intermediate key-value pairs.
- The Map function's goal is to extract relevant information from the input data and prepare it for further processing.

Reduce Stage

- After the Map stage, the intermediate key-value pairs are grouped by key.
- The grouped key-value pairs are then shuffled and sorted based on their keys.
- The purpose of the shuffle and sort phase is to bring together all the intermediate values associated with the same key and make them available to the corresponding Reduce function.
- Once the shuffling and sorting are complete, a "Reduce" function is applied to perform aggregation, analysis, or other computations on the grouped data.
- The output is a set of final key-value pairs from the computation.



SCALER Topics

جلسه جدید

تمرین مپ ریدیوس

Friends In common

- Assume the friends are stored as Person->[List of Friends]
- For each pair of users A and B that A and B is friend of each other, we want to compute the list of friend in common.

- A -> B C D
- B -> A C D E
- C -> A B D E
- D -> A B C E
- E -> B C D

- For map(A -> B C D) :

- (A B) -> B C D
- (A C) -> B C D
- (A D) -> B C D

- For map(B -> A C D E) : (Note that A comes before B in the key)

- (A B) -> A C D E
- (B C) -> A C D E
- (B D) -> A C D E
- (B E) -> A C D E

- For map(C -> A B D E) :

- (A C) -> A B D E
- (B C) -> A B D E
- (C D) -> A B D E
- (C E) -> A B D E

- For map(D -> A B C E) :

- (A D) -> A B C E
- (B D) -> A B C E
- (C D) -> A B C E
- (D E) -> A B C E

- And finally for map(E -> B C D):

- (B E) -> B C D
- (C E) -> B C D
- (D E) -> B C D

- Before we send these key-value pairs to the reducers, we group them by their keys and get:

- (A B) -> (A C D E) (B C D)
- (A C) -> (A B D E) (B C D)
- (A D) -> (A B C E) (B C D)
- (B C) -> (A B D E) (A C D E)
- (B D) -> (A B C E) (A C D E)
- (B E) -> (A C D E) (B C D)
- (C D) -> (A B C E) (A B D E)
- (C E) -> (A B D E) (B C D)
- (D E) -> (A B C E) (B C D)

- (A B) -> (C D)
- (A C) -> (B D)
- (A D) -> (B C)
- (B C) -> (A D E)
- (B D) -> (A C E)
- (B E) -> (C D)
- (C D) -> (A B E)
- (C E) -> (B D)
- (D E) -> (B C)

Matrix Multiplication

- A: n×k matrix: records: (row, col, value)
- B: k×m matrix: (row, col, value)
- Calculate C = A×B

Matrix Multiplication

- Idea 1: two step reduce-side

Matrix Multiplication

- Idea 2: one-step product

Matrix Multiplication

- What happen in k is large? But n and m is small?
- For example in linear regression problem, we have k inputs which has n features.
- Matrix X: k×n
- We need to calculate X^T× X

$$\begin{bmatrix} 9 & 3 & 5 \\ 4 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & -5 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix}$$

$$\begin{bmatrix} 9 & 3 & 5 \\ 4 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & -5 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix}$$

$$\begin{bmatrix} 9 & 18 \\ 4 & 8 \end{bmatrix}$$

$$\begin{bmatrix} 9 & 3 & 5 \\ 4 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & -5 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix}$$

$$\begin{bmatrix} 9 & 18 \\ 4 & 8 \end{bmatrix} + \begin{bmatrix} 9 & -15 \\ 3 & -5 \end{bmatrix}$$

$$\begin{bmatrix} 9 & 3 & 5 \\ 4 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & -5 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix}$$

$$\begin{bmatrix} 9 & 18 \\ 4 & 8 \end{bmatrix} + \begin{bmatrix} 9 & -15 \\ 3 & -5 \end{bmatrix} + \begin{bmatrix} 10 & 15 \\ 4 & 6 \end{bmatrix}$$

$$\begin{bmatrix} 9 & 3 & 5 \\ 4 & 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & -5 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 28 & 18 \\ 11 & 9 \end{bmatrix}$$

$$\begin{bmatrix} 9 & 18 \\ 4 & 8 \end{bmatrix} + \begin{bmatrix} 9 & -15 \\ 3 & -5 \end{bmatrix} + \begin{bmatrix} 10 & 15 \\ 4 & 6 \end{bmatrix}$$

Sorting

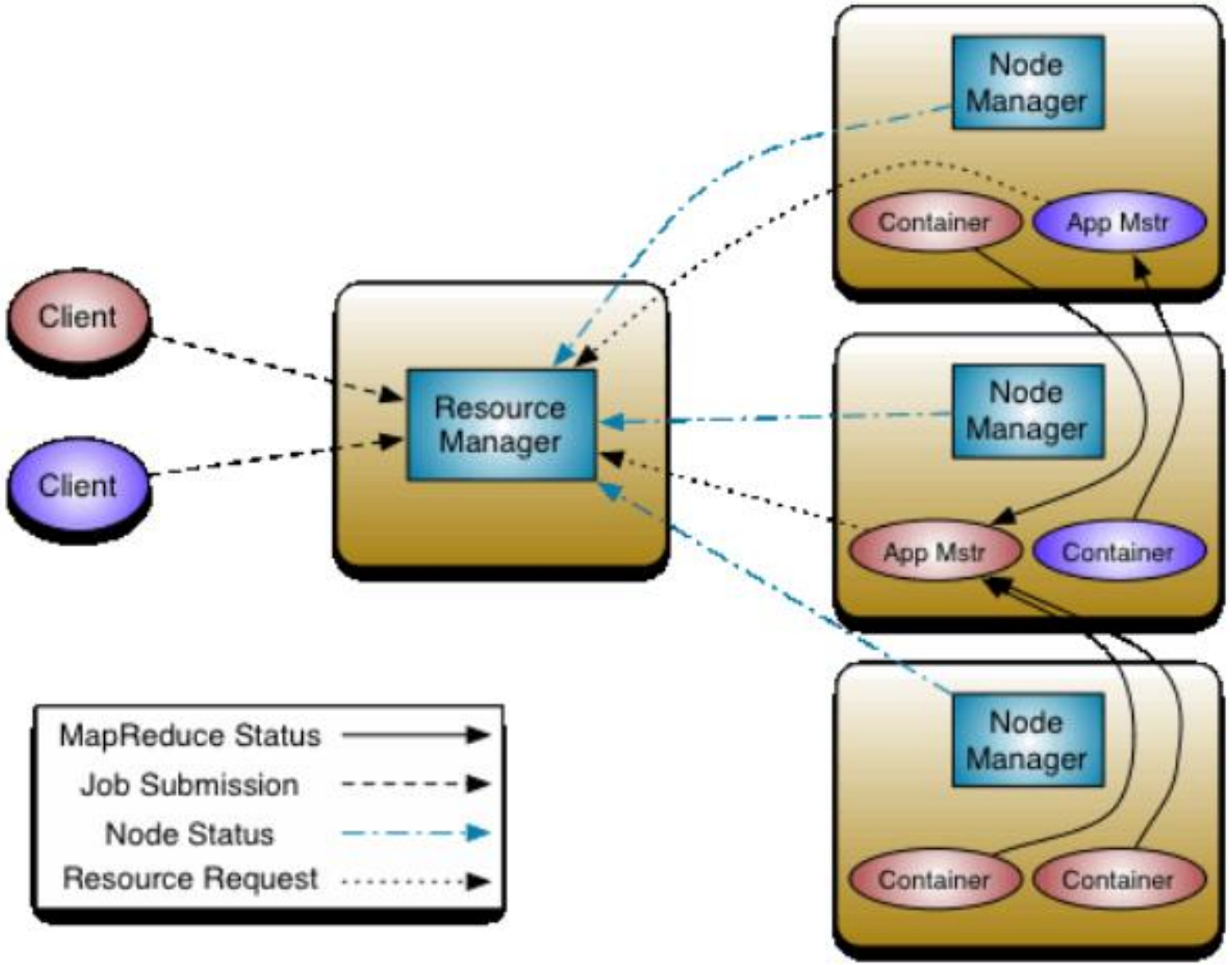
OUTPUT OF BATCH WORKFLOWS

What happen if it is not succeed?

یک مثال که MapReduce چندان هم خوب نیست.

YARN

Yet Another Resource Negotiator


Node Manager
Container
App Mstr
Client
Client
Resource Manager
Node Manager
App Mstr
Container
Node Manager
Container
Container
MapReduce Status
Job Submission
Node Status
Resource Request

کار کردن با HDFS

HADOOP MAPREDUCE JAVA API

Java API

- Job
- Input
- Mapper
- Partitioner
- GroupingComparator
- Combiner
- Reducer
- Output

Java API

- Writable interface
- DistributedCache

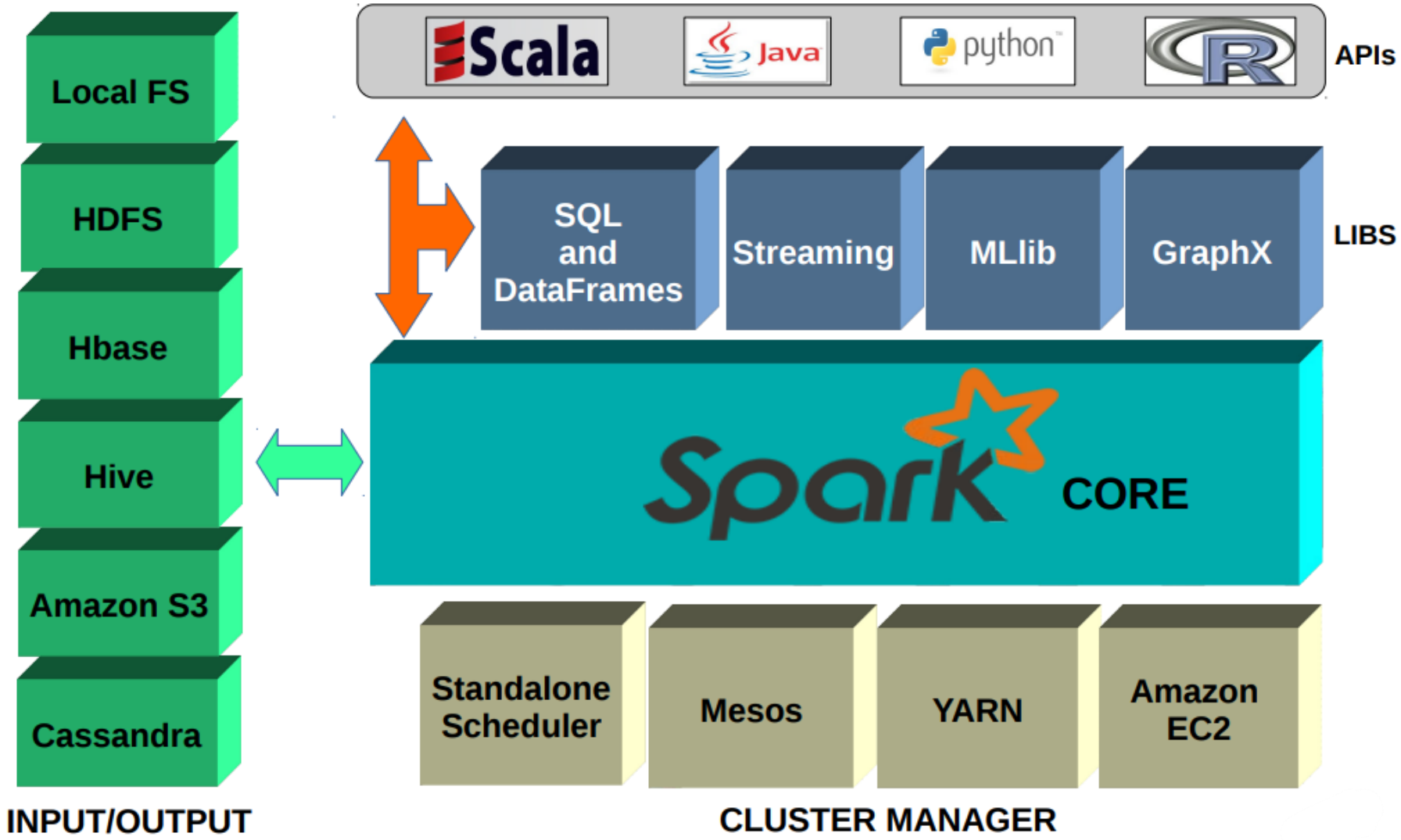
SPARK

Spark Vs. Hadoop MapReduce

	Hadoop	Spark 100TB	Spark 1PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min

Use memory!

Spark Basics: architecture



Resilient Distributed Datasets (RDDs)

Data manipulation in Spark is heavily based on RDDs. An RDD is an interface composed of:

- a set of partitions
- a list of dependencies
- a function to compute a partition given its parents
- a partitioner (optional)
- a set of preferred locations per partition (optional)

- Simply stated: an RDD is a distributed collections of items.
- an RDD is a read-only (i.e., immutable) collection of items partitioned across a set of machines that can be rebuilt if a partition is destroyed.

The RDD is the most fundamental concept in Spark since all work in Spark is expressed as:

- creating RDDs
- transforming existing RDDs
- performing actions on RDDs

Creating RDDs

Spark provides two ways to create an RDD:

- **loading** an already existing set of objects
- **parallelizing** a data collection in the driver

Creating RDDs

```
// define the spark context
val sc = new SparkContext(...)

// hdfsRDD is an RDD from an HDFS file
val hdfsRDD = sc.textFile("hdfs://...")

// localRDD is an RDD from a file in the local file system
val localRDD = sc.textFile("localfile.txt")

// define a List of strings
val myList = List("this", "is", "a", "list", "of", "strings")

// define an RDD by parallelizing the List
val listRDD = sc.parallelize(myList)
```

RDD Operations

There are **transformations** on RDDs that allow us to create new RDDs: `map, filter, groupBy, reduceByKey, partitionBy, sortByKey, join, etc`

Also, there are **actions** applied in the RDDs: `reduce, collect, take, count, saveAsTextFile, etc`

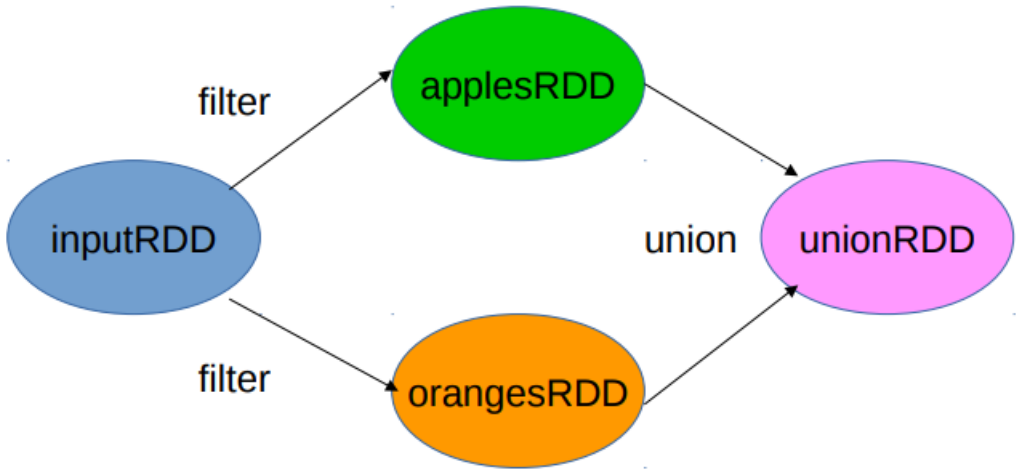
Note: computation takes place only in actions and not on transformations! (This is a form of **lazy evaluation**. More on this soon.)

RDD Operations: transformations

```
val inputRDD = sc.textFile("myfile.txt")

// lines containing the word “apple”
val applesRDD = inputRDD.filter(x => x.contains("apple"))

// lines containing the word “orange”
val orangesRDD = inputRDD.filter(x => x.contains("orange"))

// perform the union
val aoRDD = applesRDD.union(orangesRDD)
```

RDD Operations: transformations

Graphically speaking:



RDD Operations: actions

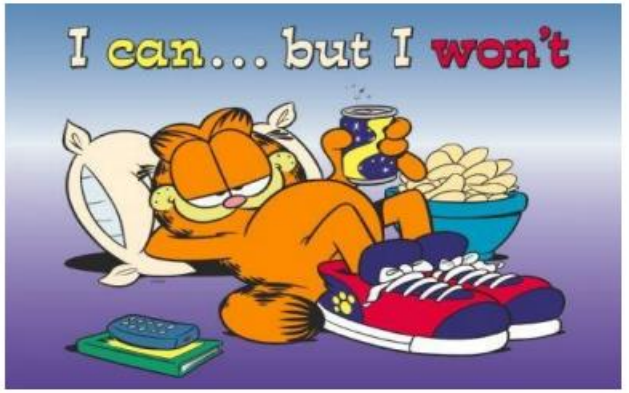
An action denotes that **something must be done**

We use the action `count()` to find the number of lines in unionRDD containing apples or oranges (or both) and then we print the 5 first lines using the action `take()`

```
val numLines = unionRDD.count()
unionRDD.take(5).foreach(println)
```

Lazy Evaluation

The benefits of being lazy



1. more optimization alternatives are possible if we see the **big picture**
2. we can avoid unnecessary computations

Ex:

Assume that from the unionRDD we need only the first 5 lines.

If we are eager, we need to compute the union of the two RDDs, materialize the result and then select the first 5 lines.

If we are lazy, there is no need to even compute the whole union of the two RDDs, since when we find the first 5 lines we may stop.

Lazy Evaluation

At any point we can **force the execution** of transformation by applying a simple action such as **`count()`**. This may be needed for debugging and testing.

Basic RDD Transformations

Assume that our RDD contains the list `{1,2,3}`.

`map()`	`rdd.map(x => x + 2)`	`{3,4,5}`
`flatMap()`	`rdd.flatMap(x => List(x-1,x,x+1))`	`{0,1,2,1,2,3,2,3,4}`
`filter()`	`rdd.filter(x => x>1)`	`{2,3}`
`distinct()`	`rdd.distinct()`	`{1,2,3}`
`sample()`	`rdd.sample(false,0.2)`	`non-predictable`

Two-RDD Transformations

These transformations require two RDDs

union()	`rdd.union(another)`
intersection()	`rdd.intersection(another)`
subtract()	`rdd.substract(another)`
cartesian()	`rdd.cartesian(another)`

Some Actions

collect()	`rdd.collect()`	`{1,2,3}`
count()	`rdd.count()`	`3`
countByValue()	`rdd.countByValue()`	`{(1,1),(2,1),(3,1)}`
take()	`rdd.take(2)`	`{1,2}`
top()	`rdd.top(2)`	`{3,2}`
reduce()	`rdd.reduce((x,y) => x+y)`	`6`
foreach()	`rdd.foreach(func)`	

DAG (DIRECTED ACYCLIC GRAPHS)

RDDs and DAGs

A set of RDDs corresponds is transformed to a Directed Acyclic Graph (DAG)

Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:

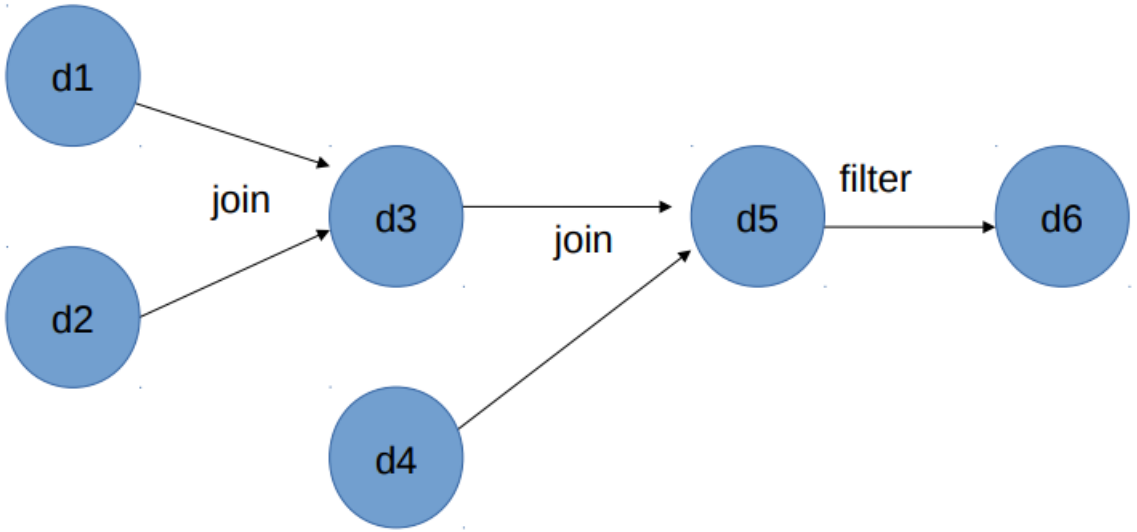
> Build stages of tasks

> Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready

> Lower level scheduler will schedule data based on locality

> Resubmit failed stages if outputs are lost

DAG Scheduling



Distributed Execution in Spark

Outline of the whole process:

1. The user submits a job with **`spark-submit`**.
2. **`spark-submit`** launches the driver program and invokes the **`main()`** method specified by the user.
3. The **driver program** contacts the **cluster manager** to ask for resources to launch **executors**.
4. The **cluster manager** launches **executors** on behalf of the **driver program**.
5. The **driver process** runs through the user application. Based on the RDD actions and transformations in the program, the **driver** sends work to **executors** in the form of **tasks**.
6. **Tasks** are run on **executor processes** to compute and save results.
7. If the **driver's `main()`** method exits or it calls **`SparkContext.stop()`** , it will terminate the **executors** and release resources from the **cluster manager**.

Under the Hood