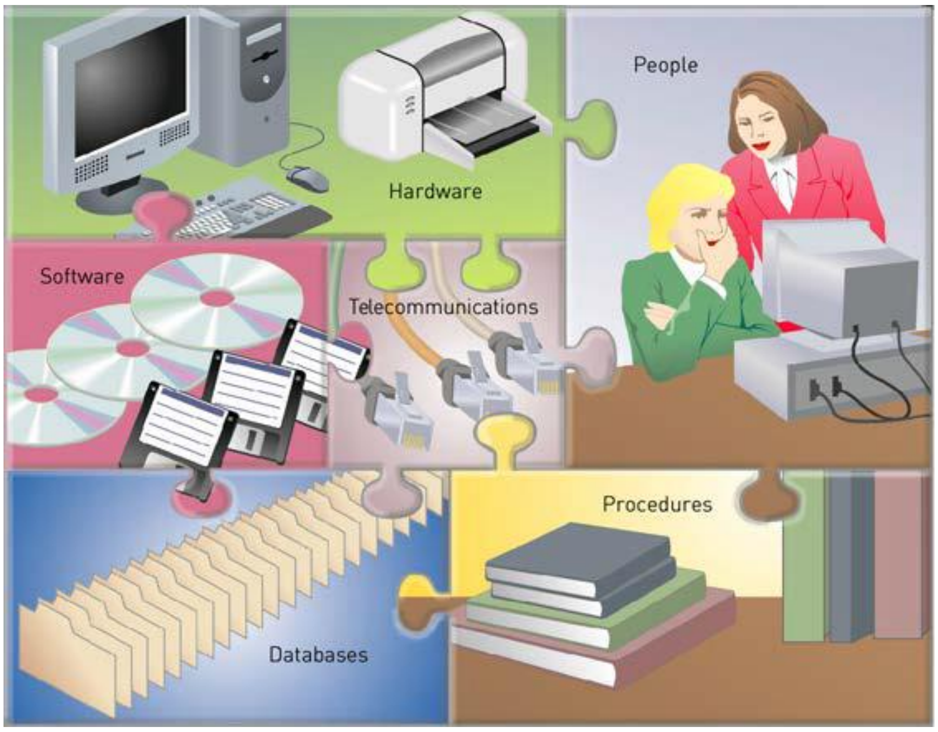
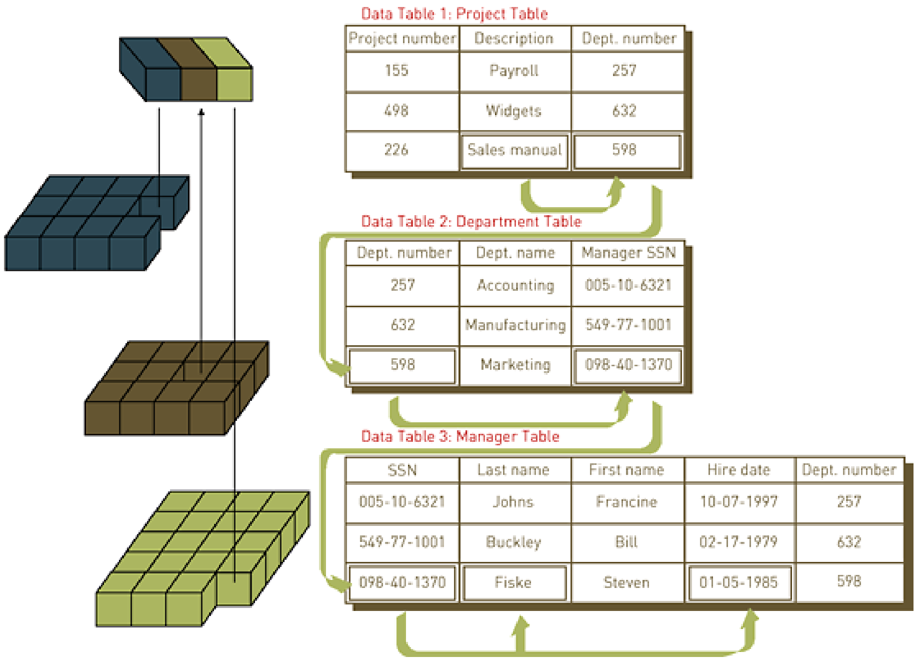
Summary Lectures DSS

1. Lecture 1: Course introduction





Data Table 1: Project Table

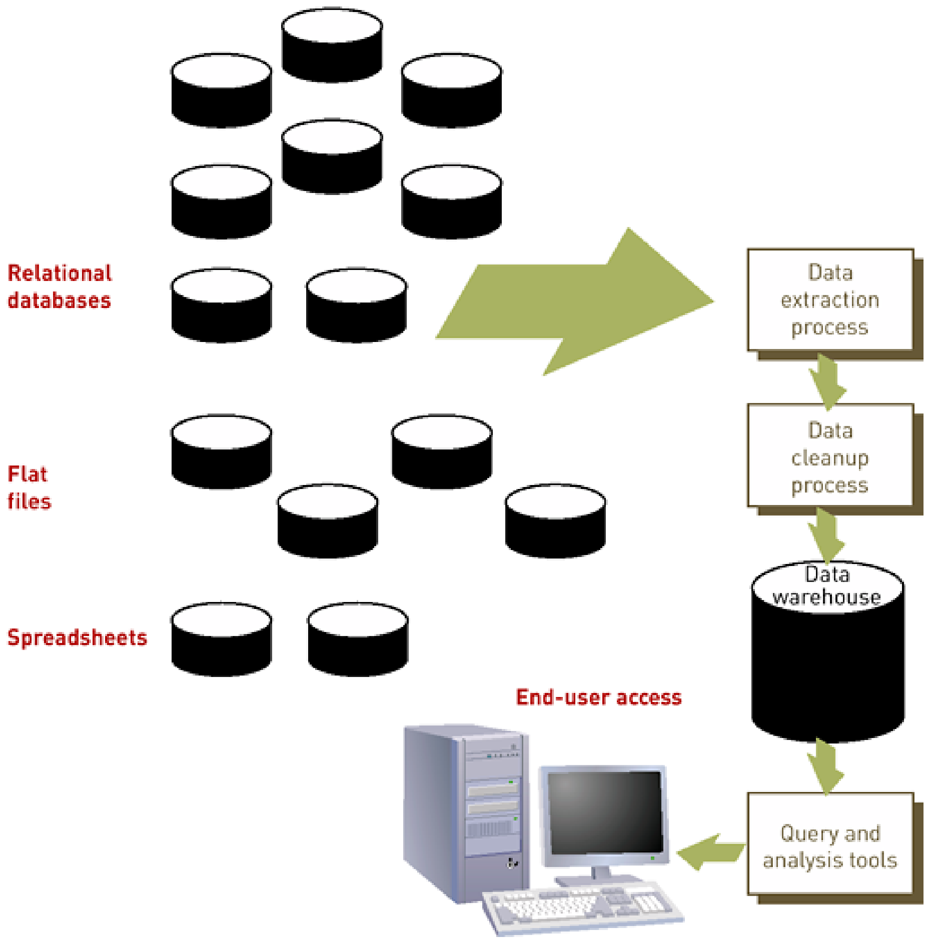
Project number	Description	Dept. number
155	Payroll	257
498	Widgets	632
226	Sales manual	598

Data Table 2: Department Table

Dept. number	Dept. name	Manager SSN
257	Accounting	005-10-6321
632	Manufacturing	549-77-1001
598	Marketing	098-40-1370

Data Table 3: Manager Table

SSN	Last name	First name	Hire date	Dept. number
005-10-6321	Johns	Francine	10-07-1997	257
549-77-1001	Buckley	Bill	02-17-1979	632
098-40-1370	Fiske	Steven	01-05-1985	598



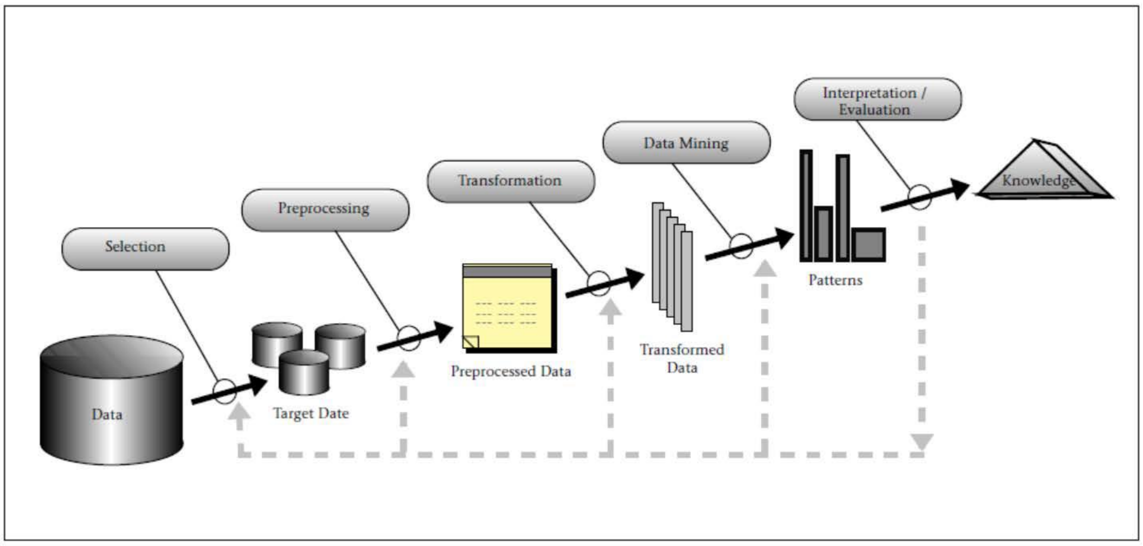
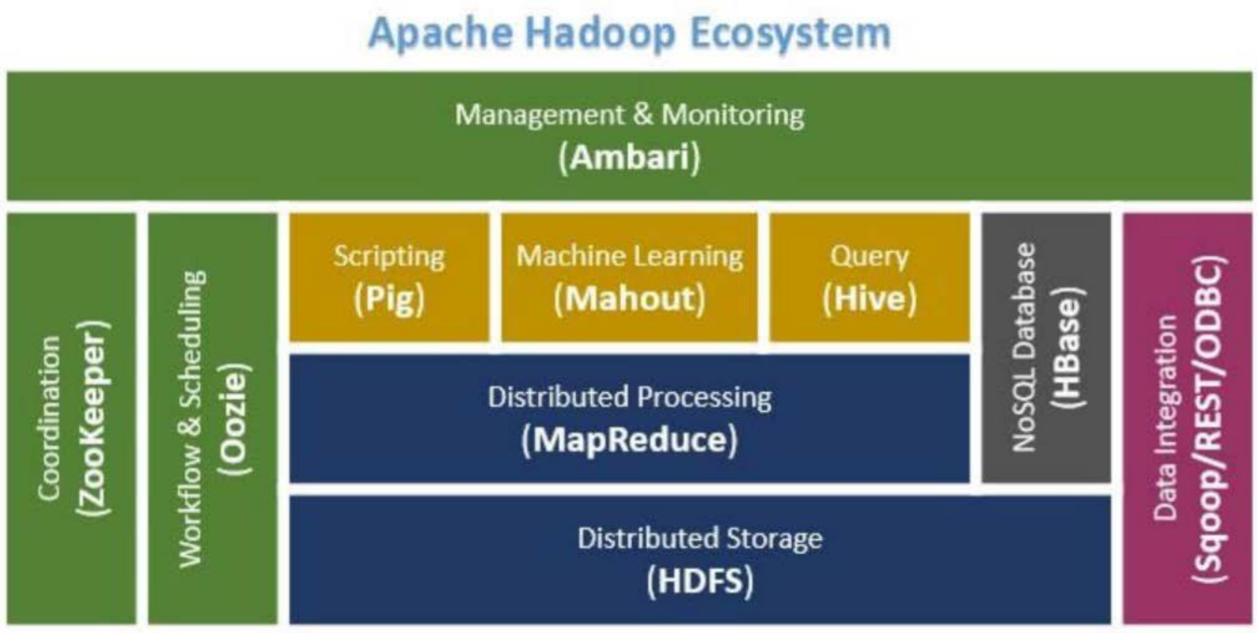


Figure 1. An Overview of the Steps That Compose the KDD Process.

2. Lecture 2: Distributed Computing with Hadoop

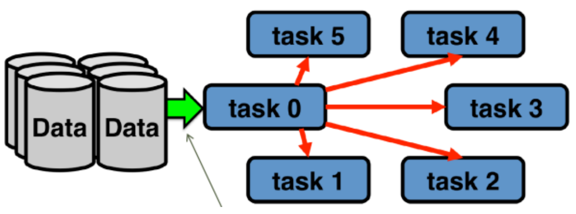
Apache Hadoop is an open source software framework for storage and large scale processing of data-sets on clusters of commodity hardware.

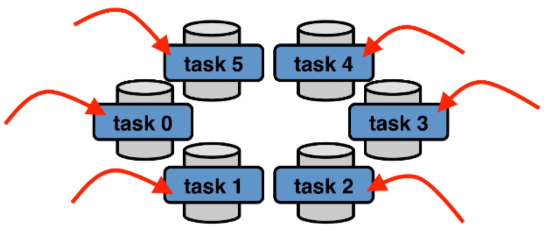

Apache Hadoop Ecosystem
Management & Monitoring (Ambari)
Coordination (ZooKeeper)
Workflow & Scheduling (Oozie)
Scripting (Pig)
Machine Learning (Mahout)
Query (Hive)
NoSQL Database (HBase)
Data Integration (Sqoop/REST/ODBC)
Distributed Processing (MapReduce)
Distributed Storage (HDFS)

1. Distributed storage: HDFS (Hadoop Distributed File System)
2. Distributed processing: MapReduce
3. Scripting: Pig
4. Machine learning: Mahout
5. Query: Hive
6. NoSQL database: HBase
7. Data integration: Sqoop/REST/ODBC
8. Coordination: ZooKeeper
9. Workflow & scheduling: Oozie
10. Management & monitoring: Ambari

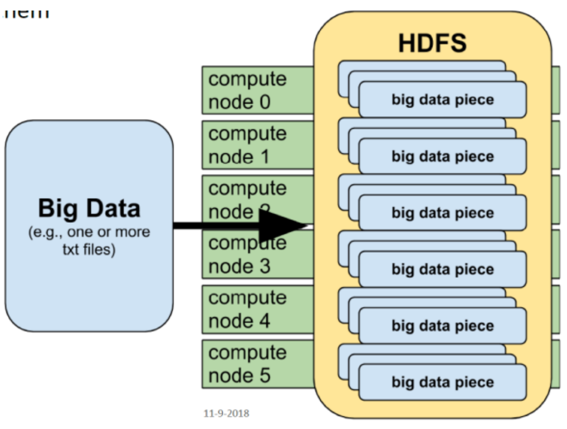
Map-Reduce brings compute to the data in contrast to traditional parallelism, which brings data to the compute resources. Hadoop accomplishes this by storing data in a replicated and distributed fashion on HDFS.

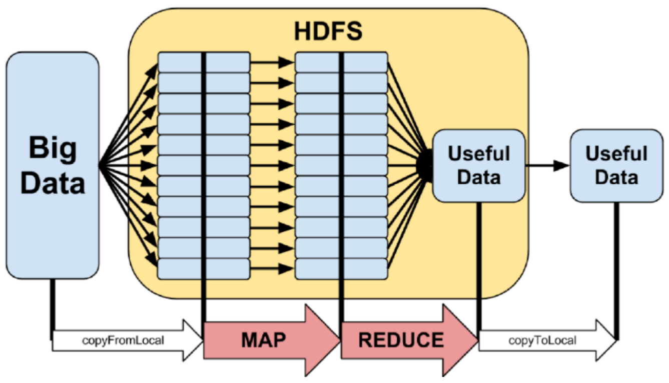
- HDFS stores files in chunks which are physically stored on multiple compute nodes.
- HDFS still presents data to users and applications as single continuous file despite the above fact





Map-reduce is ideal for operating on very large, flat (unstructured) datasets and perform trivially parallel operations on them. Hadoop jobs go through a map stage and a reduce stage where: the mapper transforms the <u>raw input data into key-value pairs</u> where multiple values for the same key may occur, the reducer transforms all of the key-value pairs sharing a common key into a single key with a single value.





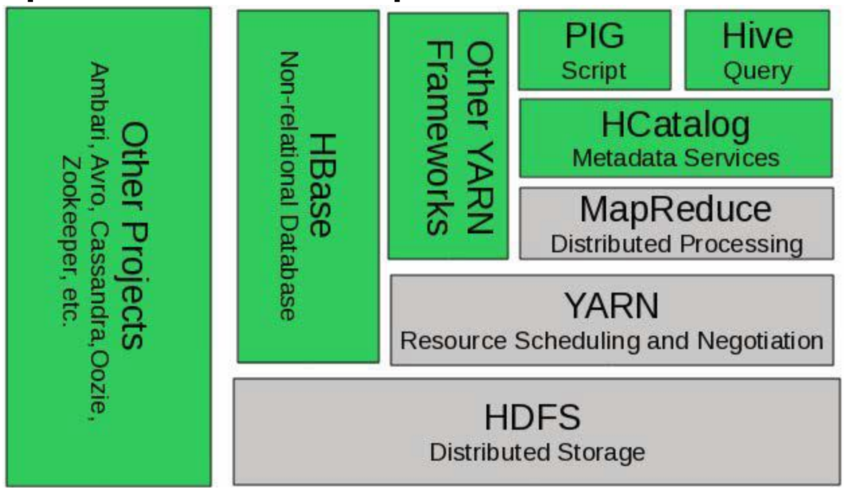
Why Hadoop:

- Move computation to data
- Scalability
- Reliability

Key issue:

- New kinds of analysis → complex algorithms

Apache framework components

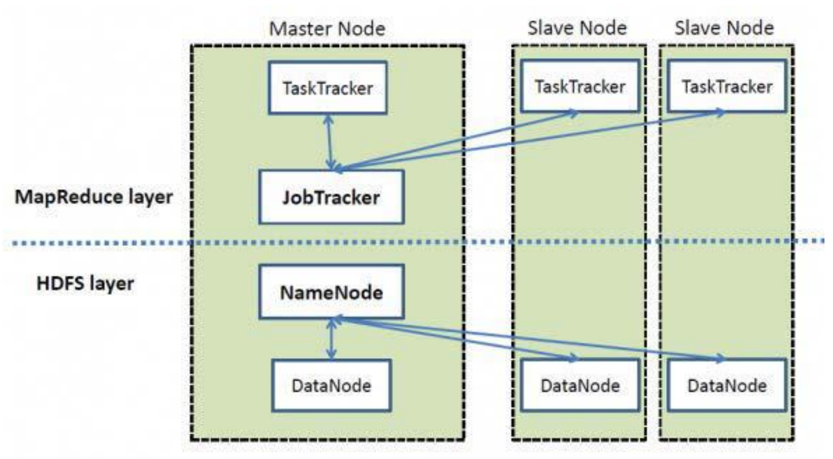


Hadoop common

Hadoop Distributed File System (HDFS)

Hadoop Yet Another Resource Negotiator (YARN)

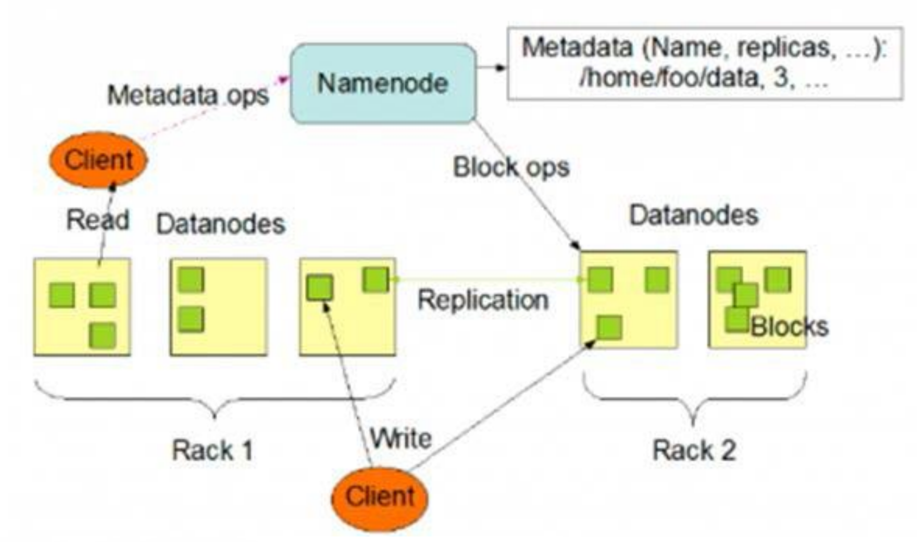
Hadoop MapReduce

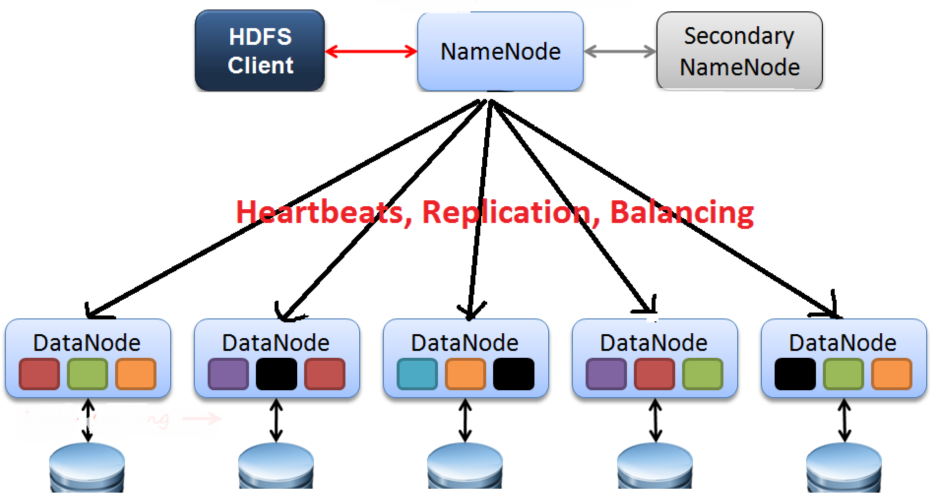


What is HDFS?

A distributed, scalable, and portable file-system written in Java for the Hadoop framework.

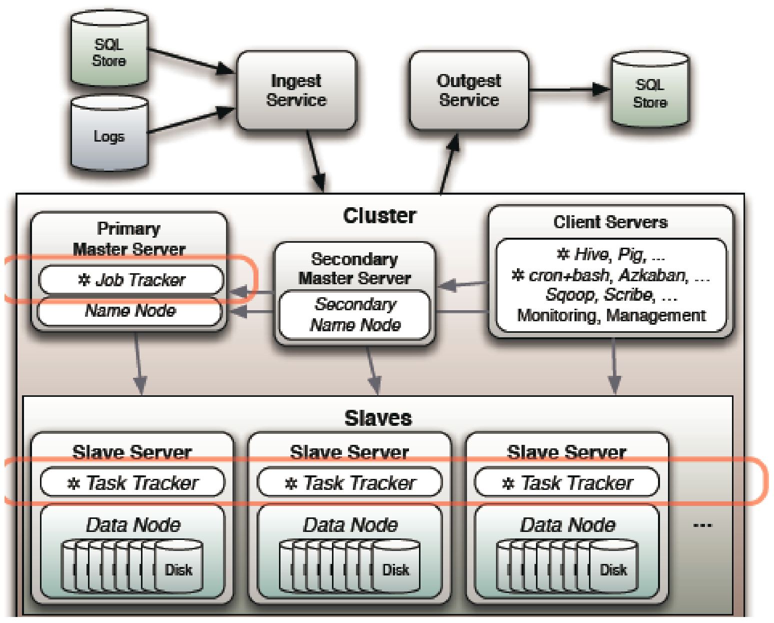
Fault-tolerant: without RAID (Redundant array of independent disks), and with commodity hardware





Secondary NameNode: replica for if the first fails.

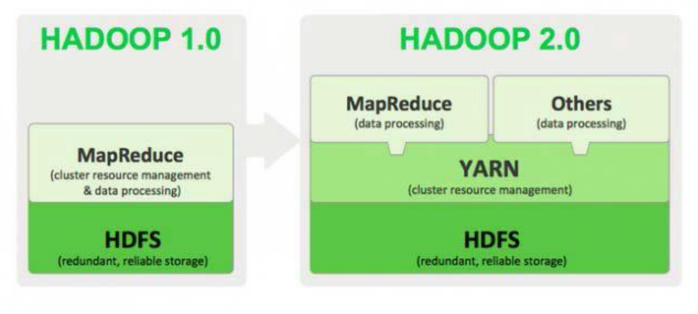
Blocks on DataNodes are replicated on multiple DataNodes for if one fails. Usually 3 times.



You have a master node (or server). This node controls the slave nodes (or servers).

Hadoop streaming: A utility to enable Map Reduce code in many languages like C, Perl, Python, C++, Bash, etc.

YARN (Yet Another Resource Negotiator)

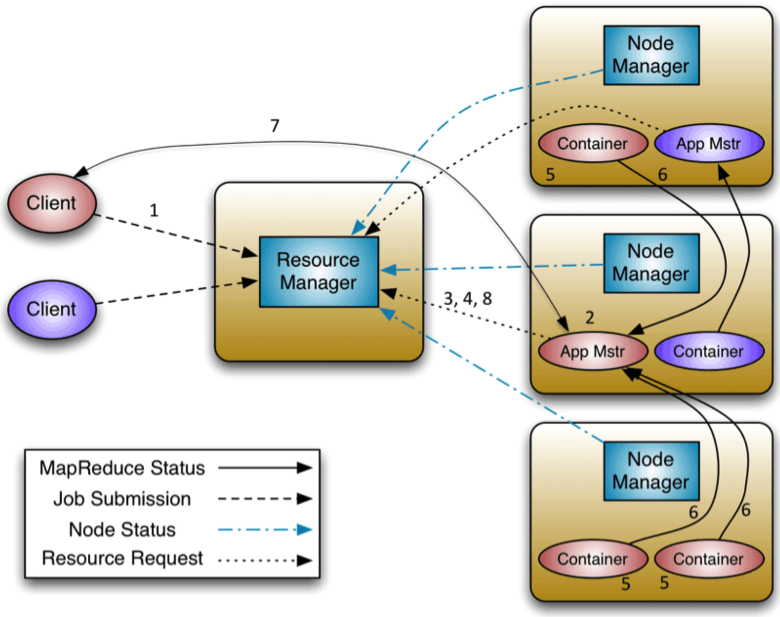

HADOOP 1.0
MapReduce
(cluster resource management & data processing)
HDFS
(redundant, reliable storage)
HADOOP 2.0
MapReduce
(data processing)
Others
(data processing)
YARN
(cluster resource management)
HDFS
(redundant, reliable storage)

YARN = Apache Hadoop NextGen MapReduce

YARN enhances the power of a Hadoop compute cluster

- MapReduce compatibility
- Supports other workloads
- Improved cluster utilization

Fundamentally, YARN splits up two major functionalities of the jobtracker, namely: resource management, and job scheduling and monitoring. The idea is to have one global resource manager, and one application master manager per application.

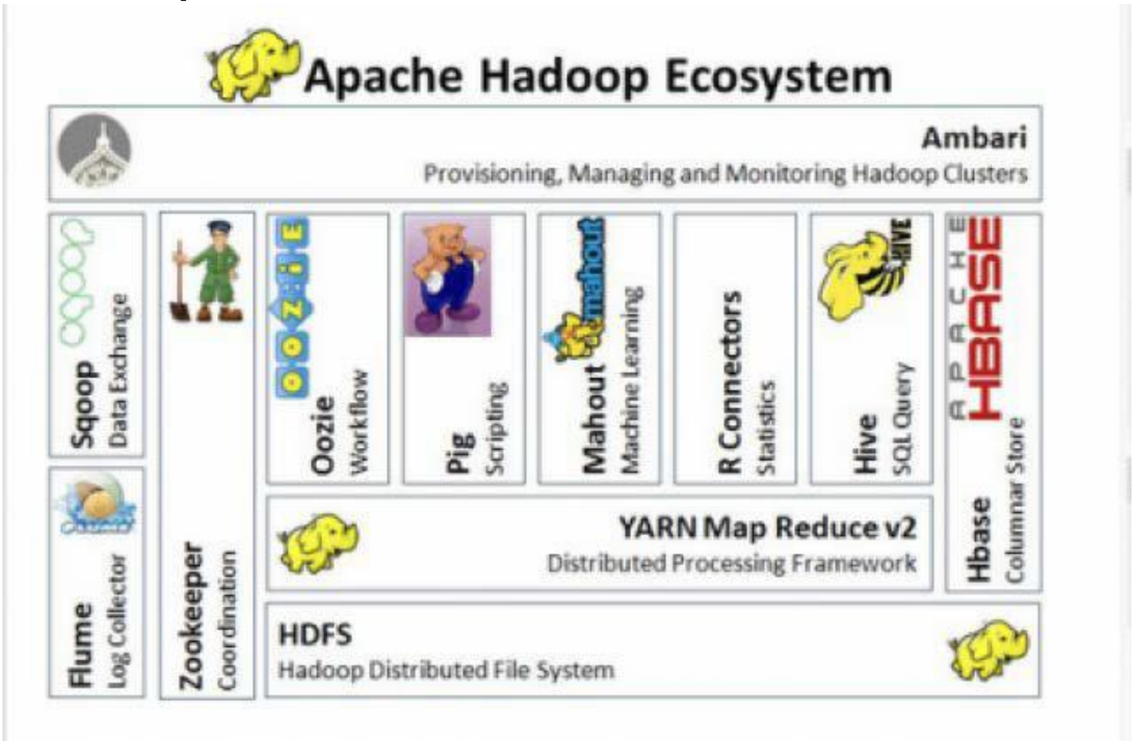


Important: client submits application through the resource manager, the resource manager than launches the application master.

The AppMstr negotiates with the containers.

The client directly communicates with the AppMstr during application execution.

The Hadoop Zoo



- Apache Sqoop
  a. SQL to Hadoop, tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.
- Apache HBase
  a. Column-oriented database management system
  b. Key-value store
  c. Based on Google Big Table
     - A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.
  d. Can hold extremely large data
  e. Dynamic data model
  f. Not a relational DBMS
- Apache Pig
  a. High level programming on top of Hadoop MapReduce
  b. Language: Pig Latin
  c. Data analysis problems as data flows
- Apache Hive
  a. Data warehouses software facilitates querying and managing large datasets residing in distributed storage
  b. Facilitates querying and managing large datasets in HDFS
  c. Mechanism to project structure onto this data and query the data using a SQL-like language called HiveQL
- Apache Oozie
  a. Workflow scheduler systems to manage Apache Hadoop jobs
     - Doesn’t replace your scheduler
  b. Oozie workflow jobs are DAGs or Directed Graphs
  c. Oozie coordinator jobs are recurrent Oozie workflow jobs that are triggered by frequency or data availability
  d. Supports MapReduce, Pig, Apache Hive, and Sqoop etc.

- Apache ZooKeeper
    a. Provides operational services for a Hadoop cluster group services
    b. Centralized service for:
        - Maintaining configuration information naming services
        - Providing distributed synchronization and proving group services
- Apache Flume
    a. Distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data.
    b. Real time loader for streaming your data into Hadoop
    c. Possible sources for Flume include Avro, files, and system logs
    d. Possible sinks include HDFS and HBase
- Cloudera Impala
    a. Open source massively parallel processing (MPP) SQL query engine for Apache Hadoop.
    b. Facilitates real-time querying of data in HDFS or HBase
        - Secret is: circumventing Map Reduce to directly access the data through a specialized distributed query engine, similar to commercial parallel RDBMSs.
- Apache Spark
    a. Multi-stage in-memory primitives provides performance up to 100 times faster for certain applications
    b. Allows user programs to load data into a cluster's memory and query it repeatedly → well-suited for machine learning.

3. Lecture 3: MapReduce (and a bit of Spark)

The motivation for MapReduce is the problem of ever-growing data. Bigger data means:

- More storage, more required processing power, the access & transport of lots of data.

The answer: bring computation to the data.

If you need to process transactional data that needs regular updating, use an RDBMS.
If you need to sweep through data, simple reporting, to organize data output, take computation to the data.

The MapReduce framework:

- User defines:
  - <key, value>
  - Mapper & reducer functions
- Hadoop handles the logistics
- Logistics have specific requirements
  - All key-value pairs with the same key need to be processed by the same reducer

The MapReduce flow:

- User defines a <u>map</u> function
- Hadoop distributes/replicates map() to data
  - Map() reads data and outputs <key,value>
    - The mapper transforms raw input data into key-value pairs where multiple values for the same key may occur
- Hadoop shuffles and groups <key,value> data
- User defines a <u>reduce</u> function
- Hadoop distributes groups to reducers()
  - Reduce() reads <key,value> and outputs your results
    - The reducer transforms all the key-value pairs with the same key into a single key with a single value

Some rules of thumb

- 1 mapper per data split
  a. E.g.: 64MB HDFS data block size, for a 500MB file: 500/64 = 8 mappers
- 1 reducer per computer core (best parallelism)
- There is a trade-off though: what is the number of output files? Processing time?
- You need good key-value properties
  a. Key-value simplicity
  b. Enabling reducers to get correct output, through shuffling & grouping
- In short: Good task decomposition:
  a. Mappers: simple and separable
  b. Reducers: easy consolidation

MapReduce Design Considerations

- Composite <keys>
- Extra info in <values>
- Cascade MapReduce jobs (in series after eachother)
  a. Use the output from the first reducer in the second mapper
- Bin keys into ranges
- Aggregate map output when possible

Potential limitations of MapReduce

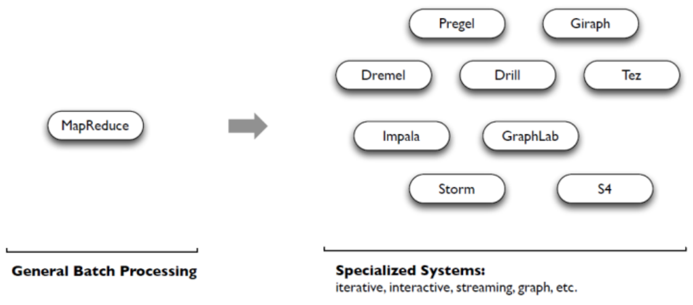
- Must fit in <key,value> paradigm
- MapReduce data not persistent
- Requires programming/debugging
- Not interactive

Beyond MapReduce are data access tools like Pig and HIVE. And of course Spark for interactivity & persistency.

Apache Spark

Shortcoming of MapReduce:

1. Force your pipeline into Map and Reduce steps
    a. What if you want other workflows? Like join of filter
    b. Solution by Spark: 20 operations, able to make combination between them.
2. Read from disk for each MapReduce job
    a. What if you have an iterative algorithm? Like machine learning
    b. Solution by Spark: in-memory caching of data, specified by the user.
3. Only native Java programming interface
    a. What if you want to use an other language?
    b. Solution by Spark: Native Python, Scala, R, interface. Interactive shells.

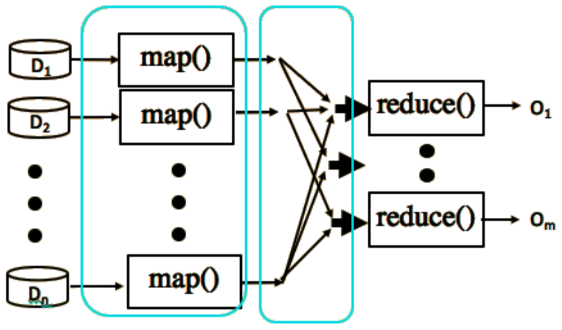


Solution is a new framework: same features of MapReduce and more.

Capable of reusing Hadoop ecosystem.

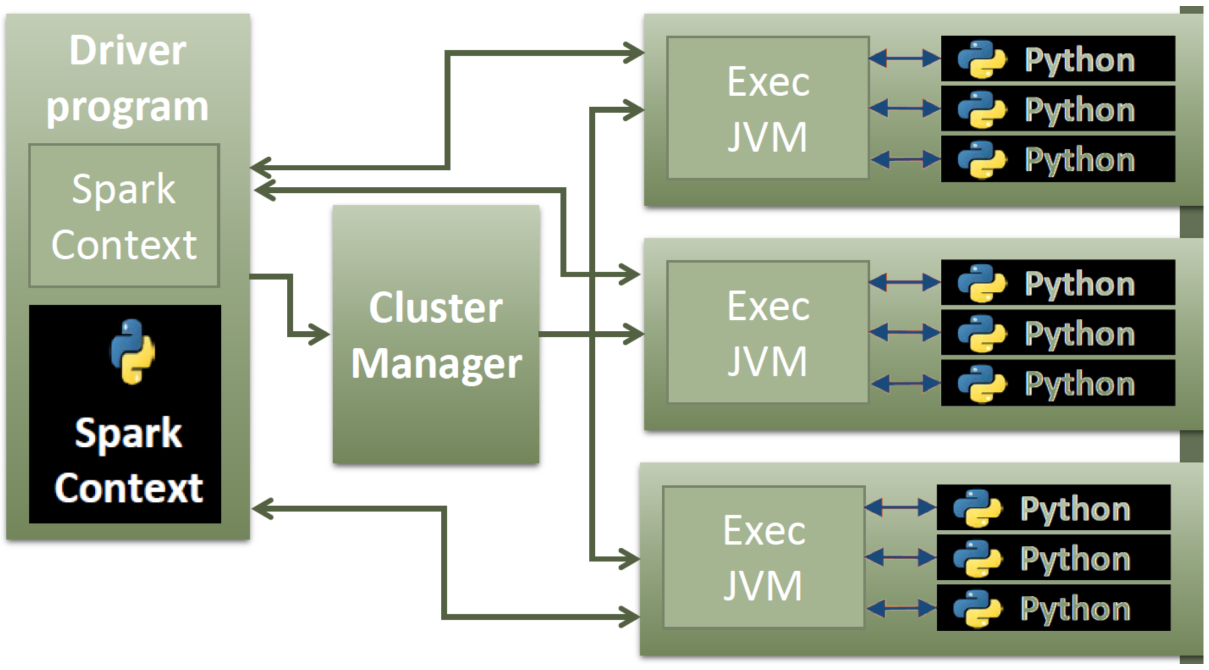
Spark. Unlike the specialized systems to the left. Spark's goal was to generalize MapReduce to support new apps within the same engine

4. Lecture 4: Spark – architecture & transformations



In MapReduce, there is one map() executor per worker node.

In Spark there is one python instance per task (worker node).



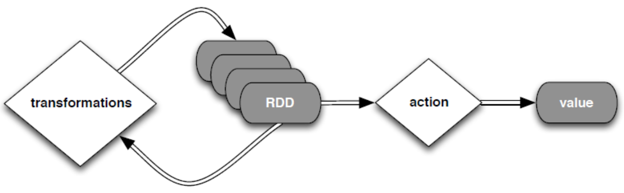
Exec JVM = Spark Executor Java Virtual Machine. This one communicates with the HDFS.

Cluster Manager: can be YARN/Standalone provision/Restart workers. Standalone when on a local VM for example. And YARN on Azure, or Amazon Elastic MapReduce (EMR)

Resilient Distributed Datasets

- RDDs are <u>immutable</u>, you cannot change just a chunk of them
- Spark works with <u>partitions</u>, not just fixed line by line processing
- Transformations are <u>lazy</u>

RDDs are created from other data sources or by copying and transforming another RDD.



RDDs track history of each partition, re-run.

Narrow Transformations

Since RDDs are immutable, you can never modify a RDD in place. You have to transform an RDD to another RDD. Which is a lazy process.

Some of the available transformations:

- map(func)
  - **a.** Apply function to each element of RDD
- flatMap(func)
  - **a.** Map then flatten output → that is without arrays
- filter(func)
  - **a.** Keep only elements where func is *true*
- sample(withReplacement, fraction, seed)
  - a. Get a random data fraction
- coalesce(numPartitions)
  - a. Merge partitions to reduce them to numPartitions

Examples:

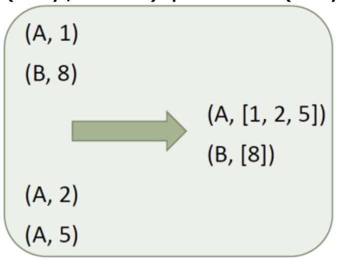
1. map(func)
   a. Define a mapper function:
   ```
   def lower(line):
     return line.lower()
   ```
   Here you define the function 'lower'.

   b. Invoke the transformation:
   ```
   lower_text_RDD = text_RDD.map(lower)
   ```
   Here you invoke the function 'lower' on a RDD: 'text_RDD'. Puts the results in another RDD: 'lower_text_RDD'.
2. filter(func)
   a. Define a function to filter out all words which start with an "a".
   ```
   def starts_with_a(word):
      return word.lower().startswith("a")
   ```
   b. Invoke the transformation:
   ```
   words_RDD.filter(starts_with_a).collect()

     Out[]:[u'A', u'ago', u'a', u'away']
   ```
   c. Watch out: filter() can result in unevenly distributed partitions.

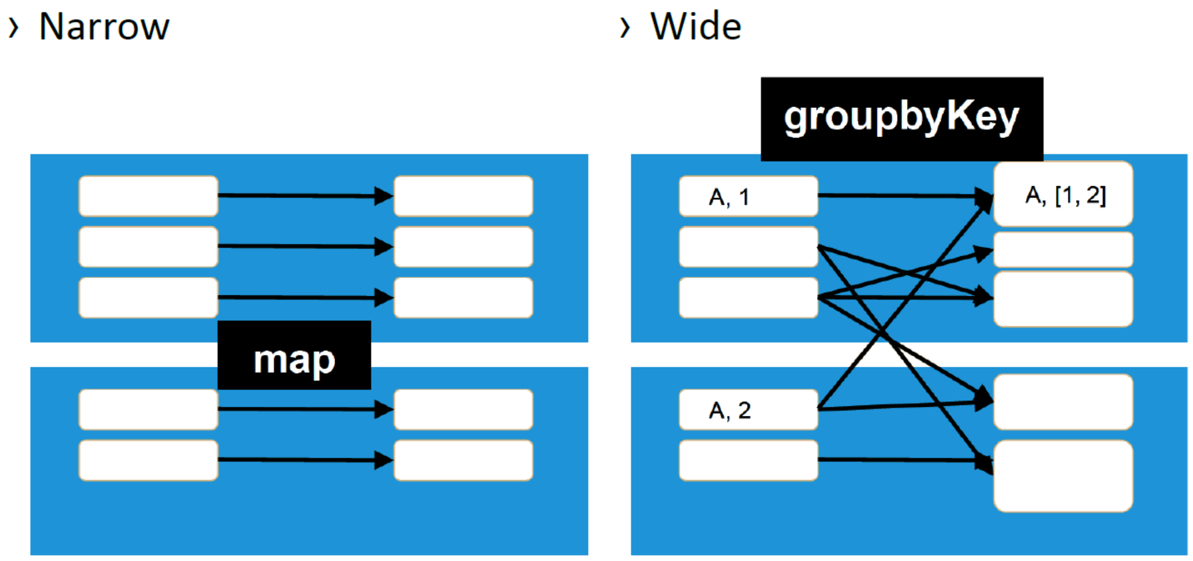
5. Lecture 5: More Spark Transformations

Wide transformations are for example:

- groupByKey()
    a. (Key,Value) pairs → (Key, iterable of all values)

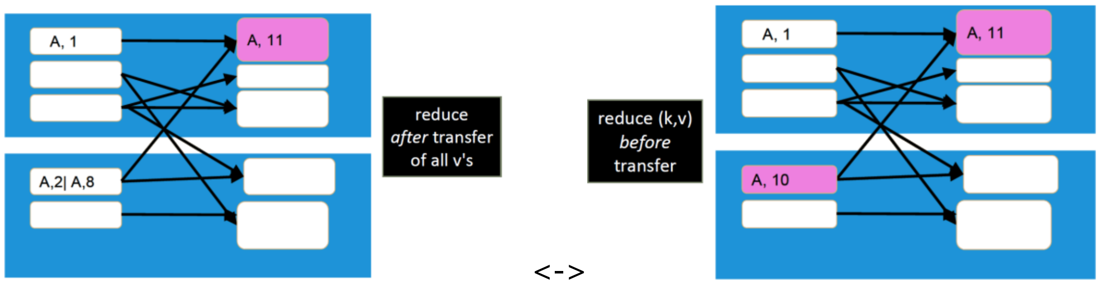


    b. Code: pairs_RDD.groupByKey().collect()
- reduceByKey(func)
    a. (Key,Value) pairs → (Key, result of reduction by func on all values)
- repartition(numPartitions)
    a. similar to *coalesce*, shuffles all data to increase or decrease number of partitions to numPartitions

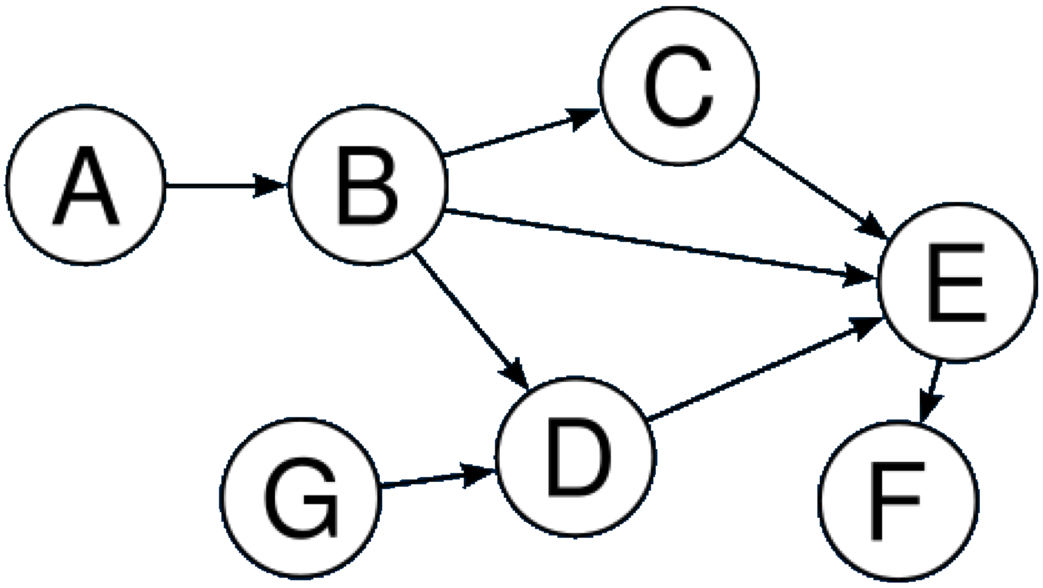


When doing wide transformations, like groupByKey, you do a shuffle from RDD to RDD. This shuffle is a global redistribution of data and therefore has a high impact on performance.

You should try to avoid shuffles, by knowing which operations cause it. Is it necessary to do this operation? Example for groupByKey(), when you plan to call reduce later in the pipeline, it is better to use reduceByKey instead. Less data must be redistributed if it is already reduced before transforming it to another RDD.


A, 1
A, 11
A,2| A,8
reduce *after* transfer of all v's
reduce (k,v) *before* transfer
A, 1
A, 11
A, 10
<->

Directed Acyclic Graph (DAG) Scheduler

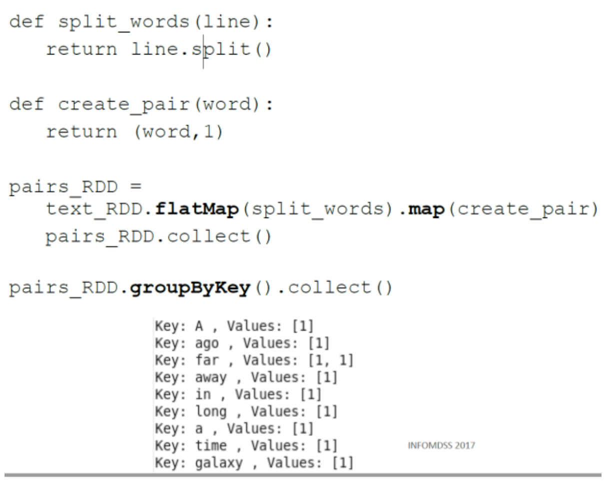

$$\begin{tikzcd}
 & & C \arrow[rd] & \\
A \arrow[r] & B \arrow[ru] \arrow[rr] \arrow[rd] & & E \arrow[d] \\
G \arrow[r] & D \arrow[rru] & & F
\end{tikzcd}$$

Nodes (A, B, C, D, E, F, & G) are RDDs. Arrows are transformations.

These Directed Acyclic Graph Schedulers are made to track dependencies:

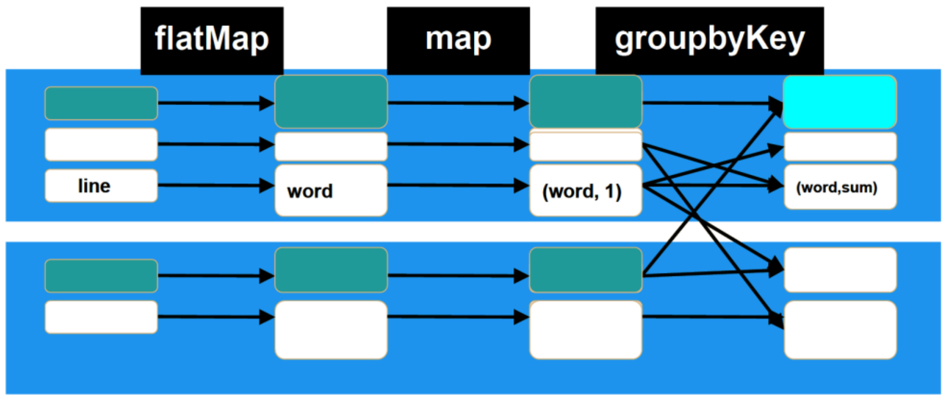
- Lineage: the data lifecycle, describe what happens to data as it goes through diverse processes
- Provenance: the origin of the data

Example:

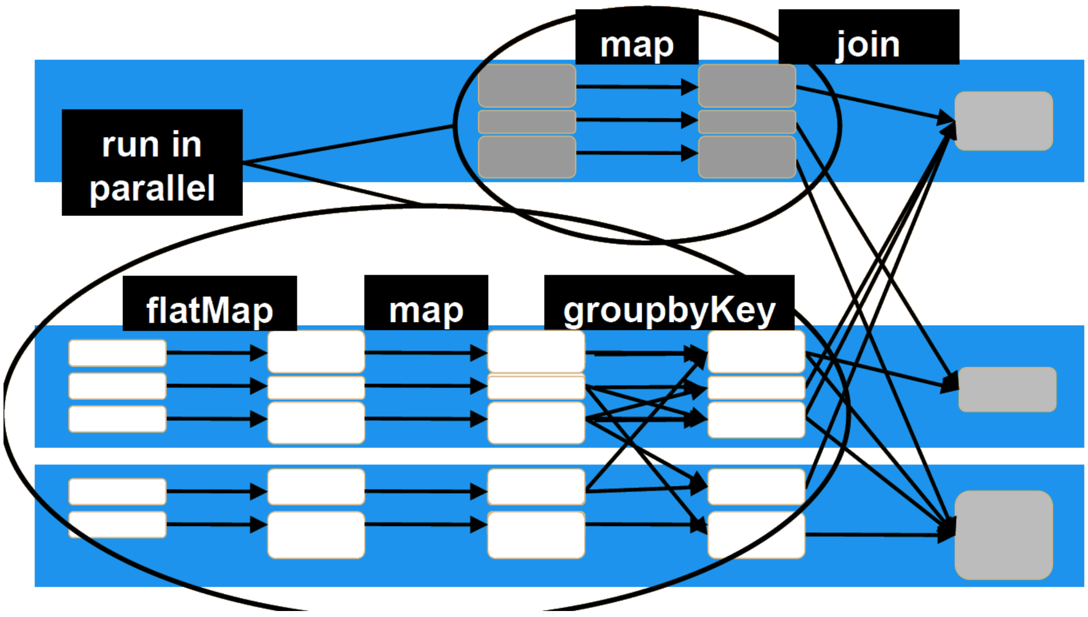


```
def split_words(line):
   return line.split()

def create_pair(word):
   return (word,1)

pairs_RDD =
   text_RDD.flatMap(split_words).map(create_pair)
   pairs_RDD.collect()

pairs_RDD.groupByKey().collect()
```

```
Key: A , Values: [1]
Key: ago , Values: [1]
Key: far , Values: [1, 1]
Key: away , Values: [1]
Key: in , Values: [1]
Key: long , Values: [1]
Key: a , Values: [1]
Key: time , Values: [1]
Key: galaxy , Values: [1]
```

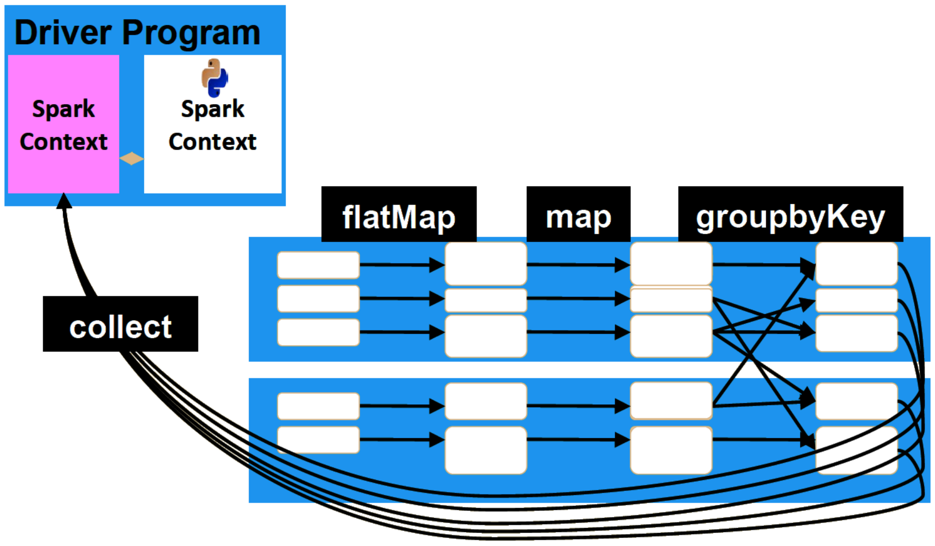


Another example, Spark DAG of transformations with join:



Actions

Actions are final stages of workflow. They trigger the execution of the DAG (since transformations are lazy, they are only performed when an action is required). The results are returned to the Driver or writes to HDFS.



Collect is the action.

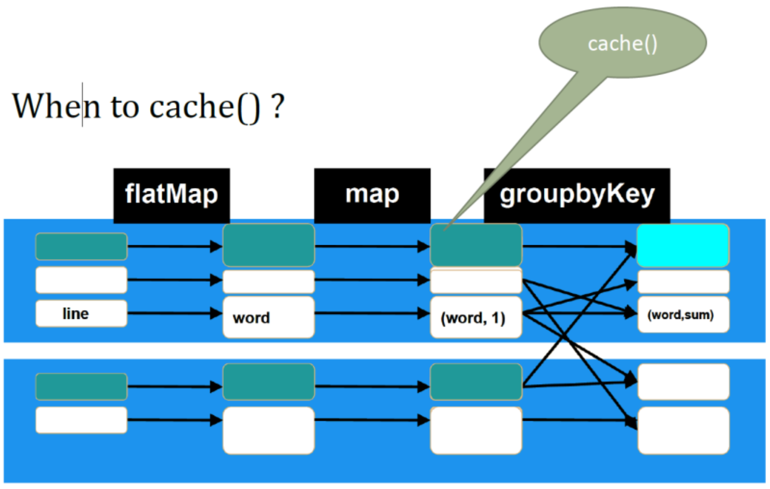
Different sorts of actions:

- collect()
  a. Copy all elements to the driver
- take(n)
  a. Copy first n elements
- reduce(func)
  a. Aggregate elements with func (takes 2 elements, returns 1)
- saveAsTextFile(filename)
  a. Save to local file or HDFS

Caching
By default, every job that is re-processed is re-processed from the HDFS. For some jobs that must be re-run, you want that to happen with caching. When?

- Generally, **not** the input data
- For validation and cleaning
- Cache for iterative algorithm

How? Most commonly in the memory, rarely on the disk. Both if you have heavy calculations. You mark the RDD with *.cache()*, if you have big data sets, you use *persist()*. It is raw storage, which if fast and simple.



Broadcast variables

- Is a large variable, used in all nodes
    - For example: a large configuration dictionary or lookup table.
- Transfer just once per Executor
- Efficient peer-to-peer transfer

```
config=sc.broadcast({"order":3,"filter":True})
config.value
```

Accumulator

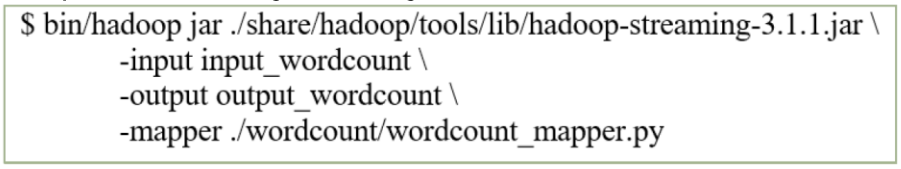
- Common pattern of accumulating (collecting) to a variable across the cluster
- Write-only on nodes

```
accum = sc.accumulator(0)

def test_accum(x):
    accum.add(x)

sc.parallelize([1,2,3,4]).foreach(test_accum)
accum.value

Out[]: ??? 10
```

1 + 2 +3 + 4 = 10

Example questions:

- What command can be used to display the content of a file?
  a. Cat
  b. Li
  c. Echo
  d. Ls
- Analyse the following code fragment. What does this command do?



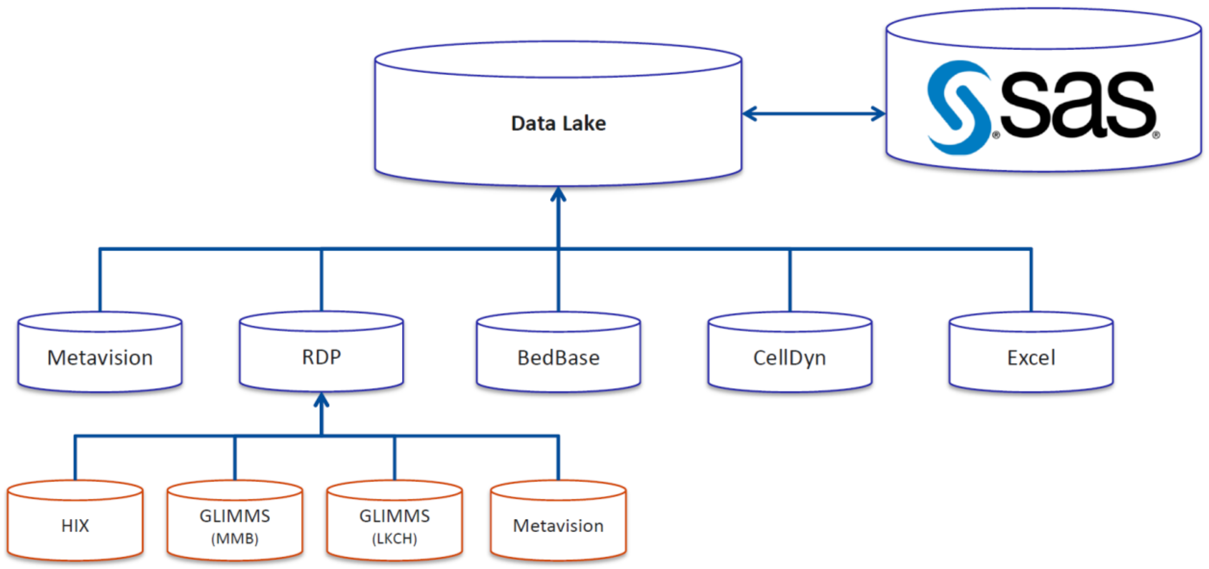
```
$ bin/hadoop jar ./share/hadoop/tools/lib/hadoop-streaming-3.1.1.jar \
        -input input_wordcount \
        -output output_wordcount \
        -mapper ./wordcount/wordcount_mapper.py
```

  a. It processes an input file with Hadoop
  b. Nothing; it will report an error stating that there is no jar file
  c. Nothing; it will report that there is no reducer
  d. It will produce three output files in output_wordcount
- Why do Davenport & Patil (2012) refer to Data scientist as being the Sexiest Job of the 21st century?
  a. Data scientist want to build things, not just give advice in the role of consultant
  b. The shortage of data scientists is becoming a serious constraint in some sectors
  c. Data scientist today are akin to the Wall Street "quants" of the 1980s and 1990s
  d. All of the above
- What is generally considered to be part of the Apache Basic Hadoop Modules?
  a. YARN
  b. Impala
  c. MapReduce
  d. HDFS

6. Guest lecture Neonatology

At the neonatology they want to use data to develop algorithms for prediction and a real-time alert system → individualized care.

Therefore, they are connecting data:



RDP = Research database

Research projects:

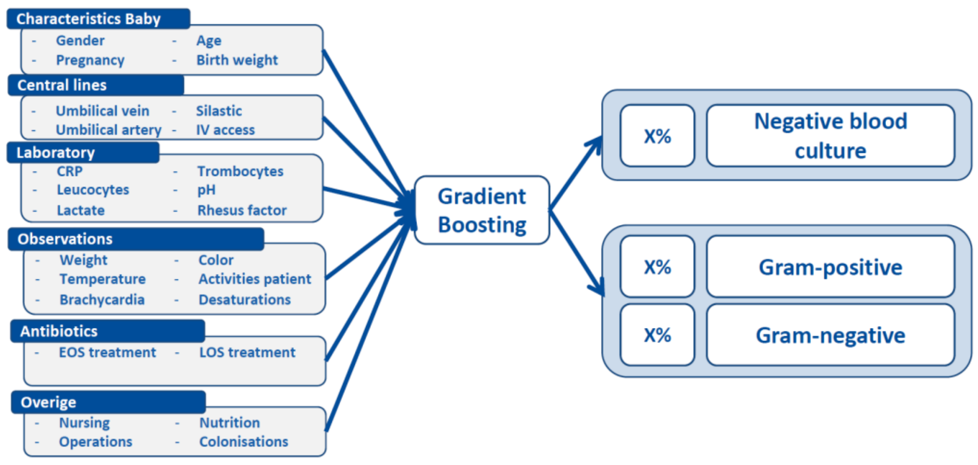
ADAM (Applied Data Analytics in Medicine):

- Electronic Patient Record
- Patient Portal
- eHealth
- Applied Data Analytics
- Digital Hospital

Big Data for Small Babies:

- Investigating the onset and incidence of sepsis in preterm infants
- Prediction vs. therapy advice
    - For example: when to apply antibiotics?

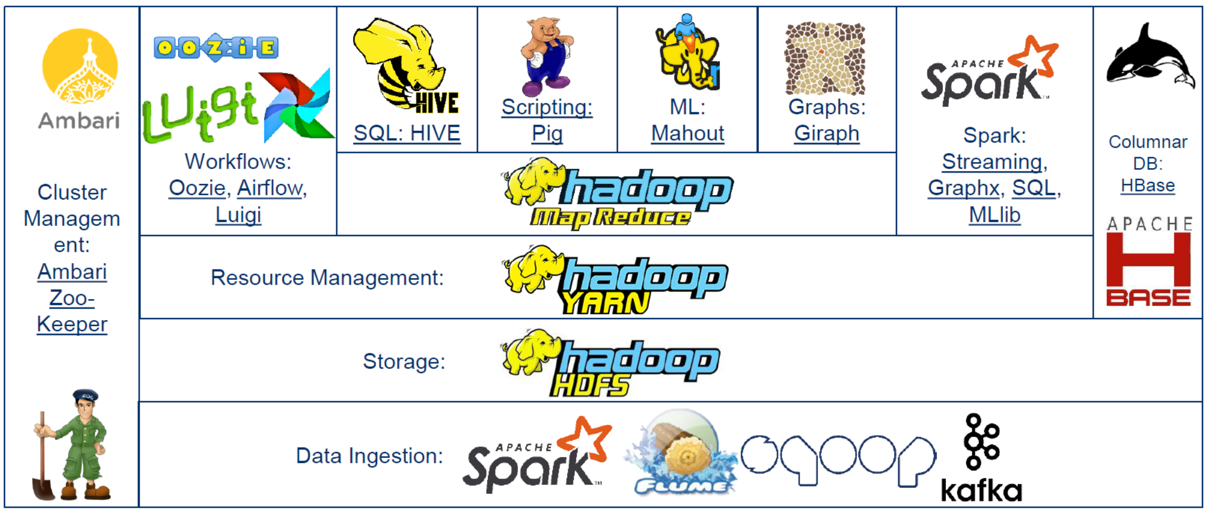
They were already able to do some gradient boosting (machine learning) on variables of babies to predict negative blood cultures.


Characteristics Baby
- Gender - Age
- Pregnancy - Birth weight
Central lines
- Umbilical vein - Silastic
- Umbilical artery - IV access
Laboratory
- CRP - Trombocytes
- Leucocytes - pH
- Lactate - Rhesus factor
Observations
- Weight - Color
- Temperature - Activities patient
- Brachycardia - Desaturations
Antibiotics
- EOS treatment - LOS treatment
Overige
- Nursing - Nutrition
- Operations - Colonisations
Gradient Boosting
X% Negative blood culture
X% Gram-positive
X% Gram-negative

7. Guest lecture ORTEC

The difference between data scientists and data engineers:

- Data scientists create models and extracts business information using data
- Data engineers extract, transform, and load the data for analysis. Implements models for production.



Ambari
Cluster Management: Ambari Zoo-Keeper
Workflows: Oozie, Airflow, Luigi
SQL: HIVE
Scripting: Pig
ML: Mahout
Graphs: Giraph
Spark: Streaming, Graphx, SQL, MLlib
Columnar DB: HBase
APACHE HBASE
hadoop MapReduce
Resource Management: hadoop YARN
Storage: hadoop HDFS
Data Ingestion: Apache Spark, Flume, sqoop, kafka

Artifact design:

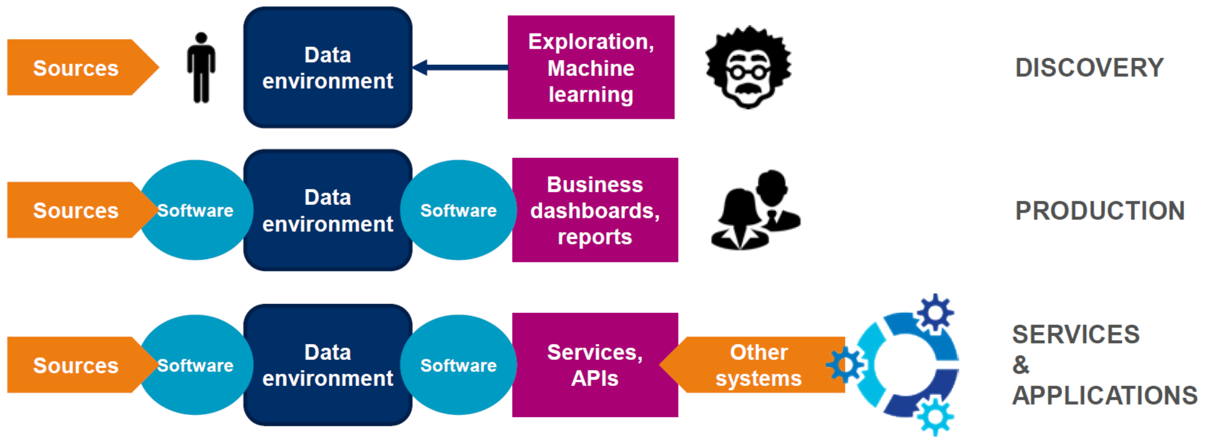
1. Define context and goals
2. Determine input
3. Estimate output
4. Determine processing
5. Evaluate results
6. Deploy outcomes

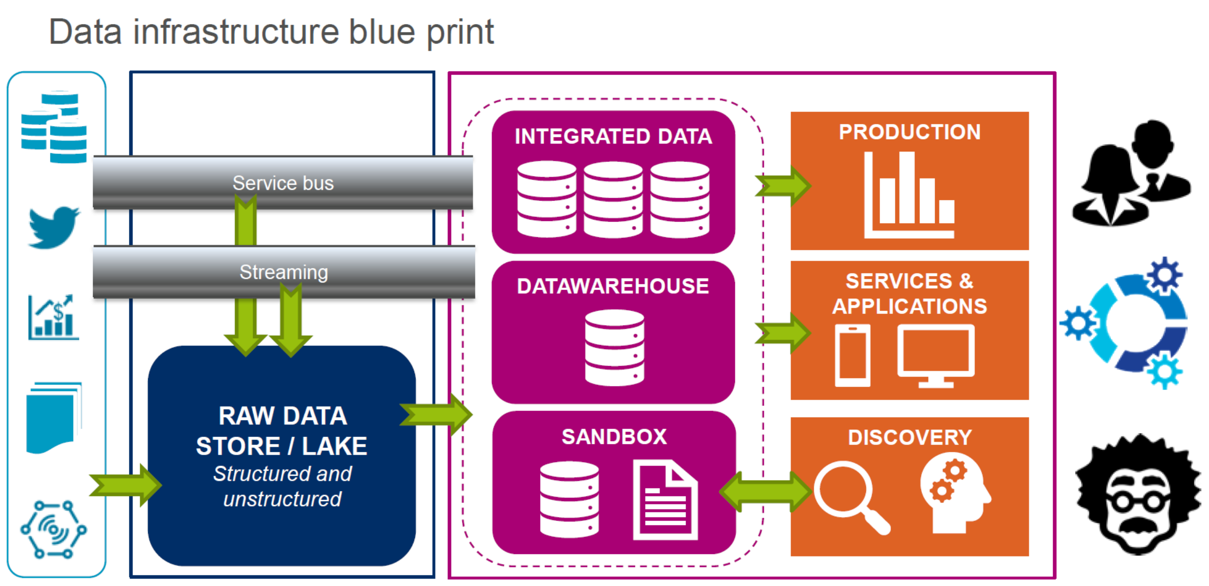
Define context and goals:

- Data processing objectives
- Available resources
- Model specifications

Type of application

Different flow, expectations, requirements





Determine input:

- Volume: a lot of data
- Velocity: fast or slow?
- Variety: is the data in different forms?
- Variability: does the data change?
- Veracity: is the data messy?
- Value: what is the data worth?

Getting the data, push or pull:

- Pull: pull data in your data store/lake/base.
    - Structured files
    - Direct DB access
    - API
    - Web scraping
- Push: get data pushed in your data store/lake/base.
    - Streaming or message bus
    - API

Determine processing:
About selecting the right tool for:

- Input processing
    a. Data type (structured, semi, or unstructured)
    b. Processing type (batch or stream)
    c. Data ingestion tools: Sqoop, Flume, Kafka, or Spark.
- Data transformations
    a. Type of processing (general-purpose, SQL, machine learning, graph)
    b. Data type
    c. Engine of choice (Hadoop or Spark)
    d. Language (Java, Scala, Python, R, PigLatin, HQL)
- Output processing
    a. Same as input processing
- Set up workflow
    a. Pick your processing engine (Hadoop, Spark, or both)
    b. Storage (local machine, azure, or amazon)
    c. Pick your workflow, based on the first two steps
    d. Workflow management: Oozie, Airflow, Luigi

Evaluate results:

- Performance:
    a. Results
    b. Timing & latency
    c. Costs
- Distributed system:
    a. Scalability
    b. Distribution transparency
    c. Resource sharing

Deploy outcomes:

- Deployment plan
- Conclusions & lessons learned

Proof of concept vs. production:

- Speed
- Data strategy
- Security
- Scalability
- Business continuity

These points are less important in the PoC, but become very important when going to production.

Important lessons from Stijn:

- Always make sure that you have a goal in mind, and document it.
- See data engineers and data scientists as a team, one cannot live without the other.