Distributed Computing with MapReduce

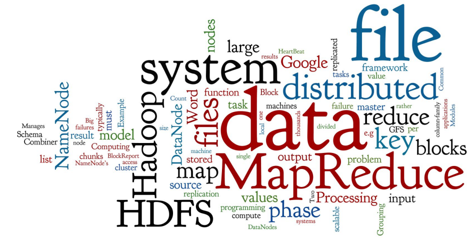
Lecture 2 of *NoSQL Databases* (PA195)

David Novak & Vlastislav Dohnal

Faculty of Informatics, Masaryk University, Brno

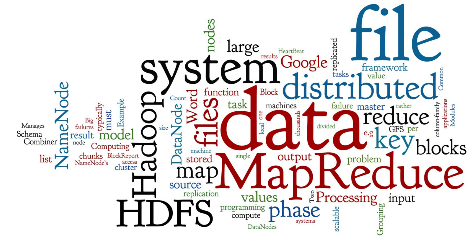
http://disa.fi.muni.cz/vlastislav-dohnal/teaching/nosql-databases-fall-2019/

Agenda



- Distributed Data Processing
- Google MapReduce
  - Motivation and History
  - Google File System (GFS)
  - MapReduce: Schema, Example, MapReduce Framework
- Apache Hadoop
  - Hadoop Modules and Related Projects
  - Hadoop Distributed File System (HDFS)
  - Hadoop MapReduce
- MapReduce in Other Systems

Agenda



- Distributed Data Processing
- Google MapReduce
  - Motivation and History
  - Google File System (GFS)
  - MapReduce: Schema, Example, MapReduce Framework
- Apache Hadoop
  - Hadoop Modules and Related Projects
  - Hadoop Distributed File System (HDFS)
  - Hadoop MapReduce
- MapReduce in Other Systems

Distributed Data Processing

What is the best way of doing distributed processing?

Centralized (and in memory)

Don't do it, if don't have to

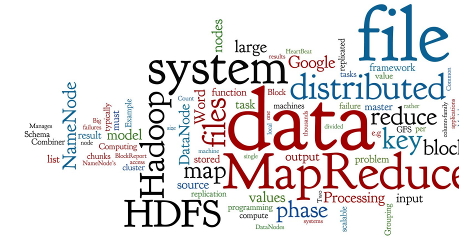
Big Data Processing

- Big Data analytics (or data mining)
  - need to process large data volumes quickly
  - want to use computing cluster instead of a super-computer

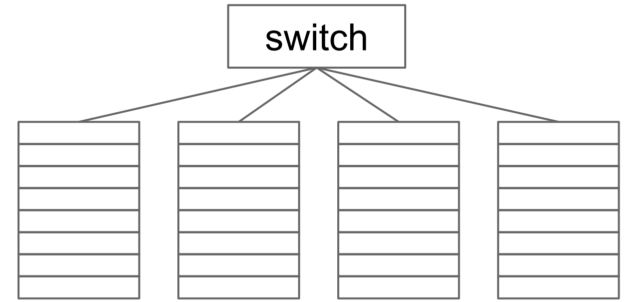
- Communication (sending data) between compute nodes is expensive

=> model of "moving the computing to data"

Big Data Processing II



Computing cluster architecture:

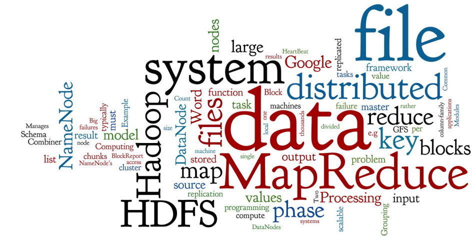


racks with compute nodes

- HW failures are rather rule than exception, thus
  1. Files must be stored redundantly
     - over different racks to overcome also rack failures
  2. Computations must be divided into independent tasks
     - that can be restarted in case of a failure

source: J. Leskovec, A. Rajaraman, and J. D. Ullman, Mining of Massive Datasets. 2014.

Agenda

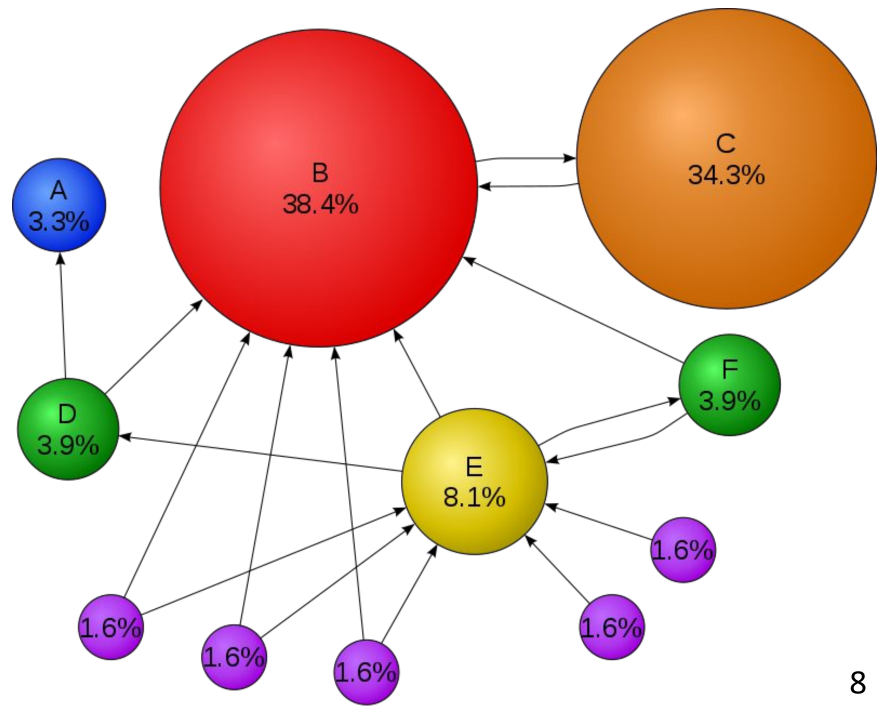


- Distributed Data Processing
- Google MapReduce
  - Motivation and History
  - Google File System (GFS)
  - MapReduce: Schema, Example, MapReduce Framework
- Apache Hadoop
  - Hadoop Modules and Related Projects
  - Hadoop Distributed File System (HDFS)
  - Hadoop MapReduce
- MapReduce in Other Systems

PageRank

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is.

The underlying assumption is that more important websites are likely to receive more links from other websites.



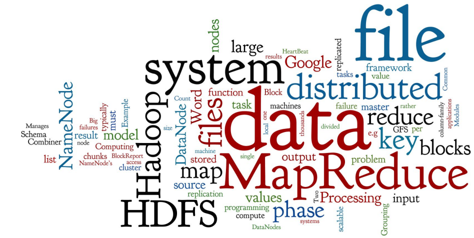
https://en.wikipedia.org/wiki/PageRank

MapReduce: Origins

- In 2003, Google had the following problem:
  1. How to rank tens of billions of webpages by their “importance” (PageRank) in a “reasonable” amount of time?
  2. How to compute these rankings efficiently when the data is scattered across thousands of computers?

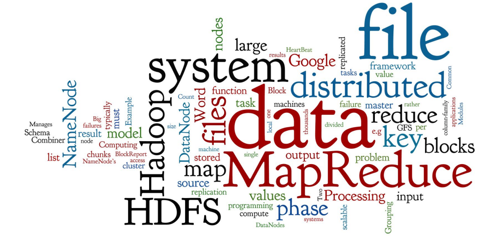
- Additional factors:
  1. Individual data files can be enormous (terabyte or more)
  2. The files were rarely updated
     - the computations were read-heavy, but not very write-heavy
     - If writes occurred, they were appended at the end of the file

Google Solution



- Google found the following solutions:
  - Google File System (GFS)
    - A distributed file system
  - MapReduce
    - A programming model for distributed data processing

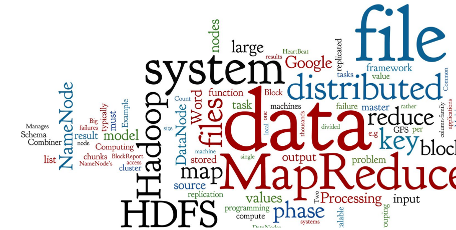
Google File System (GFS)



- Files are divided into chunks (typically 64 MB)
  - The chunks are replicated at three different machines
    - ...in an “intelligent” fashion, e.g. never all on the same computer rack
  - The chunk size and replication factor are tunable

- One machine is a master, the other chunkservers
  - The master keeps track of all file metadata
    - mappings from files to chunks and locations of the chunks
  - To find a file chunk, client queries the master, and then contacts the relevant chunkservers
  - The master’s metadata files are also replicated

GFS: Schema



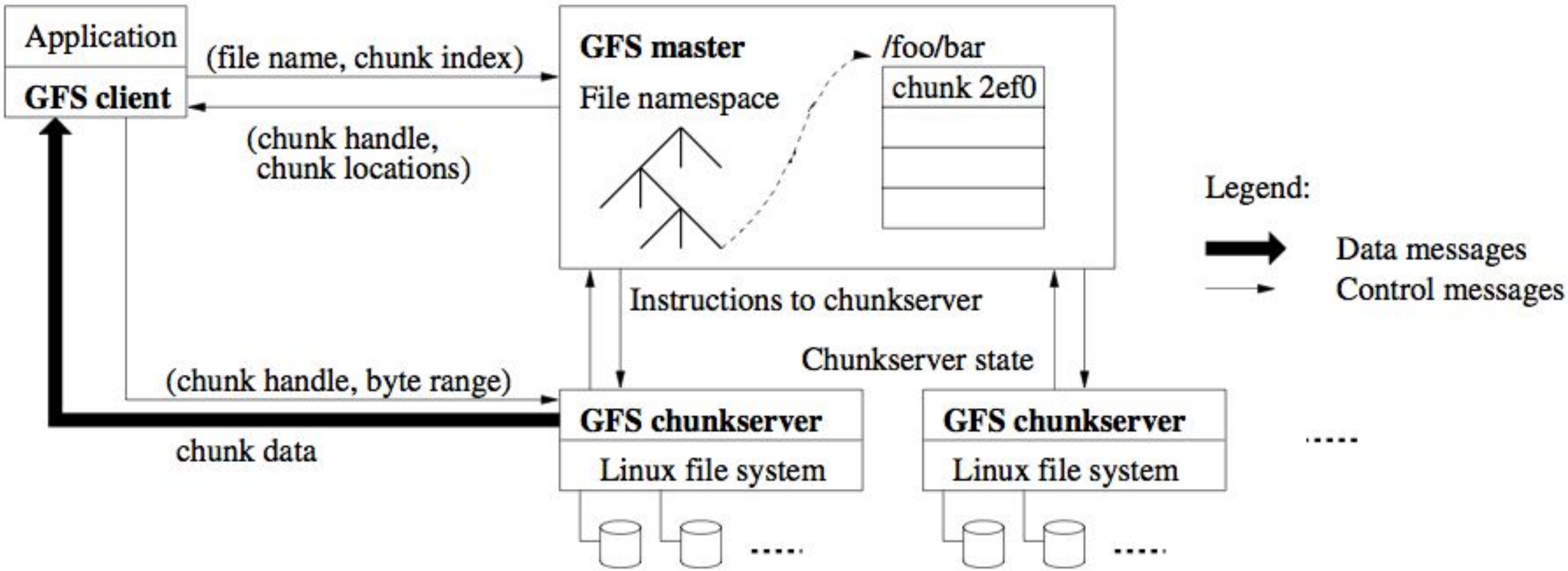
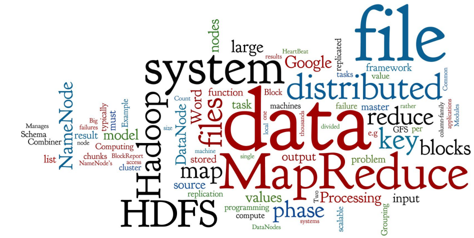


Figure 1: GFS Architecture

source: http://dl.acm.org/citation.cfm?id=945450

MapReduce (1)

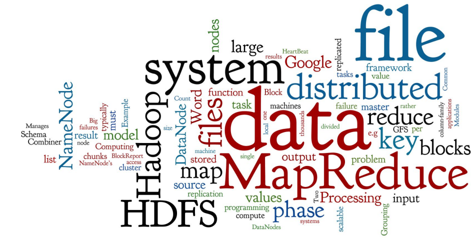


- MapReduce is a programming model sitting on the top of a Distributed File System
  - Originally: no data model – data stored directly in files

- A distributed computational task has three phases:
  1. The map phase: data transformation
  2. The grouping phase
     - done automatically by the MapReduce Framework
  3. The reduce phase: data aggregation

- User must define only map & reduce functions

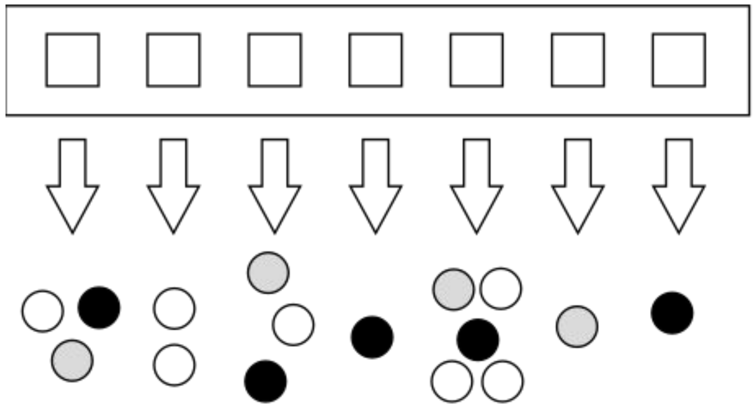
Map



- Map function simplifies the problem in this way:
  - Input: a single data item (e.g. line of text) from a data file
  - Output: zero or more (key, value) pairs

- The keys are not typical "keys":
  - They do not have to be unique
  - A map task can produce several key-value pairs with the same key (even from a single input)

- Map phase applies the map function to all items

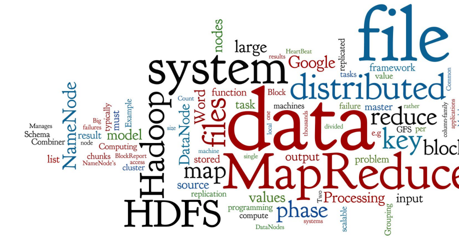


input data

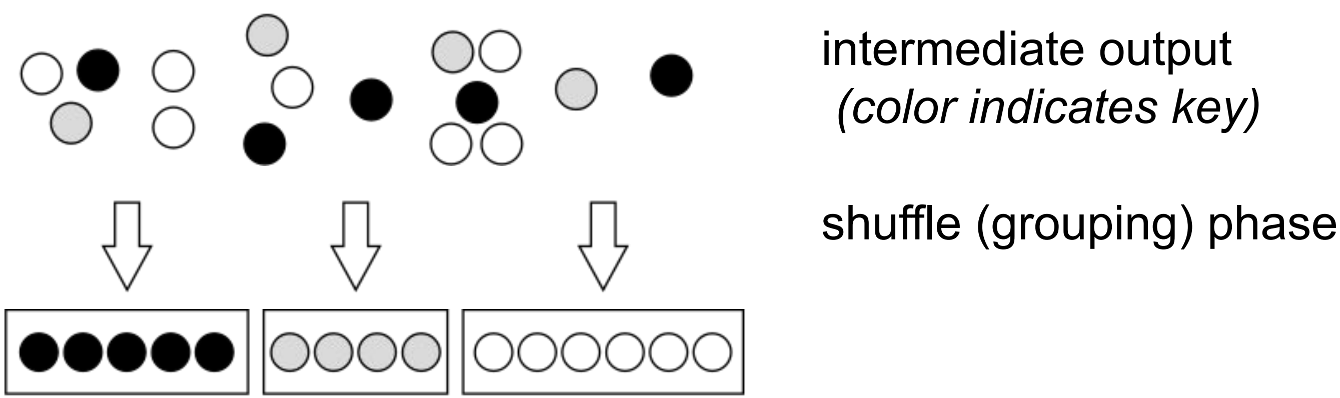
map function

output data
(color indicates key)

Grouping Phase

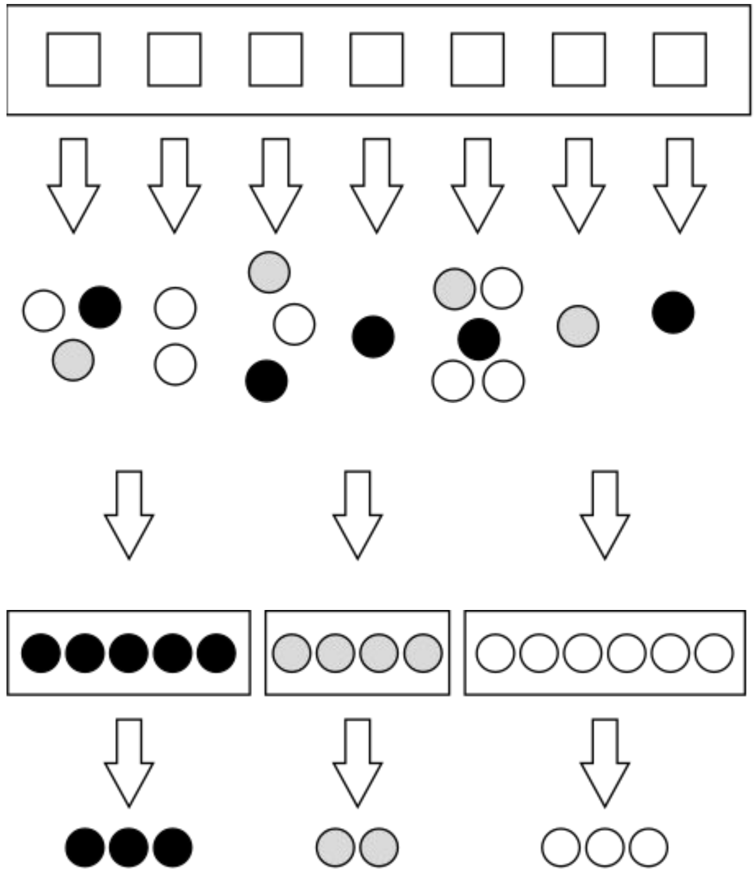


- Grouping (Shuffling): The key-value outputs from the map phase are grouped by key
  - Values sharing the same key are sent to the same reducer
  - These values are consolidated into a single list (key, list)
    - This is convenient for the reduce function
  - This phase is realized by the MapReduce framework



Reduce Phase

- Reduce: combine the values for each key
  - to achieve the final result(s) of the computational task
  - Input: (key, value-list)
    - value-list contains all values generated for given key in the Map phase
  - Output: (key, value-list)
    - zero or more output records



input data

map function

intermediate output
(color indicates key)

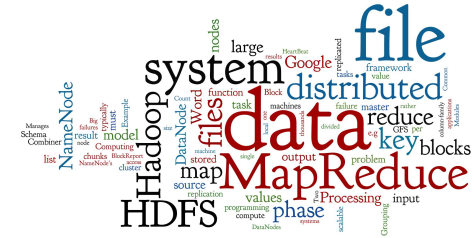
shuffle (grouping) phase

input data

reduce function

output data

Example: Word Count

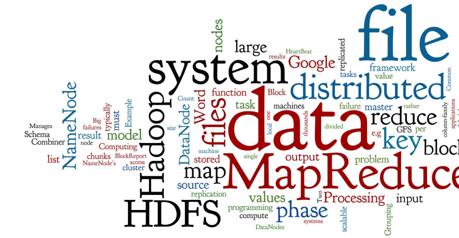


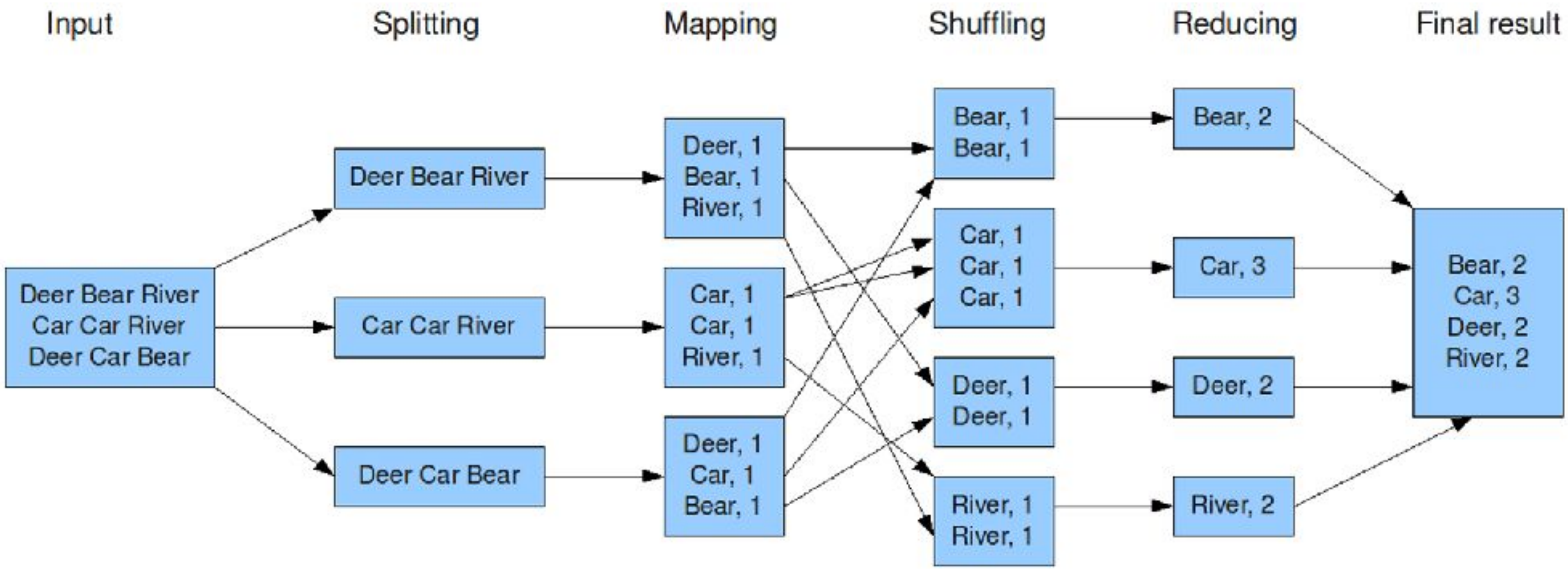
Task: Calculate word frequency in a set of documents

```
map(String key, Text value):
  // key: document name (ignored)
  // value: content of document (words)
foreach word w in value:
    emitIntermediate(w, 1);
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
int result = 0;
foreach v in values:
    result += v;
emit(key, result);
```

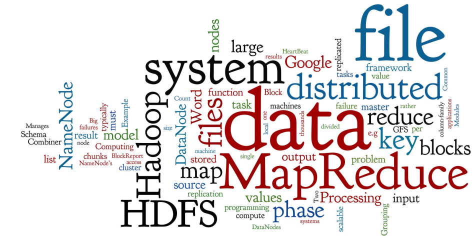
Example: Word Count (2)





source: http://www.cs.uml.edu/~jlu1/doc/source/report/MapReduce.html

MapReduce: Combiner



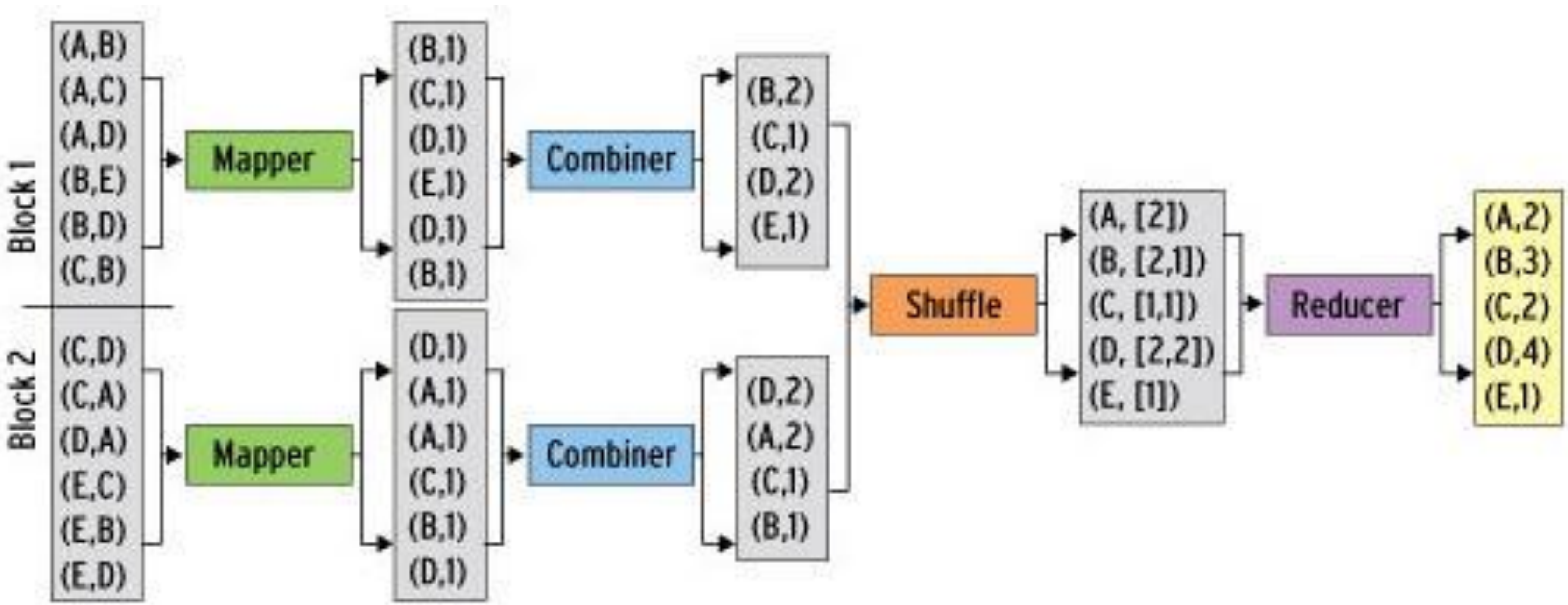
- If the reduce function is commutative & associative
    - The values can be combined in any order and combined per partes (grouped)
        - with the same result (e.g. Word Counts)
- ...then we can do "partial reductions"
    - Apply the same reduce function right after the map phase, before shuffling and redistribution to reducer nodes
- This (optional) step is known as the combiner
    - Note: it's still necessary to run the reduce phase

Example: Word Count, Combiner

Task: Calculate word frequency in a set of documents

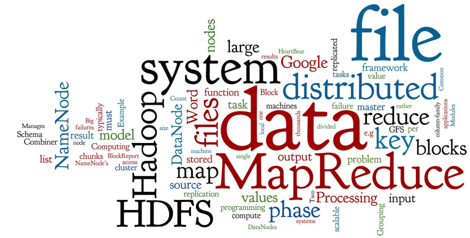
```
combine(String key, Iterator values):
  // key: a word
  // values: a list of local counts
int result = 0;
foreach v in values:
    result += v;
emit(key, result);
```

Example: Word Count with Combiner



source: http://www.admin-magazine.com/HPC/Articles/MapReduce-and-Hadoop

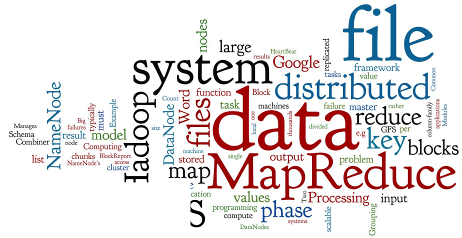
MapReduce Framework

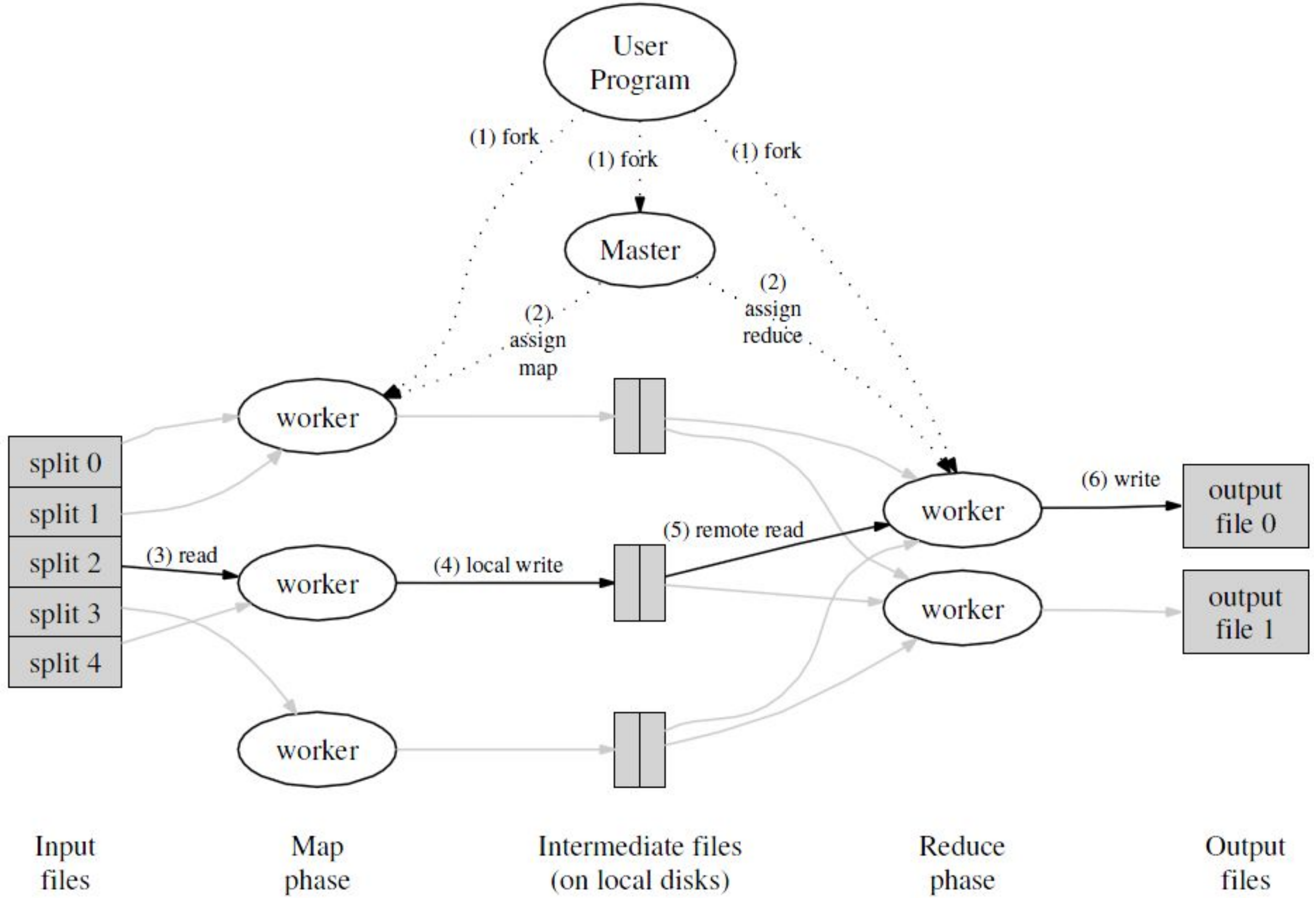


- MapReduce framework takes care about
  - Distribution and parallelizing of the computation
  - Monitoring of the whole distributed task
  - The grouping phase
    - putting together intermediate results
  - Recovering from any failures

- User must define only map & reduce functions
  - but can define also other additional functions (see below)

MapReduce Framework (2)





source: Dean, J. & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters

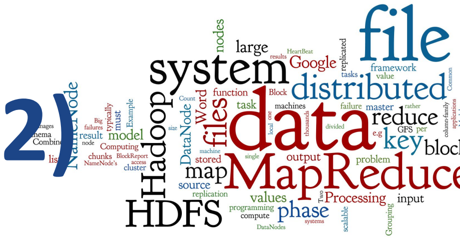
MapReduce Framework: Details

1. Input reader (function)
   - defines how to read data from underlying storage

2. Map (phase)
   - master node prepares *M* data splits and *M* idle Map tasks
   - pass individual splits to the Map tasks that run on workers
   - these map tasks are then running
   - when a task is finished, its intermediate results are stored

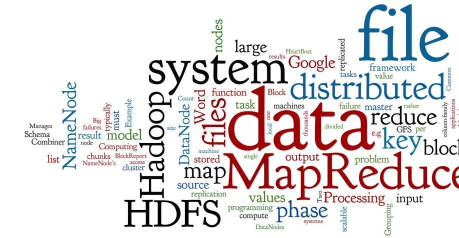
3. Combiner (function, optional)
   - combine local intermediate output from the Map phase

MapReduce Framework: Details (2)



4. Partition (function)
   - to partition intermediate results for individual Reducers
5. Comparator (function)
   - sort and group the input for each Reducer
6. Reduce (phase)
   - master node creates *R* idle Reduce tasks on workers
   - Partition function defines a data batch for each reducer
   - each Reduce task uses Comparator to create key-values pairs
   - function Reduce is applied on each key-values pair
7. Output writer (function)
   - defines how the output key-value pairs are written out

MapReduce: Example II



Task: Calculate graph of web links

- what pages reference (<a href=””>) each page (backlinks)

```
map(String url, Text html):
  // url: web page URL
  // html: HTML text of the page (linearized HTML tags)
foreach tag t in html:
    if t is <a> then:
        emitIntermediate(t.href, url);
```

```
reduce(String key, Iterator values):
  // key: target URLs
  // values: a list of source URLs
emit(key, values);
```

Example II: Result

Input: `(page_URL, HTML_code)`

```
("http://cnn.com", "<html>...<a href="http://cnn.com">link</a>...</html>")
("http://ihned.cz", "<html>...<a href="http://cnn.com">link</a>...</html>")
("http://idnes.cz",
   "<html>...<a href="http://cnn.com">x</a>...
       <a href="http://ihned.cz">y</a>...<a href="http://idnes.cz">z</a>
    </html>")
```

Intermediate output after Map phase:

```
("http://cnn.com", "http://cnn.com")
("http://cnn.com", "http://ihned.cz")
("http://cnn.com", "http://idnes.cz")
("http://ihned.cz", "http://idnes.cz")
("http://idnes.cz", "http://idnes.cz")
```

Intermediate result after shuffle phase (the same as output after Reduce phase):

```
("http://cnn.com", ["http://cnn.com", "http://ihned.cz", "http://idnes.cz"] )
("http://ihned.cz", [ "http://idnes.cz" ])
("http://idnes.cz", [ "http://idnes.cz" ])
```

MapReduce: Example III

Task: What are the lengths of words in the input text

- output = how many words are in the text for each length

```
map(String key, Text value):
  // key: document name (ignored)
  // value: content of document (words)
foreach word w in value:
    emitIntermediate(length(w), 1);
```

```
reduce(Integer key, Iterator values):
  // key: a length
  // values: a list of counts
int result = 0;
foreach v in values:
    result += v;
emit(key, result);
```

MapReduce: Features

- MapReduce uses a "shared nothing" architecture
  - Nodes operate independently, sharing no memory/disk
  - Common feature of many NoSQL systems

- Data partitioned and replicated over many nodes
  - Pro: Large number of read/write operations per second
  - Con: Coordination problem – which nodes have my data, and when?

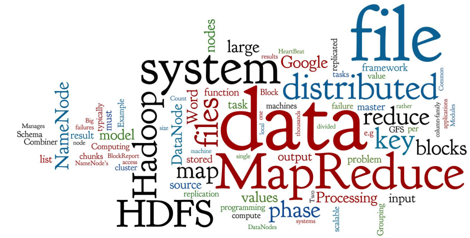
Applicability of MapReduce

- MR is applicable if the problem is parallelizable

- Two problems:
  1. The programming model is limited
     (only two phases with a given schema)
  2. There is no data model - it works only on “data chunks”

- Google’s answer to the 2nd problem was BigTable
  - The first column-family system (2005)
  - Subsequent systems: HBase (over Hadoop), Cassandra,...

Agenda



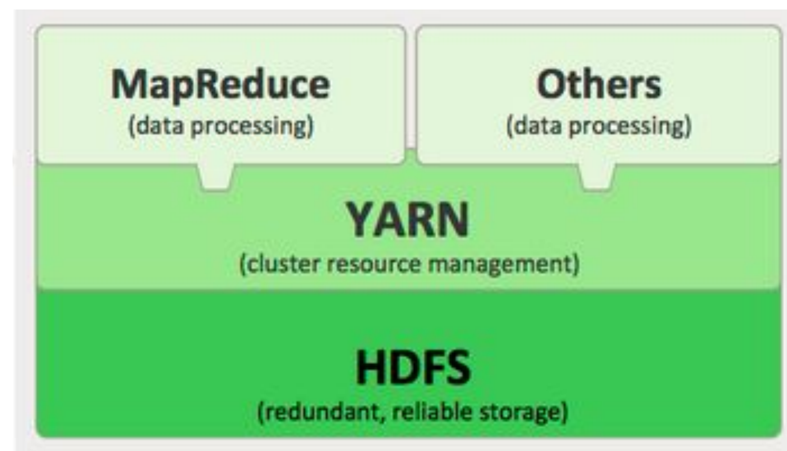
- Distributed Data Processing
- Google MapReduce
  - Motivation and History
  - Google File System (GFS)
  - MapReduce: Schema, Example, MapReduce Framework
- Apache Hadoop
  - Hadoop Modules and Related Projects
  - Hadoop Distributed File System (HDFS)
  - Hadoop MapReduce
- MapReduce in Other Systems

Apache Hadoop

- Open-source software framework
  - Implemented in Java
- Able to run applications on large clusters of commodity hardware
  - Multi-terabyte data-sets
  - Thousands of nodes
- Derived from the idea of Google's MapReduce and Google File System

Hadoop: Modules

- Hadoop Common
  - Common support functions for other Hadoop modules
- Hadoop Distributed File System (HDFS)
  - Distributed file system
  - High-throughput access to application data
- Hadoop YARN
  - Job scheduling and cluster resource management
- Hadoop MapReduce
  - YARN-based system for parallel data processing

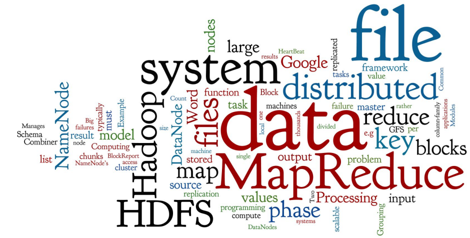


source: https://goo.gl/NPuuJr

HDFS (Hadoop Distributed File System)

- Free and open source
- Cross-platform (pure Java)
  - Bindings for non-Java programming languages
- Highly scalable
- Fault-tolerant
  - Idea: “failure is the norm rather than exception”
    - A HDFS instance may consist of thousands of machines and each can fail
  - Detection of faults
  - Quick, automatic recovery
- Not the best in efficiency

HDFS: Data Characteristics



- Assumes:
  - Streaming data access
    - reading the files from the beginning till the end
  - Batch processing rather than interactive user access
- Large data sets and files
- Write-once / read-many
  - A file once created does not need to be changed often
  - This assumption simplifies coherency

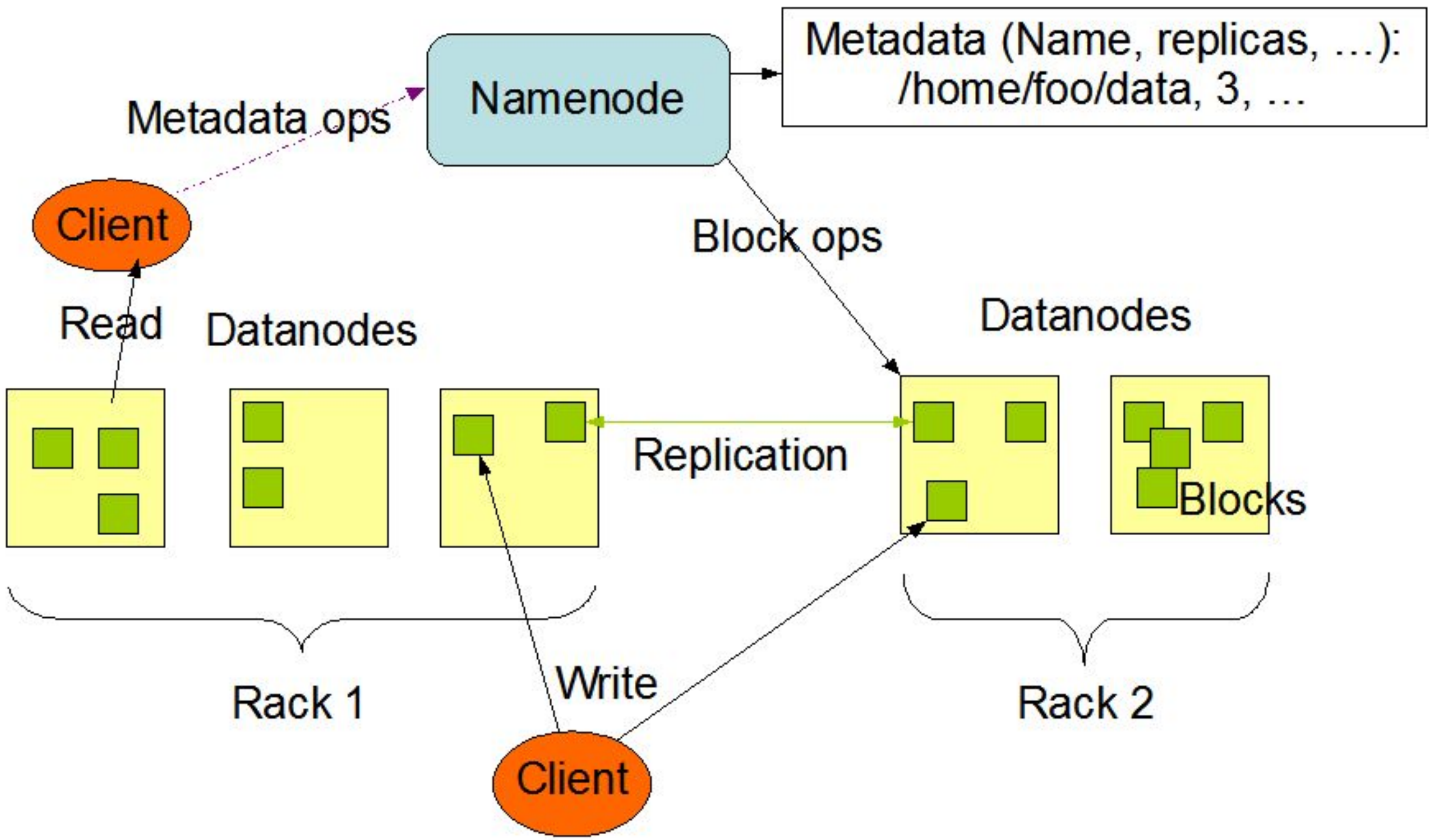
- Optimal applications for this model: MapReduce, web-crawlers, data warehouses, …

HDFS: Basic Components

- Master/slave architecture
- HDFS exposes file system namespace
  - File is internally split into blocks
- NameNode - master server
  - Manages the file system namespace
    - Opening/closing/renaming files and directories
    - Regulates file accesses
  - Determines mapping of blocks to DataNodes
- DataNode - manages file blocks
  - Block read/write/creation/deletion/replication
  - Usually one per physical node

HDFS: Schema

HDFS Architecture



HDFS: NameNode

- NameNode has a structure called FsImage
  - Entire file system namespace + mapping of blocks to files + file system properties
  - Stored in a file in NameNode's local file system
  - Designed to be compact
    - Loaded in NameNode's memory (4 GB of RAM is sufficient)

- NameNode uses a transaction log called EditLog
  - to record every change to the file system's meta data
    - E.g., creating a new file, change in replication factor of a file, ..
  - EditLog is stored in the NameNode's local file system

HDFS: DataNode

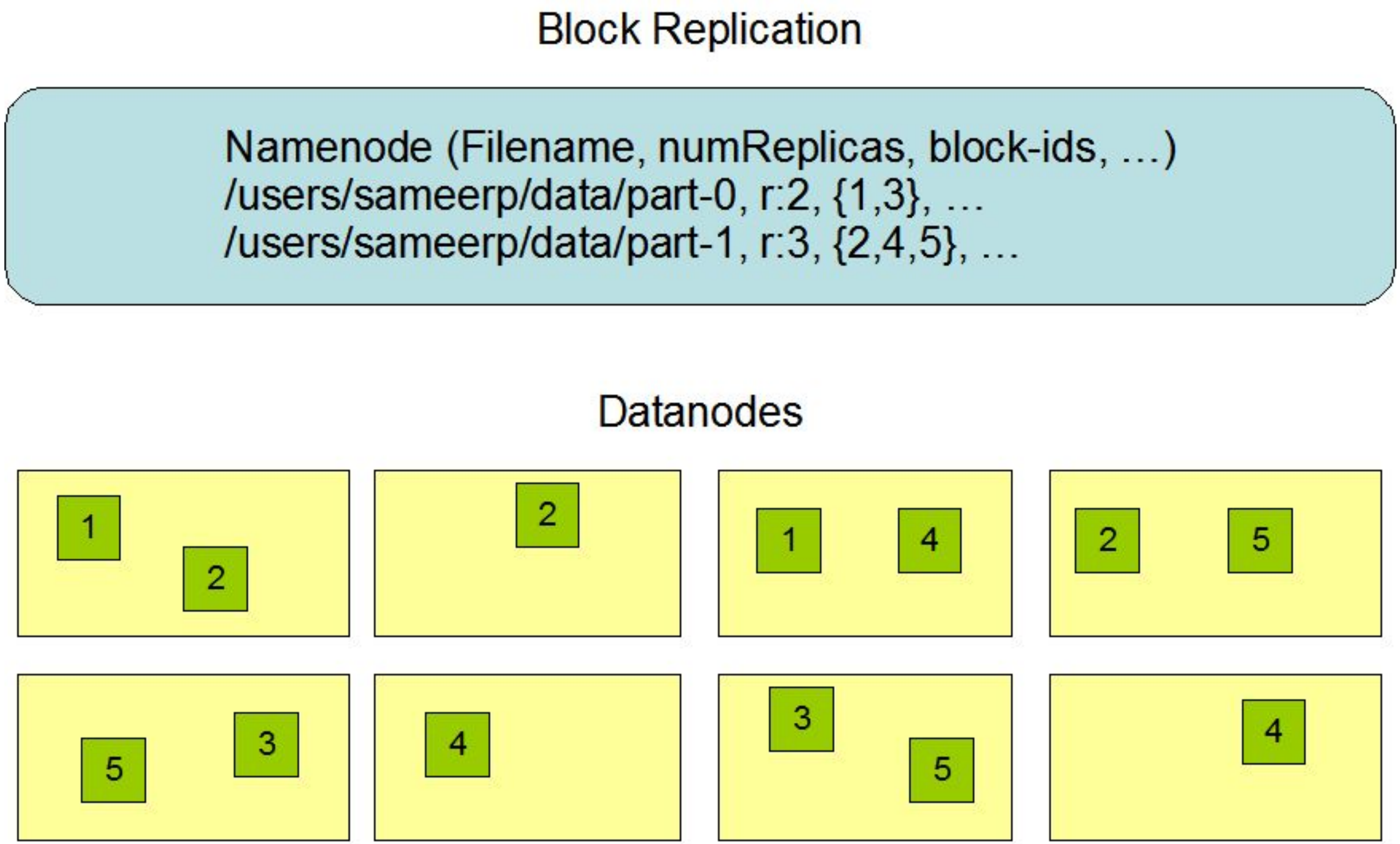
- Stores data in files on its local file system
  - Each HDFS block in a separate file
  - Has no knowledge about HDFS file system

- When the DataNode starts up:
  - It generates a list of all HDFS blocks = BlockReport
  - It sends the report to NameNode

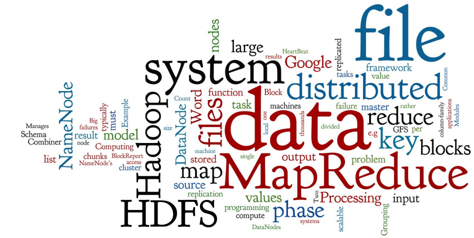
HDFS: Blocks & Replication

- HDFS can store very large files across a cluster
  - Each file is a sequence of blocks
  - All blocks in the file are of the same size
    - Except the last one
    - Block size is configurable per file (default 128MB)
  - Blocks are replicated for fault tolerance
    - Number of replicas is configurable per file
- NameNode receives HeartBeat and BlockReport from each DataNode
  - BlockReport: list of all blocks on a DataNode

HDFS: Block Replication



HDFS: Reliability



- Primary objective: to store data reliably in case of:
  - NameNode failure
  - DataNode failure
  - Network partition
    - a subset of DataNodes can lose connectivity with NameNode

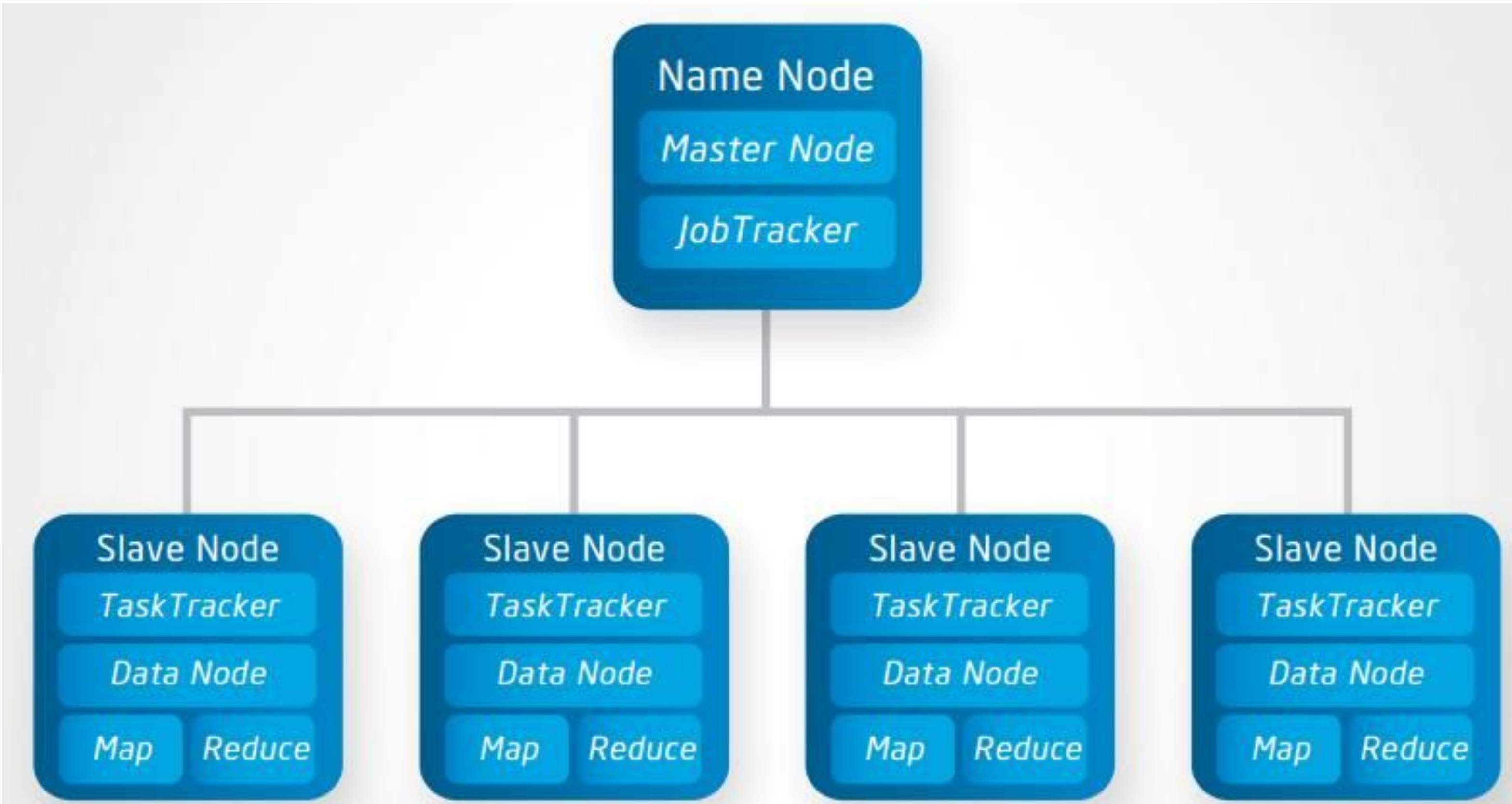
- In case of absence of a HeartBeat message
  - NameNode marks DataNodes without HeartBeat and does not send any I/O requests to them
  - The death of a DataNode typically results in re-replication

Hadoop: MapReduce

- Hadoop MapReduce **requires**:
  - Distributed file system (typically HDFS)
  - Engine that can distribute, coordinate, monitor and gather the results (typically YARN)

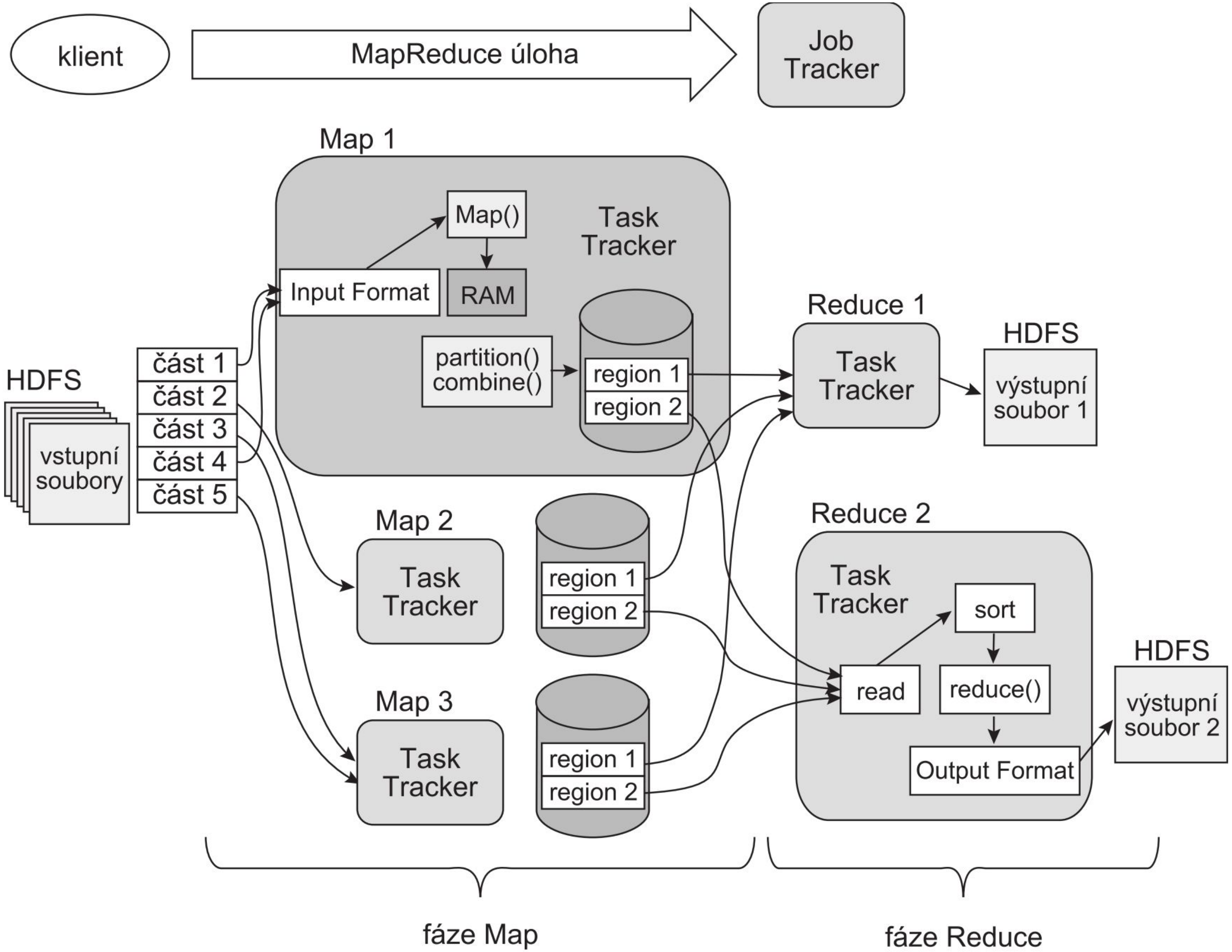
- Two main **components**:
  - JobTracker (master) = scheduler
    - tracks the whole MapReduce job
    - communicates with HDFS NameNode to run the task close to the data
  - TaskTracker (slave on each node) – is assigned a Map or a Reduce task (or other operations)
    - Each task runs in its own JVM

Hadoop HDFS + MapReduce



source: http://bigdata.black/architecture/hadoop/what-is-hadoop/

Hadoop MapReduce: Schema

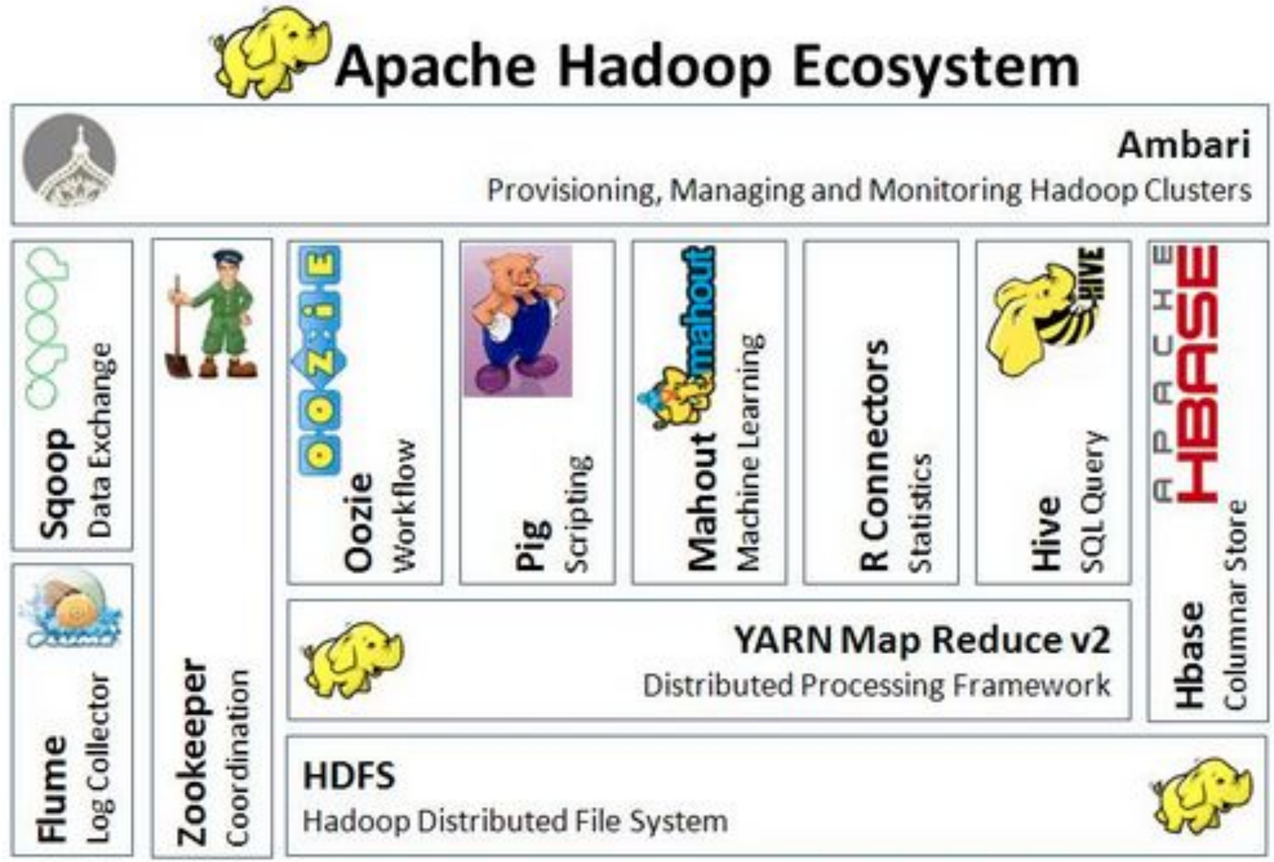


Hadoop MR: WordCount Example (1)


```
public class Map
      extends Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private final Text word = new Text();

  @Override protected void map(LongWritable key, Text value,
      Context context) throws ... {
    String string = value.toString()
    StringTokenizer tokenizer = new StringTokenizer(string);
    while (tokenizer.hasMoreTokens()) {
      word.set(tokenizer.nextToken());
      context.write(word, 1);
    }
  }
}
```

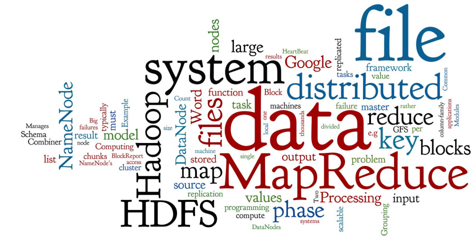
Hadoop MR: WordCount Example (2)


```
public class Reduce
      extends Reducer<Text, IntWritable, Text, IntWritable> {

   @Override
   public void reduce (Text key, Iterable<IntWritable> values,
         Context context) throws ... {
      int sum = 0;
      for (IntWritable val : values) {
         sum += val.get();
      }
      context.write(key, new IntWritable(sum));
   }
}
```



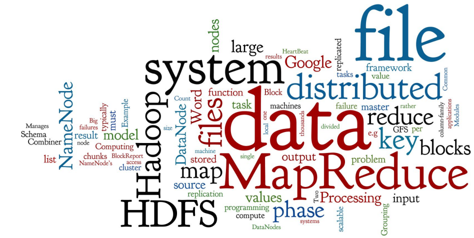
source: http://www.dineshonjava.com/2014/11/hadoop-architecture.html#.WLU6aBLyso8

Hadoop: Related Projects



- Avro: a data serialization system
- HBase: scalable distributed column-family database
- Cassandra: scalable distributed column-family database
- ZooKeeper: high-performance coordination service for distributed applications
- Hive: data warehouse: ad hoc querying & data summarization
- Pig: high-level data-flow language and execution framework for parallel computation
- Chukwa: a data collection system for managing large distributed systems
- Mahout: scalable machine learning and data mining library

Agenda



- Distributed Data Processing
- Google MapReduce
    - Motivation and History
    - Google File System (GFS)
    - MapReduce: Schema, Example, MapReduce Framework
- Apache Hadoop
    - Hadoop Modules and Related Projects
    - Hadoop Distributed File System (HDFS)
    - Hadoop MapReduce
- **MapReduce in Other Systems**

MapReduce: Implementation

hadoop MapReduce

Spark

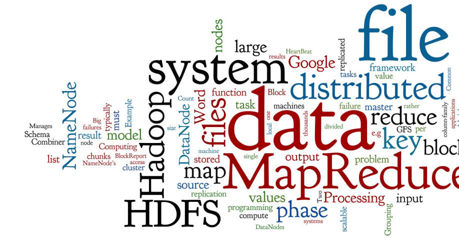
mongoDB

Amazon Elastic MapReduce

riak

Cassandra

Apache Spark



- Engine for distributed data processing
  - Runs over Hadoop Yarn, Apache Mesos, standalone, …
  - Can access data from HDFS, Cassandra, HBase, AWS S3

- Can do MapReduce
  - Is much faster than pure Hadoop
    - They say 10x on the disk, 100x in memory
  - The main reason: intermediate data in memory

- Different languages to write MapReduce tasks
  - Java, Scala, Python, R

homepage: http://spark.apache.org/

Apache Spark: Example

- Example of a MapReduce task in Spark Shell
  - The shell works with Scala language
  - Example: Word count

```
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

- Comparison of Hadoop and Spark: [link]

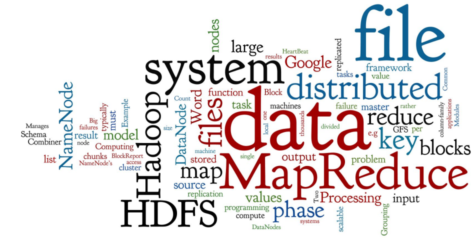
MapReduce in MongoDB


```
collection "accesses":
{
  "user_id": <ObjectId>,
  "login_time": <time_the_user_entered_the_system>,
  "logout_time": <time_the_user_left_the_system>,
  "access_type": <type_of_the_access>
}
```

- How much time did each user spend logged in
  - Counting just accesses of type “regular”

```
db.accesses.mapReduce(
  function() { emit (this.user_id, this.logout_time - this.login_time); },
  function(key, values) { return Array.sum( values ); },
  {
    query: { access_type: "regular" },
    out: "access_times"
  }
)
```

References



- RNDr. Irena Holubova, Ph.D. MMF UK course NDBI040: Big Data Management and NoSQL Databases
- Dean, J. & Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In OSDI 2004 (pp 137-149)
- Firas Abuzaid, Perth Charernwattanagul (2014). Lecture 8 “NoSQL” of Stanford course CS145. link
- J. Leskovec, A. Rajaraman, and J. D. Ullman, Mining of Massive Datasets. 2014.
- I. Holubová, J. Kosek, K. Minařík, D. Novák. Big Data a NoSQL databáze. Praha: Grada Publishing, 2015. 288 p.