LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR DATENBANKSYSTEME UND DATA MINING

Lecture Notes to

Big Data Management and Analytics

Winter Term 2018/2019

Batch Processing Systems

DBS

Outline

- **Distributed File Systems (re-visited)**
- **MapReduce**
  - Motivation
  - Programming Model
  - Example Applications
- **Hadoop MapReduce**
  - System Architecture
  - Workflow
  - YARN

NoSQL and RBDMS

- **Drawbacks of RDBMS**
  - Database system are difficult to scale.
  - Database systems are difficult to configure and maintain
  - Diversification in available systems complicates its selection
  - Peak provisioning leads to unnecessary costs

- **Advantages of NoSQL systems:**
  - Elastic scaling
  - Less administration
  - Better economics
  - Flexible data models

NoSQL and Batch Systems

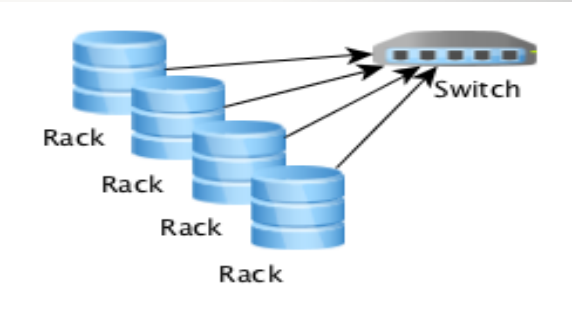
- NoSQL drops a lot of functionality of RDBMS:
  - no real data dictionaries, but semi-structured models for providing meta-data. (Still hard to access without explicit knowledge of the data model.)
  - Transaction processing cmp. CAP-theorem
  - often limited access control (no user groups, roles)
  - limited indexing / efficiency is most replaced with scalability
- So what's left:
  - storing massive amount of data in cluster environments (sharding and replication)
  - eventual consistency (at some point after the change every instance of the data is replication consistent
  - some database like APIs (e.g., CQL)

But then:
What's makes the DBMS so much different from a File-System?

Distributed File Systems

- majority of analysis is still run on files
- machine learning, statistics and data mining methods usually access all available data
- most data mining and statistics methods require a well-defined input and not semi-structured objects

  (data cleaning, transformation…)



⇒ Scalable Data Analytics often suffices with a distributed file system

⇒ Analytics methods are parallelized on top of the distributed file systems

Distributed File Systems

Past

- most computing is done on a single processor:
  - one main memory
  - one cache
  - one local disk, …

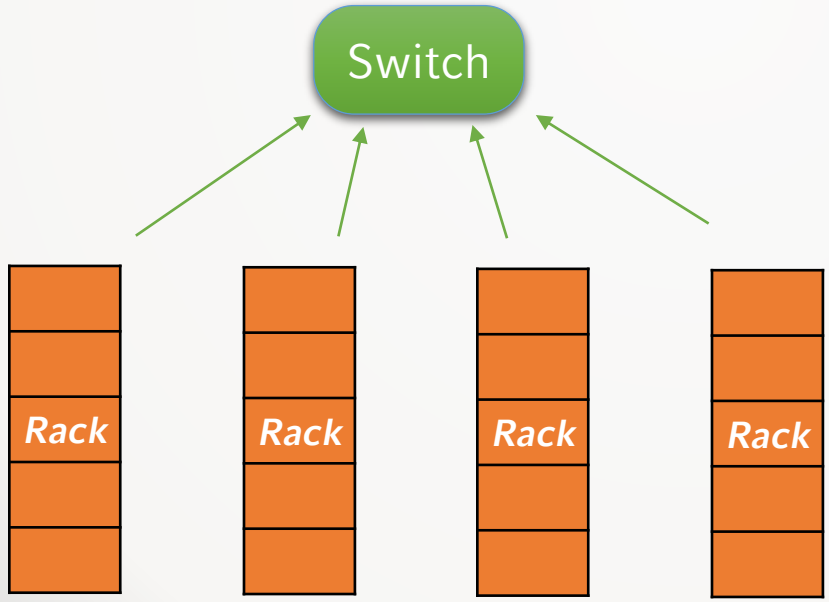
New Challenges:

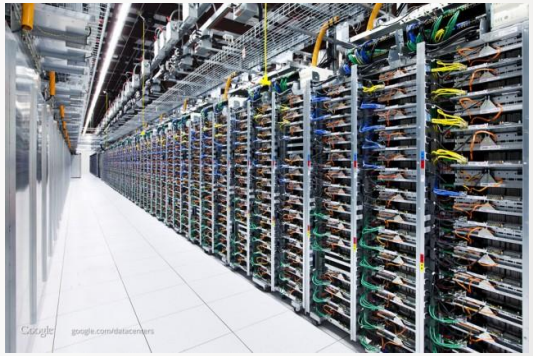
- Files must be stored redundantly:
  - If one node fails, all of its files would be unavailable until the node is replaced (see File Management)
- Computations must be divided into tasks:
  - a task can be restarted without affecting other tasks (see MapReduce)
- use of commodity hardware

Distributed File Systems

Parallel computing architecture

- Referred to as *cluster computing*
- Physical Organization:
  - compute nodes are stored on racks (8-64)
  - nodes on a single rack connected by a network





Racks of servers (and switches at the top), at Google's Mayes County, Oklahoma data center [extremetech.com]

Nodes within a rack are connected by a network, typically gigabit Ethernet

Distributed File Systems

Parallel computing architecture

Large-Scale File-System Organization:

- Characteristics:
  - files are several terabytes in size (Facebook's daily logs: 60TB; 1000 genomes project: 200TB; Google Web Index; 10+ PB)
  - files are rarely updated
  - reads and appends are common

- **Exemplary distributed file systems:**
  - Google File System (GFS)
  - Hadoop Distributed File System (**HDFS**, by Apache)
  - CloudStore
  - HDF5
  - S3 (Amazon EC2)
  - …

Distributed File Systems

Parallel computing architecture

- **Large-Scale File-System Organisation:**
  - Organisation:
    - files are divided into chunks (typically 16-64MB in size)
    - chunks are replicated n times (i.e default in HDFS: n=3) at n different nodes (optimally: replicas are located on different racks optimising fault tolerance)
  - how to find files?
    - existence of a master node
    - holds a meta-file (directory) about location of all copies of a file → all participants using the DFS know where copies are located

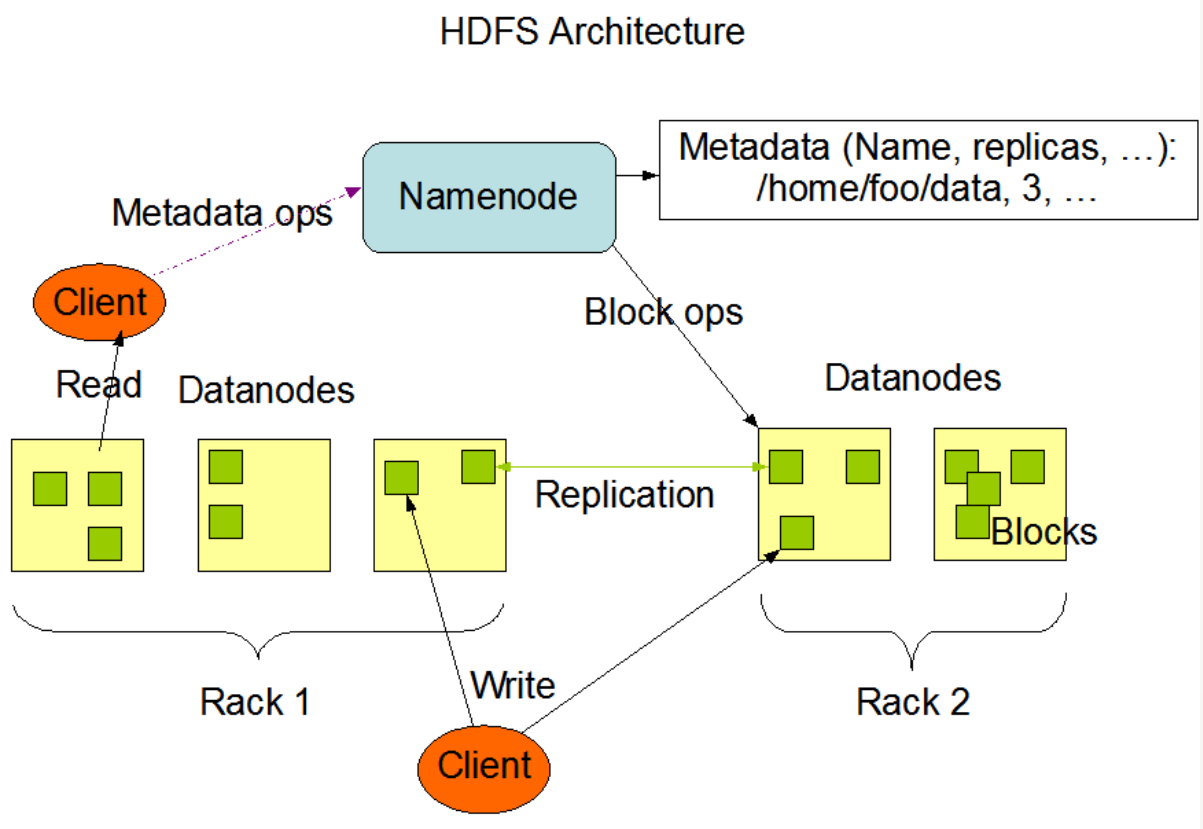
Hadoop Distributed File System (HDFS)

Apache Hadoop - Architecture of HDFS

- HDFS: A distributed file system that provides high-throughput access to application data
- HDFS has a master/slave architecture
  - **NameNode** as master
    - arbitrator and repository for all HDFS metadata
    - manages the file system namespace
    - mapping of blocks to DataNodes
    - regulates access to files by clients
  - **DataNodes** as clients:
    - manage storage attached to the nodes that they run on
    - storing “blocks” of files
    - responsible for serving read and write requests from the clients
    - perform block creation, deletion and replication upon instruction from the NameNode

HDFS-Architecture

Apache Hadoop - Hadoop Distributed File System (HDFS)



Source: http://hortonworks.com/hadoop/hdfs/#section_2

Data Storage Operations on HDFS

- **Characteristics:**
  - Write Once, Read Often Model
  - Content of individual files cannot be modified, but we can append new data at the end of a file

- **Operations:**
  - create a new file
  - delete a file
  - rename a file
  - append content to the end of a file
  - modify file attributes (owner, read, write)

Partitioning the input data

HDFS Blocks

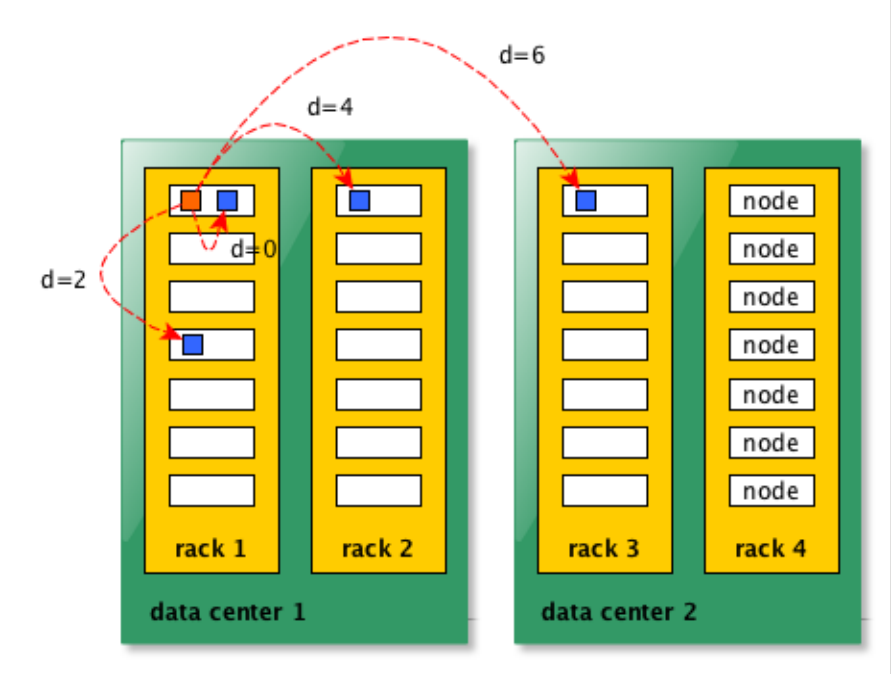
- File is divided into blocks (default:64 MB) and duplicated in multiple nodes (default: 3 replicas)
  => Fault tolerance
- Dividing files into blocks is common for a FS, e.g. default block size in Linux is 4KB.
  => Difference of HDFS is the **scale**
- Hadoop was designed to operate at the petabyte scale
- Every data block stored in HDFS has its own metadata and needs to be tracked by a central server
- files in HDFS are write-once and have strictly one writer at any time

Partitioning the input data

HDFS Blocks

Network Topology and Distance

- process on the same node
  distance(/d1/r1/n1, /d1/r1/n1) = 0
- different nodes / same rack
  distance(/d1/r1/n1, /d1/r1/n2) = 2
- different racks/same datacenter
  distance(/d1/r1/n1, /d1/r2/n3) = 4
- different datacenters
  distance(/d1/r1/n1, /d2/r3/n4) = 6

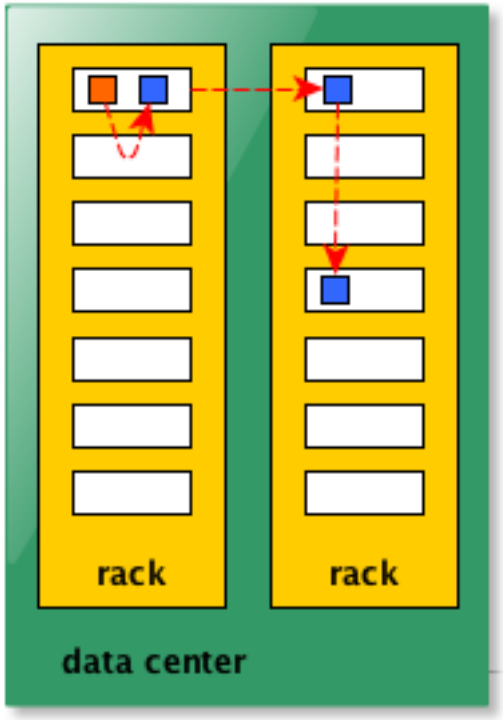


Partitioning the input data

HDFS Blocks

Hadoop's default **replica placement** strategy:

- **1st** replica:
  - same node as the client
  - if node outside cluster, choose a random node
- **2nd** replica:
  - *off-rack*, chosen at random
- **3rd** replica:
  - same rack as 2nd, but on a different node, chosen at random
- further replicas:
  - random nodes; avoid placing too many replica on same rack



HDFS Robustness

- **Data Disk Failure**, **Heartbeats** and **Re-replication**
- **Cluster Rebalancing**
  - move data from one DataNode to another if free space falls below a certain threshold
- **Data Integrity**
  - checksum checking of each block of a file
- **Metadata Disk Failure**
  - replica of NameNode + copies of log files
- **Snapshots**
  - support of storing a copy of data at a particular instant of time

Data Disk Failure, Heartbeats and Re-replication

- Each DataNode sends a Heartbeat message to the NameNode periodically.
- (subsets of) DataNodes may lose connectivity with the NameNode
  → detected by absence of Heartbeats
- Affected DataNodes marked as dead
  → no new IO requests from NameNode.
- Any data that was registered to a dead DataNode is not available to HDFS any more.
- DataNode death may cause the replication factor of some blocks to fall below their specified value
  → re-replication (initiated by NameNode) needed.
- re-replication also required if the replication factor of a file has been increased.

Cluster Rebalancing

- The HDFS architecture is compatible with data rebalancing schemes.
- A scheme might automatically move data from one DataNode to another if the free space on a DataNode falls below a certain threshold.
- In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster.
- These types of data rebalancing schemes are not yet implemented.

Data Integrity

- It is possible that a block of data fetched from a DataNode arrives corrupted (e.g. because of faults in a storage device, network faults, or buggy software.)
- The HDFS client software implements **checksum checking** on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace.
- When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block.

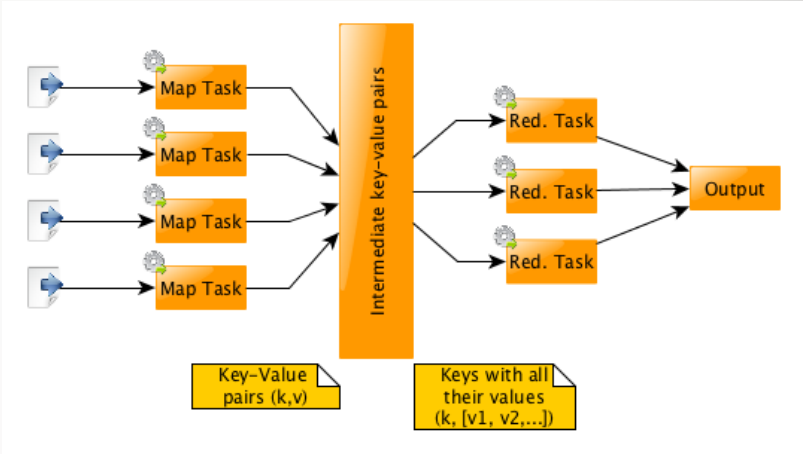
Metadata Disk Failure

- FsImage and EditLog are central data structures of HDFS.
- Corruption of these files can cause the HDFS instance to be non-functional.
  → maintaining multiple copies of the FsImage and EditLog.
- Updates to each FsImage or EditLog instance only synchronously → degrades the rate of namespace transactions per second (but HDFS applications are not metadata intensive)
- When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.
- The NameNode machine is a single point of failure for an HDFS cluster. If the NameNode machine fails, manual intervention is necessary. Currently, automatic restart and failover of the NameNode software to another machine is not supported.

Snapshots

- Snapshots support storing a copy of data at a particular instant of time.
- One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time.
- HDFS does not currently support snapshots but will in a future release.

Overview



MapReduce

- Motivation
- Programming Model
  - Recap Functional Programming
- Examples

MapReduce

Motivation: MapReduce - Comparison to Other Systems

MapReduce vs. RDBMS

	MapReduce	*RDBMS*
Data size	***Petabytes***	***Gigabytes***
Access	***Batch***	***Interactive and Batch***
Updates	***Write once, read many times***	***Read & Write many times***
Structure	***Dynamic schema***	***Static schema***
Integrity	***Low***	***High (normalized data)***
Scaling	***Linear***	***Non-linear***

MapReduce

Motivation: MapReduce - Comparison to Other Systems

MapReduce vs. Grid Computing

- Accessing large data volumes becomes a problem in High performance computing (HPC), as the **network bandwidth** is the bottleneck <-> Data Locality in MapReduce
- in HPC, programmers have to explicitly handle the **data flow** <-> MapReduce operates only in higher level, i.e. data flow is implicit
- handling **partial failures** <-> MapReduce as a *shared-nothing-architecture (no dependence of tasks);* detects failures and reschedules missing operations

MapReduce

Motivation: Large Scale Data Processing

In General:

- MapReduce can be used to manage large-scale computations in a way that is tolerant of hardware faults

- System itself manages automatic parallelisation and distribution, I/O scheduling, coordination of tasks that are implemented in ***map()*** and ***reduce()*** and copes with unexpected system failures or stragglers

- several implementations: Google's internal implementation, open-source implementation Hadoop (using HDFS), …

MapReduce

Programming Model - General Processing

- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:

map (in_key, in_value) -> list (out_key, intermediate_value)

- Processes input key/value pair; one Map()-Call for every pair
- Produces a set of intermediate pairs

reduce (out_key, list(intermediate_value)) -> list (out_value)

- combines all intermediate values for a particular key; one Reduce()-call per unique key
- produces a set of merged output values (usually just one output value)

MapReduce

Programming Model – Recap: Functional Programming

- MapReduce is inspired by similar primitives in LISP, SML, Haskell and other languages

- The general idea of higher order functions (map and fold) in functional programming (FP) languages are transferred in the environment of MapReduce:
  - **map** in MapReduce <-> **map** in FP
  - **reduce** in MapReduce <-> **fold** in FP

MapReduce

Programming Model – Recap: Functional Programming

- MAP:
  - 2 parameters: applies a function on each element of a list
  - the type of the elements within the result list can differ from the type of the input list
  - the size of the result list remains the same

In Haskell:

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

Example:

***Main>** map (\x -> (x,1)) [“Big”,”Data”,”Management”,”and”,”Analysis”]
[(“Big”,1),(“Data”,1),(”Management”,1),(“and”,1),(“Analysis”,1)]

MapReduce

Programming Model – Recap: Functional Programming

- FOLD:
  - 3 parameters: traverse a *list* and apply a function *f()* to each element plus an *accumulator*. *f()* returns the next *accumulator* value
  - in functional programming: foldl and foldr

In Haskell (analog foldr):

```
foldl :: (b->a->b) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = fold f (f acc x) xs
```

Example:

```
*Main> foldl (\acc (key,value) -> acc + value) 0 [("Big", 1), ("Big", 1), ("Big",1)]
3
```

MapReduce

Programming Model - General Processing of MapReduce

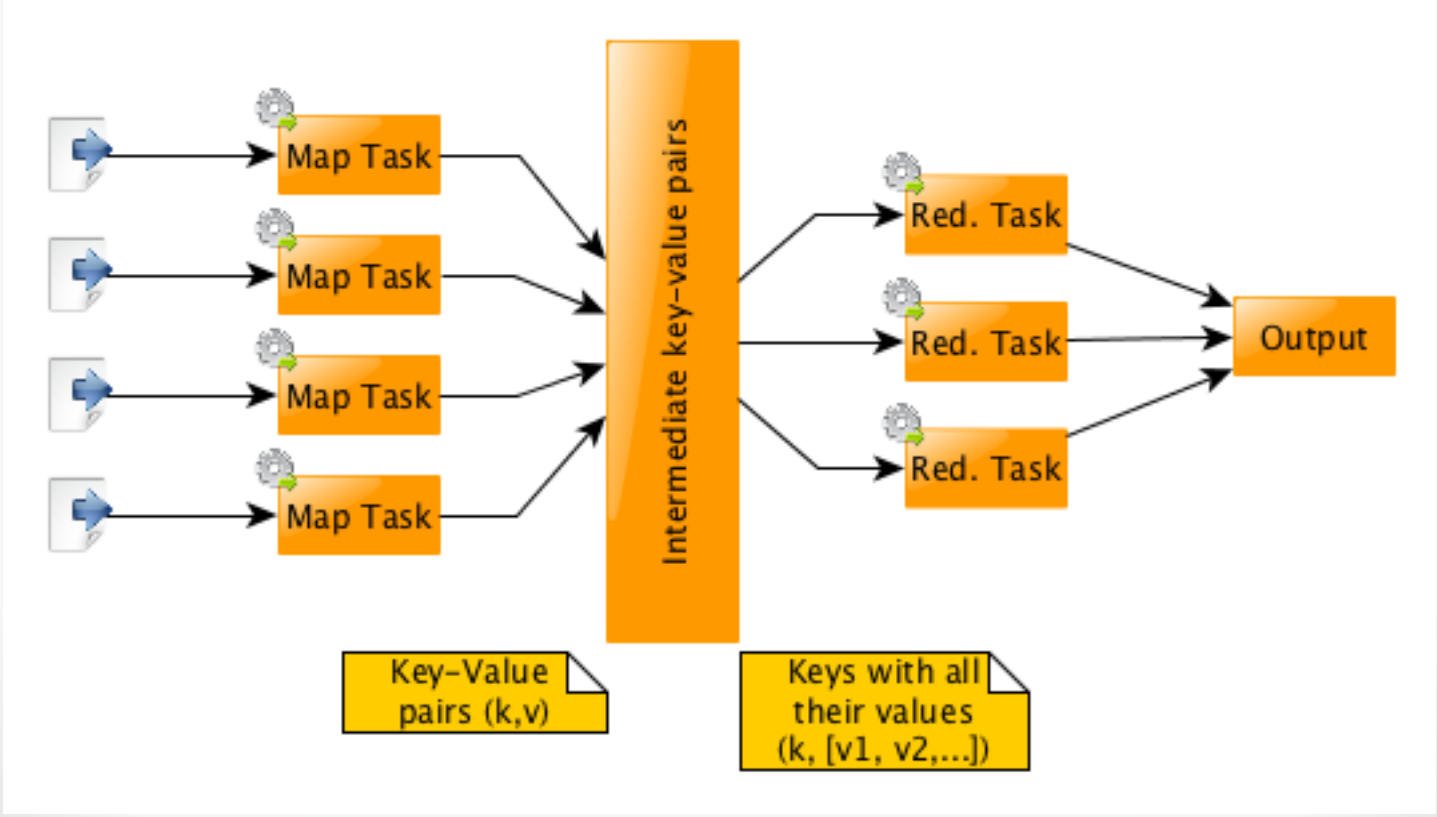
- 1. Chunks from a DFS are attached to Map tasks turning each chunk into a sequence of *key-value* pairs.

- 2. key-value pairs are collected by a master controller and sorted by key. The keys are divided among all Reduce tasks.

- 3. Reduce tasks work on each key separately and combine all the values associated with a specific key.

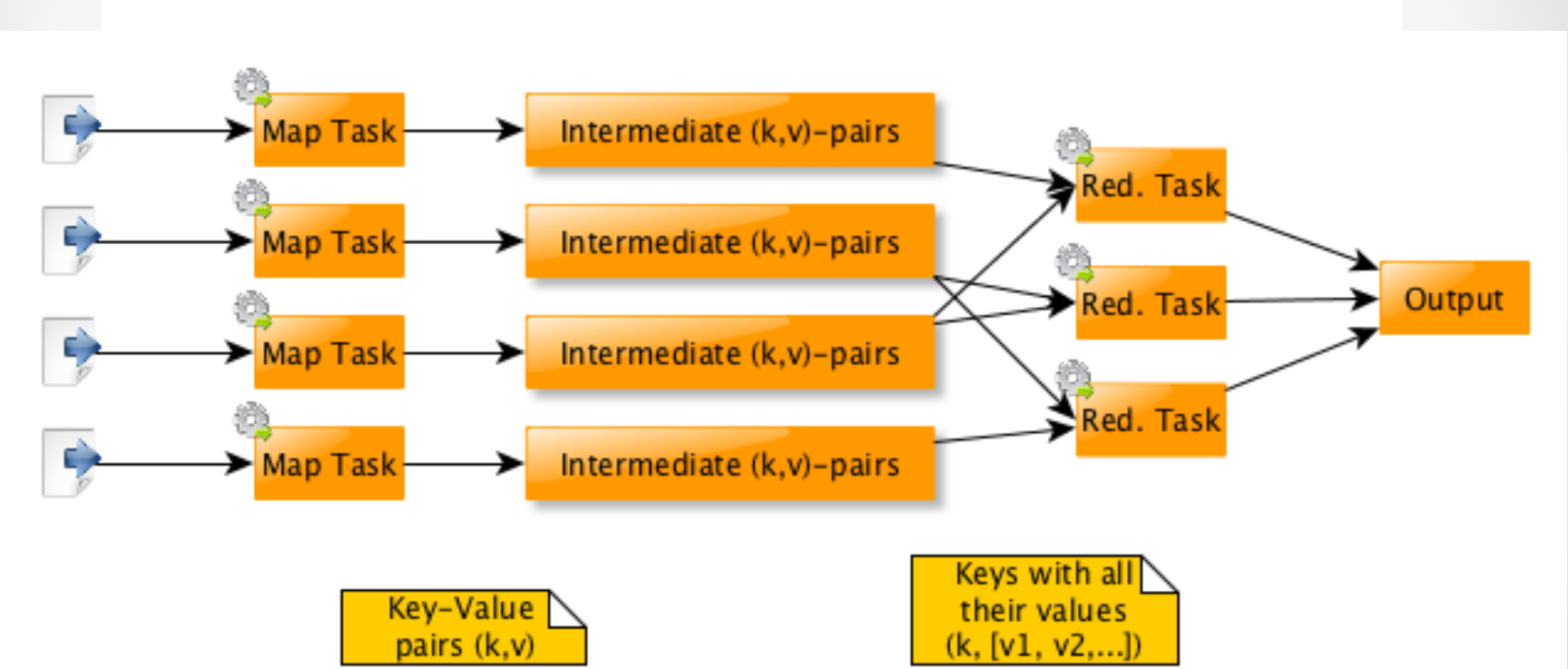
MapReduce

Programming Model - High-level MapReduce diagram



MapReduce

Programming Model - High-level MapReduce diagram



MapReduce

Programming Model - General Processing

- Programmer's task: specify map() and reduce();
- MapReduce environment takes care of:
  - **Partitioning** the input data
  - **Scheduling**
  - **Shuffle and Sort** (performing the group-by-key step)
  - Handling machine **failures** and **stragglers**
  - Managing of required inter-machine **communication**

MapReduce

Programming Model - General Processing

Partitioning the input data

- data files are divided into blocks (default in GFS/HDFS: 64 MB) and replicas of each are stored on different nodes
- Master schedules map() tasks in close proximity to data storage
    - map() tasks are executed physically on the same machine where one replica of an input file is stored (or, at least on the same rack → communication via network switch)
    - → Goal: conserve network bandwidth (c.f Grid Computing)

→ achieves to read input data at local disk speed, rather than limiting read rate by rack switches

MapReduce

Programming Model - General Processing

Scheduling

- **One master, many workers**
  - split input data into M map tasks
  - reduce phase partitioned into R tasks
  - tasks are assigned to workers dynamically
- **Master assigns each map task to a free worker**
  - considers proximity of data to worker
    - → worker reads task input (optimal: from local disk)
    - → output: files containing intermediate (key,value)-pairs sorted by key
- **Master assigns each reduce task to a free worker**
  - worker reads intermediate (key, value)-pairs
  - worker merges and applies reduce()-function for output

MapReduce

Programming Model - General Processing

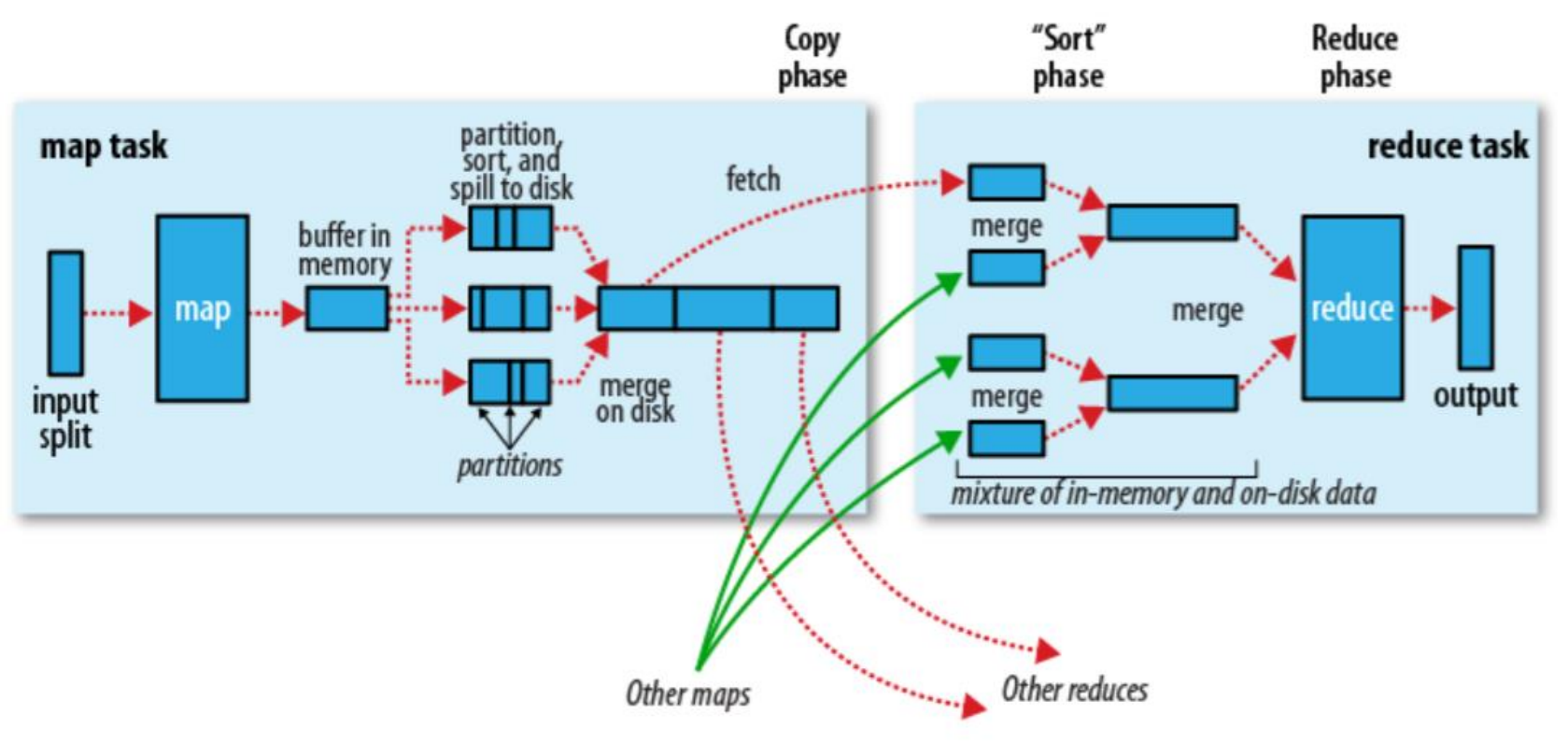
Shuffle and Sort (performing the group-by-key step)

- input to every reducer is sorted by key
- Shuffle: sort and transfer the map outputs to the reducers as inputs
- Mappers need to separate output intended for different reducers
- Reducers need to collect their data from all(!) mappers
  - keys at each reducer are processed in order

MapReduce

Programming Model - General Processing

Shuffle and Sort (performing the group-by-key step)



Quelle: Oreilly, Hadoop - The Definitive Guide 3rd Edition, May 2012

MapReduce

Programming Model - General Processing

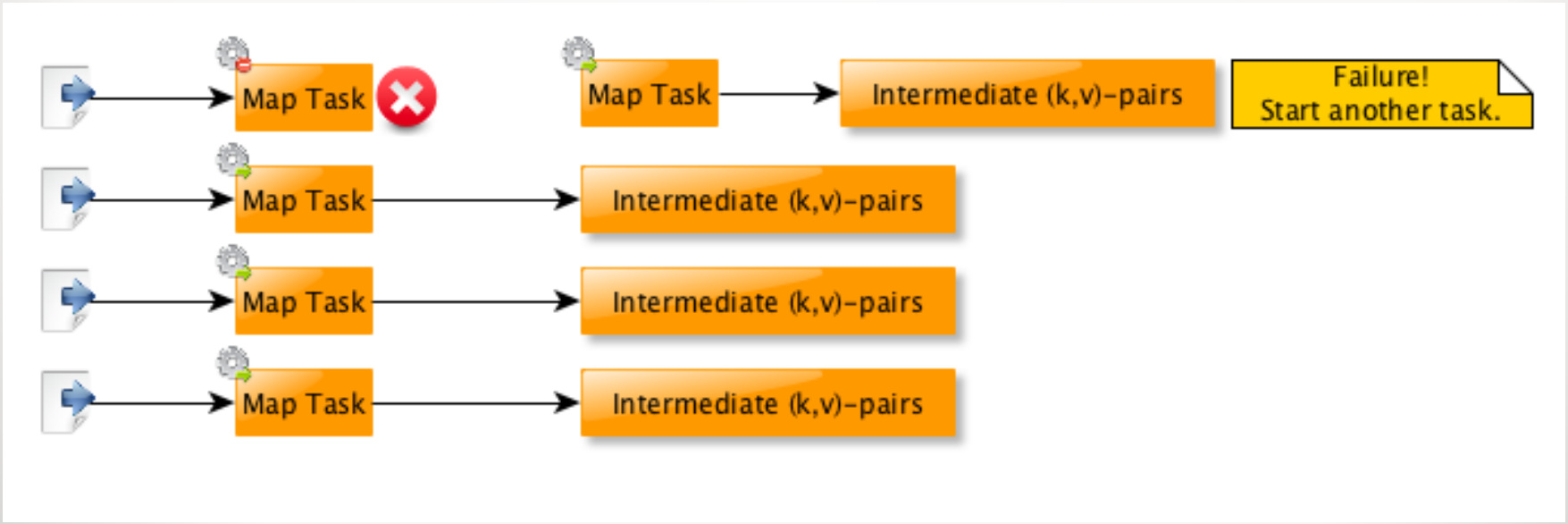
Handling machine **failures** and **stragglers**

- General: master pings workers periodically to detect failures
  - **Map worker failure**
    - Map tasks completed or in-progress at worker are reset to idle
    - all reduce workers will be notified about any re-execution
  - **Reduce worker failure**
    - only in-progress tasks at worker will be re-executed
    - → output stored in global FS
  - **Master failure**
    - master node is replicated itself. 'Backup' master recovers last updated log files (metafile) and continues
    - if no 'backup' master → MR task is aborted and client is notified

MapReduce

Programming Model - General Processing

Handling machine **failures** and **stragglers**
- Failures



MapReduce

Programming Model - General Processing

Handling machine **failures** and **stragglers**

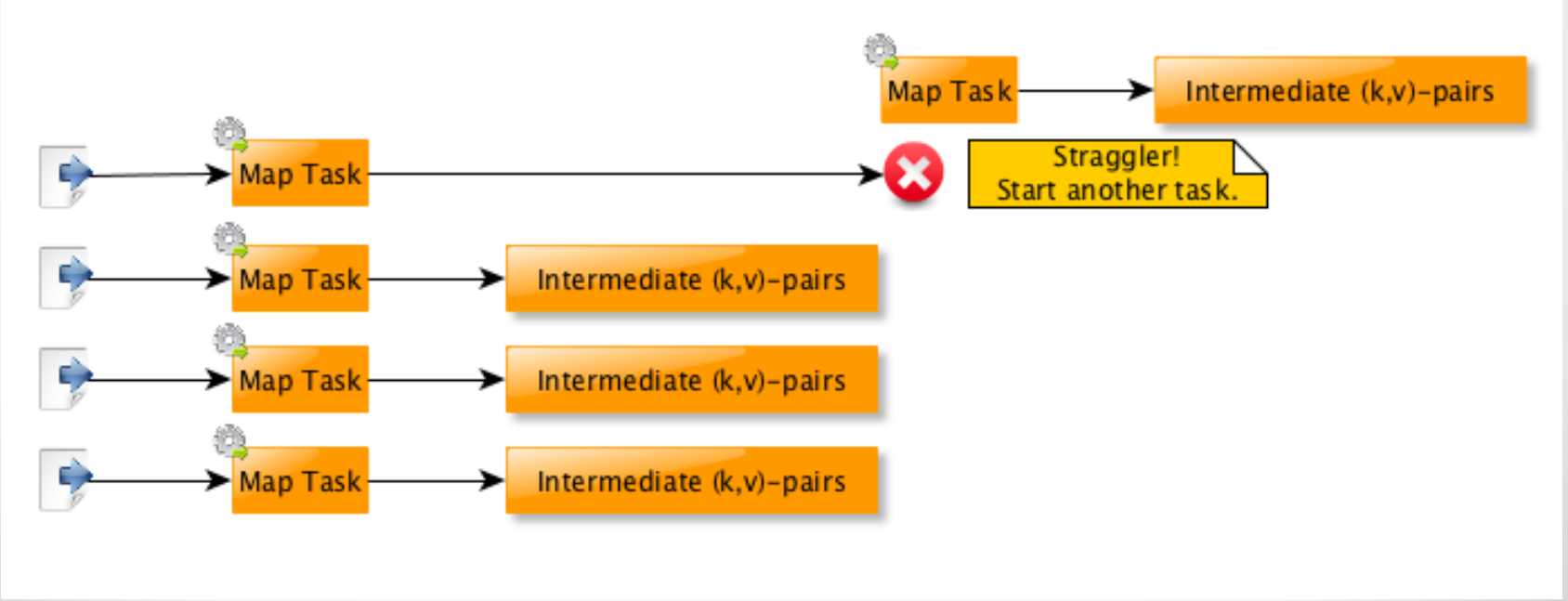
- **Stragglers**
  - slow workers lengthen the termination of a task
  - close to completion, backup copies of the remaining in-progress tasks are created
  - Causes: hardware degradation, software misconfiguration, …
  - if a task is running slower than expected, another equivalent task will be launched as backup → *speculative execution of tasks*
  - when a task completes successfully, any duplicate task are killed

MapReduce

Programming Model - General Processing

Handling machine **failures** and **stragglers**

- **Stragglers**



MapReduce

Programming Model - General Processing

Managing required inter-machine **communication**

- Task status (idle, in-progress, completed)
- Idle tasks get scheduled as workers become available
- In completion of a map task, the worker sends the location and sizes of its intermediate files to the master
- Master pushes this info to reducers
- Fault tolerance: master pings workers periodically to detect failures

MapReduce

Programming Model - General Processing - Workflow

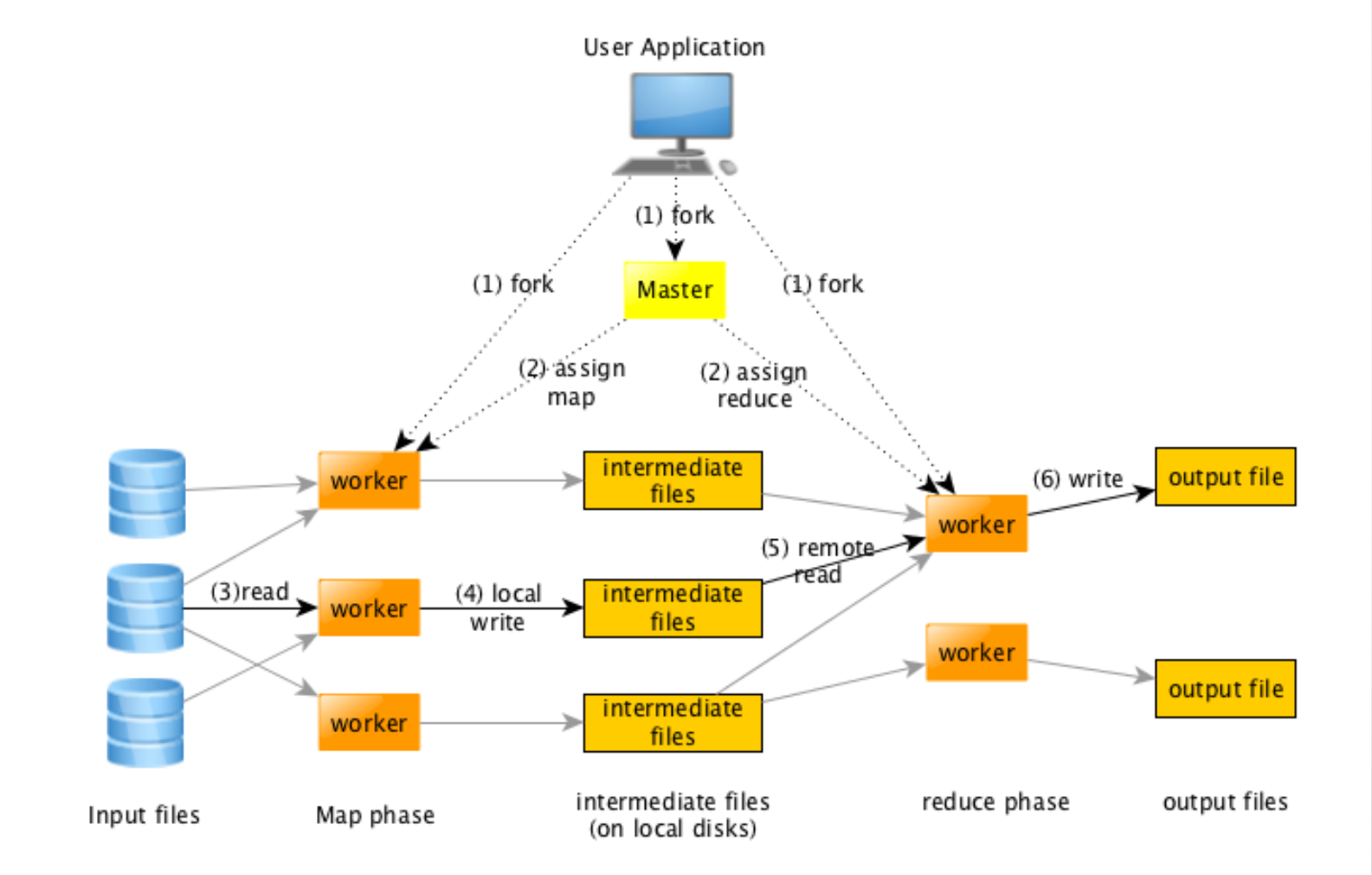
Workflow of MapReduce (as original implemented by Google):

1. Initiate the MapReduce environment on a cluster of machines
2. One Master, the rest are workers that are assigned tasks by the master
3. A map task reads the contents of an input split and passes them to the MAP-function. The results are buffered in memory
4. The buffered (key,value)-pairs are written to local disk. The location of these (intermediate) files are passed back to the master
5. A reduce worker who has been notified by the master, uses remote procedure calls to read the buffered data.
6. Reduce worker iterates over the sorted intermediate (key,value)-pairs and passes them to the REDUCE-function

$\Rightarrow$ On completion of all tasks, the master notifies the user program.

MapReduce

Programming Model - Low-level MapReduce diagram



MapReduce

Example #1 WordCount

- *Setting:* text documents, e.g. web server logs
- *Task:* count occurrence of distinct words appearing in the input file, e.g find popular URLs in server logs

Challenges:

- File is too large for to fit in a single machines's memory
- parallel execution

⇒ Solution: Apply MapReduce

MapReduce

Example #1 WordCount

- *Goal*: Count word occurrence in a set of documents
- *Input*: "Wer A sagt, muss auch B sagen! Wer B sagt, braucht B nicht nochmal sagen!"

map (k1, v1) -> list (k2, v2)

```
map (String key, String value):
    //key:document name
    //value: content of document
    for each word w in value do:
        emitIntermediate(w, "1")
```

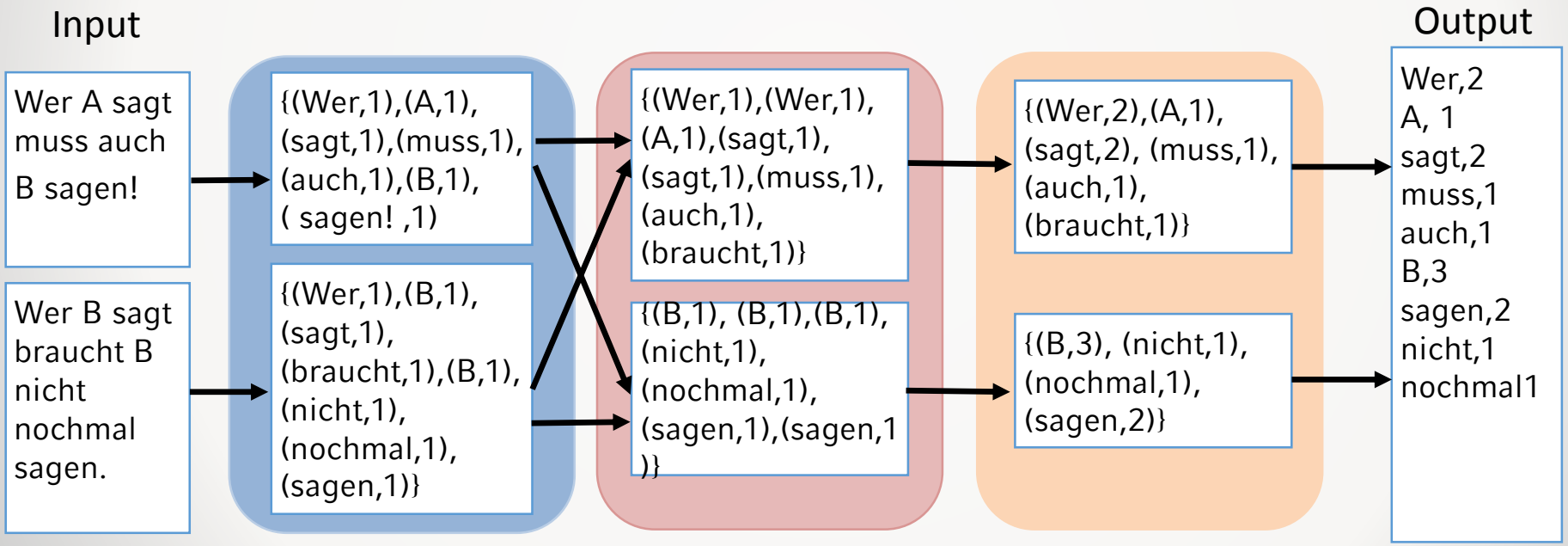
reduce (k2, list(v2)) -> list(v2)

```
reduce (String key, Iterator values):
    //key:a word
    //values: a list of counts
    int result = 0;
    for each v in values do:
        result += parseInt(v);
    emit(result.toString())
```

MapReduce

Example #1 WordCount

- In a *parallel environment*:
  - worker: 2



MapReduce

Example #2 k-Means

Randomly initialize k centers:

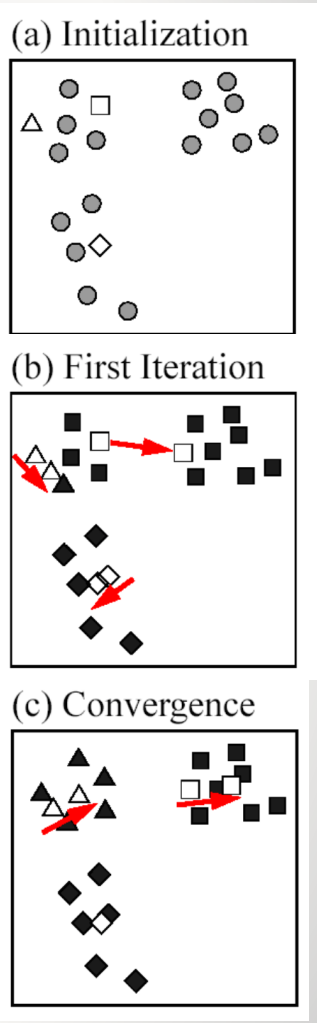
$$\mu^{(0)} = \mu_1^{(0)}, \ldots, \mu_k^{(0)}$$

Classify: Assign each point j∈{1,..,m} to nearest center:

$$z^j \leftarrow \arg\min_i ||\mu_i - x^j||_2^2$$

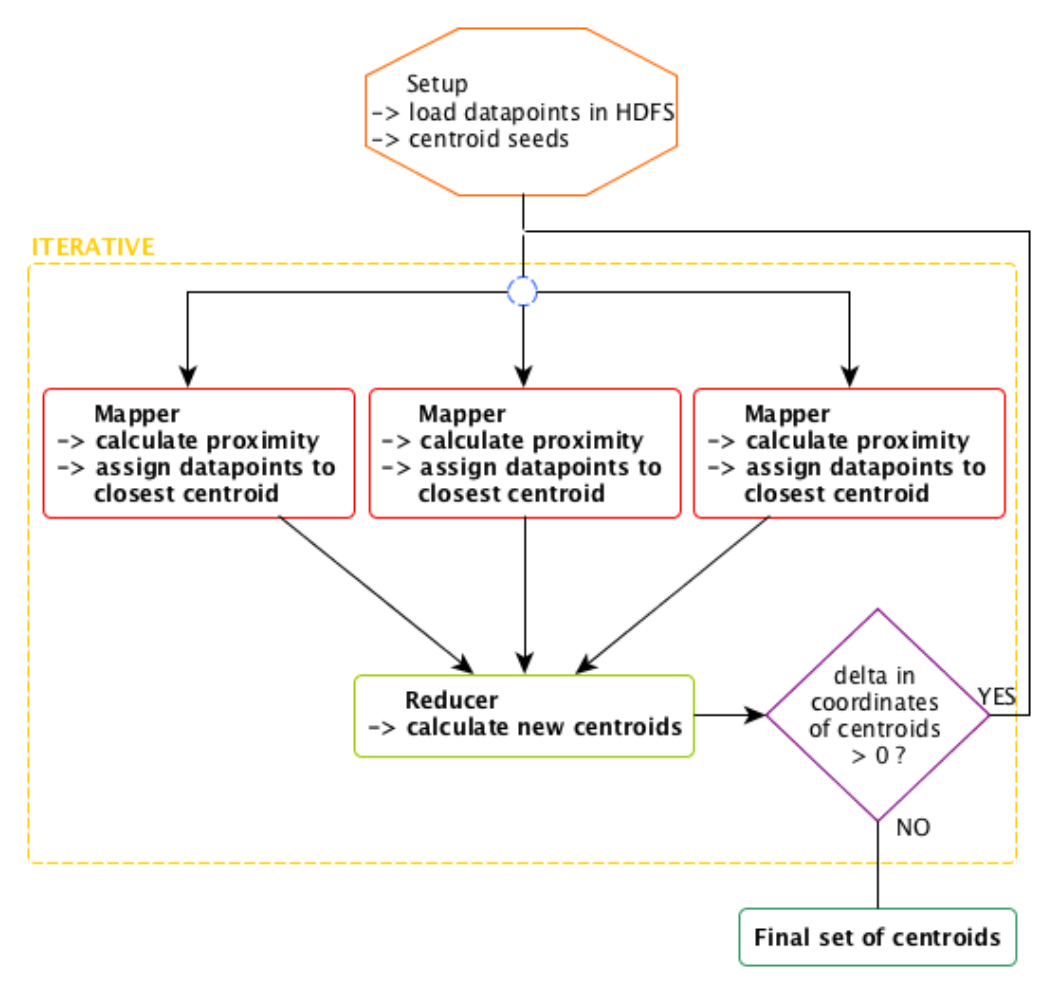
Recenter: μ_i becomes center of assigned points:

$$\mu_i^{(t+1)} \leftarrow \arg\min_\mu \sum_{j:z^j=i} ||\mu - x^j||_2^2$$



MapReduce

Example #2 k-Means - MapReduce - Scheme



MapReduce

Example #2 k-Means - Classification Step As Map

Classify: Assign each point j∈{1,..,m} to nearest center:

$$z^j \leftarrow \arg\min_i ||\mu_i - x^j||_2^2$$

Map:

Input:

- subset of d-dimensional objects of M={x_1,..x_m} in each mapper
- initial set of centroids $\mu^{(0)} = \mu_1^{(0)}, \ldots, \mu_k^{(0)}$

Output:

- list of objects assigned to nearest centroid. This list will later be read by the reducer program

MapReduce

Example #2 k-Means - Classification Step As Map

Classify: Assign each point $j \in \{1,..,m\}$ to nearest center:

```
for all x_i in M do
        bestCentroid <- null
        minDist <- inf
        for all c in C do
                dist <- l2Dist(x, c)
                if bestCentroid == null || dist < mindist then
                        minDist <- dist
                        bestCentroid <- c
                endif
        endfor
        outputlist << (bestCentroid, x_i)
endfor
return outputlist
```

MapReduce

Example #2 k-Means - Recenter Step as Reduce

Recenter: μ_i becomes centroid of assigned points

$$\mu_i^{(t+1)} \leftarrow \arg\min_{\mu} \sum_{j:z^j=i} ||\mu - x^j||_2^2$$

Note: equivalent to averaging the points!

$$\mu_i^{(t+1)} \leftarrow \sum_{j:z^j=i} x^j / \sum_{j:z^j=i} 1$$

Reduce:

Input:

- list of (key,value)-pairs, where key = bestCentroid and value = objects assigned to this centroid

Output:

- (key,value), where key = oldcentroid and value = newBestCentroid, which is the new centroid calculated for that bestCentroid

MapReduce

Example #2 k-Means - Recenter Step as Reduce

Recenter: μ_i becomes centroid of its points:

```
assignmentList <- outputlists  // lists from mappers are merged together (shuffle)

    for all (key,values) in assignmentList do
        newCentroid, sumOfObjects, numOfObjects <- null
        for all obj in values do
            sumOfObjects += obj
            numOfObjects ++
        endfor
        newCentroid <- (sumOfObjects / numOfObjects)
        newCentroidList << (key, newCentroid)
    endfor
    return newCentroidList
```

Big Data in Hadoop

Apache Hadoop - Historical Background

- 2003: Google publishes its cluster architecture & DFS (GFS)
- 2004: MapReduce as a new programming model is introduced by Google, working on top of GFS
  - written in C++ as a closed-source project
- 2006: Apache & Yahoo! publish Hadoop & HDFS
  - Java implementation of Google MapReduce and GFS
- 2008: Hadoop becomes an independent Apache project

- **Today**: Hadoop widely used as a general-purpose storage and analysis platform for big data

Big Data in Hadoop

Apache Hadoop - Google: The Data Challenge

Keynote Speech, Jeffrey Dean (Google), 2006, PACT'06:
Problems:
- 20+ billion web pages x 20KB = 400+ terabytes
- One computer can read 30-35 MB/sec from disk
  - → four months to read the web
- takes even longer to 'do' something with the data
- But: same problem with 1000 machines, < 3 hours

MapReduce CACM'08 article:
- 100,000 MapReduce jobs executed in Google every day
- Total data processed > 20 PB per day

Big Data in Hadoop

Apache Hadoop

- open-source SW for reliable, scalable, distributed computing
- Some provided modules:
  - **Hadoop Common**:
    - common utilities for distributed filesystems and general I/O (serialization, Java RPC, persistent data structures)
  - **Hadoop Distributed File System** (HDFS):
    - A distributed file system that provides high-throughput access to application data
  - **Hadoop YARN**:
    - framework for job scheduling and cluster resource management
  - **Hadoop MapReduce**:
    - distributed data-processing model and execution environment running on large clusters of commodity machines

Big Data in Hadoop

Apache Hadoop

- Tools within Hadoop

Tool	*Description*
HBase	***Distributed, column-oriented database***
Hive	***Distributed data warehouse***
Pig	***Higher-level data flow language and parallel execution framework***
ZooKeeper	***Distributed coordination service***
Sqoop	***Tool for bulk data transfer between structured data stores and HDFS***
Oozie	***Complex job workflow service***
Chukwa	***System for collecting management data***
Mahout	***Machine learning and data mining library***
BigTop	***Packaging and testing***
Avro	***serialization system for cross-language RPC and persistent data storage***

Big Data in Hadoop

Apache Hadoop - Common Use Cases

- Log Data Analysis
  - most common: write once & read often
- Fraud Detection
- Risk Modeling
- Social Sentiment Analysis
- Image Classification
- Graph Analysis
- …

Big Data in Hadoop

Apache Hadoop - HDFS/MapReduce layer composition

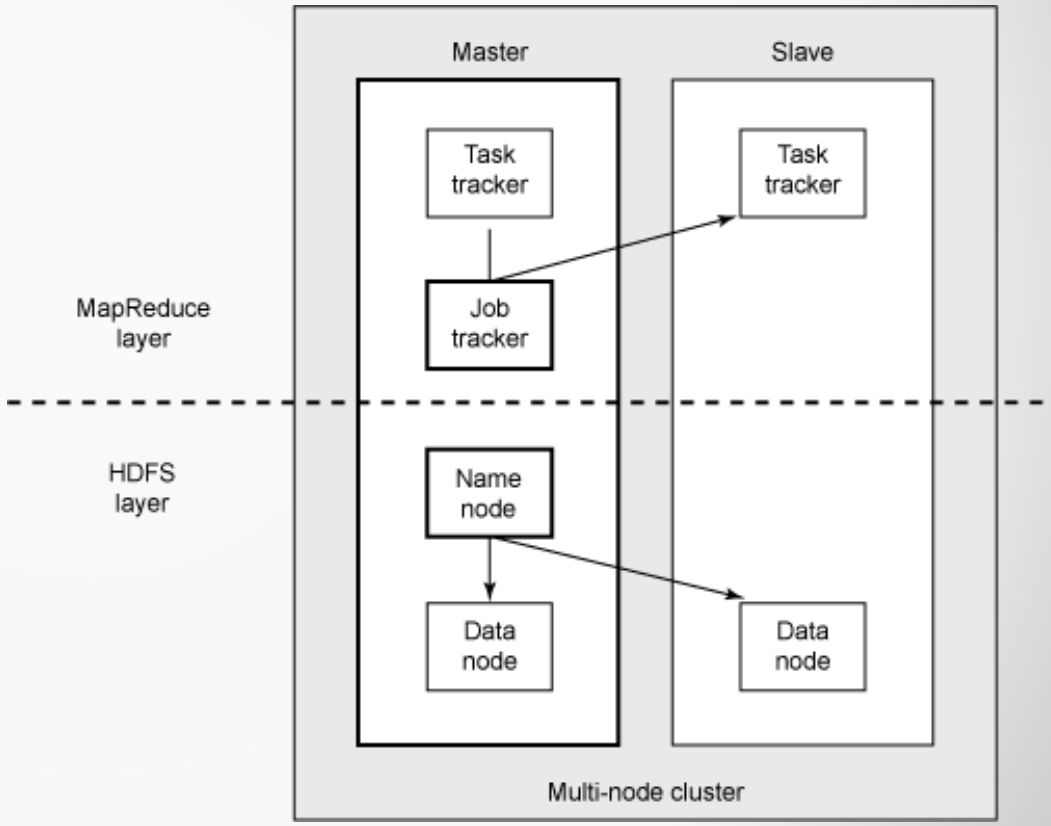
Bringing it all together…

MapReduce tasks are divided into *trackers*:

- Master: JobTracker
- Slave: TaskTracker

HDFS Tasks divided into *nodes*:

- Master: NameNode
- Slaves: DataNodes



Big Data in Hadoop

Apache Hadoop - HDFS/MapReduce layer composition

Bringing it all together…

JobTracker:

- coordinates all the jobs running on a system
- scheduling tasks to run on TaskTrackers
- keeps record of the overall progress of each job
- on Task failure: reschedule tasks on different TaskTracker

TaskTracker:

- execute tasks
- send progress reports to JobTracker

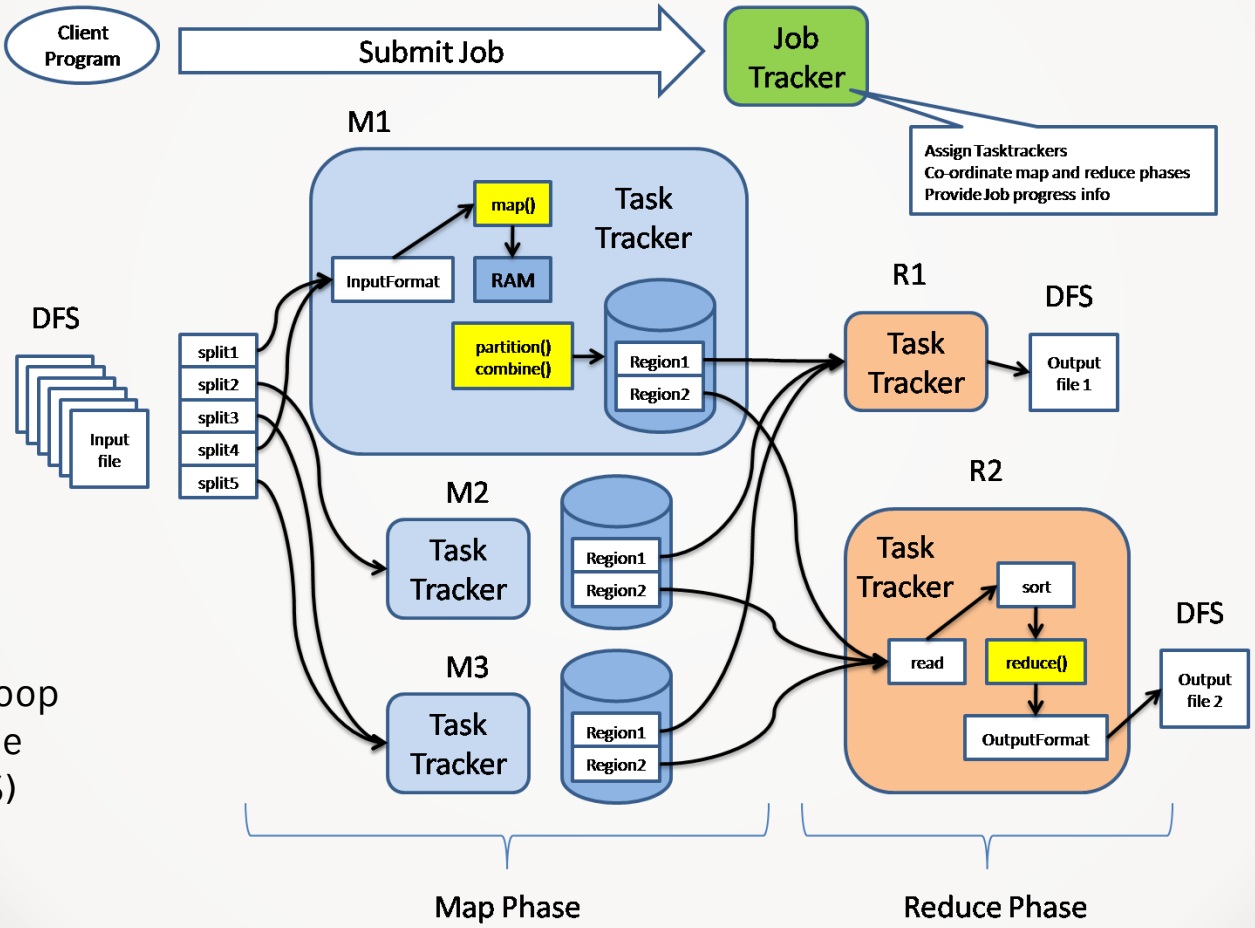
Big Data in Hadoop

Apache Hadoop - Workflow of MapReduce in Hadoop

1. Client submits an application request to the JobTracker
2. JobTracker determines processing resources to execute the entire application, e.g. selection of TaskTrackers based on their proximity to the data source
3. JobTracker identifies state of the slave nodes and queues all map tasks and reduce tasks for execution
4. When processing slots become available on slave nodes, map tasks are deployed
5. JobTracker monitors task progress. On failure, the task is restarted on next available slot.
6. When map tasks have finished, reduce tasks process the interim results sets
7. The result set is returned to the client application

Big Data in Hadoop

Apache Hadoop - MapReduce in Hadoop



DFS: e.g. Hadoop Distributed File System (HDFS)

Source: https://www.ibm.com/developerworks/cloud/library/cl-openstack-deployhadoop/

Big Data in Hadoop

Apache Hadoop - Limitations of MapReduce

- MapReduce is a successful batch-oriented programming model

- Increasing demand for additional processing modes:
  - Graph Analysis
  - Stream data processing
  - Text Analysis
  - →Demand is growing for real-time and ad-hoc analysis
  - → Analysts with specific queries including only subsets of data and need an instant response

⇒ Solution of the Hadoop/MapReduce-World: **YARN**

Big Data in Hadoop

Apache Hadoop - MapReduce 2.0 or YARN

- **YARN: Yet Another Resource Negotiator**
- ***idea***: split up the two major functionalities of the JobTracker (resource management and job scheduling/monitoring), into separate daemons:
  - ResourceManager
  - ApplicationMaster

- MasterNode runs the ResourceManager and on each slave runs a NodeManager → Data-computation framework.

- ApplicationMaster works with the NodeManager(s) to execute and monitor the tasks

Big Data in Hadoop

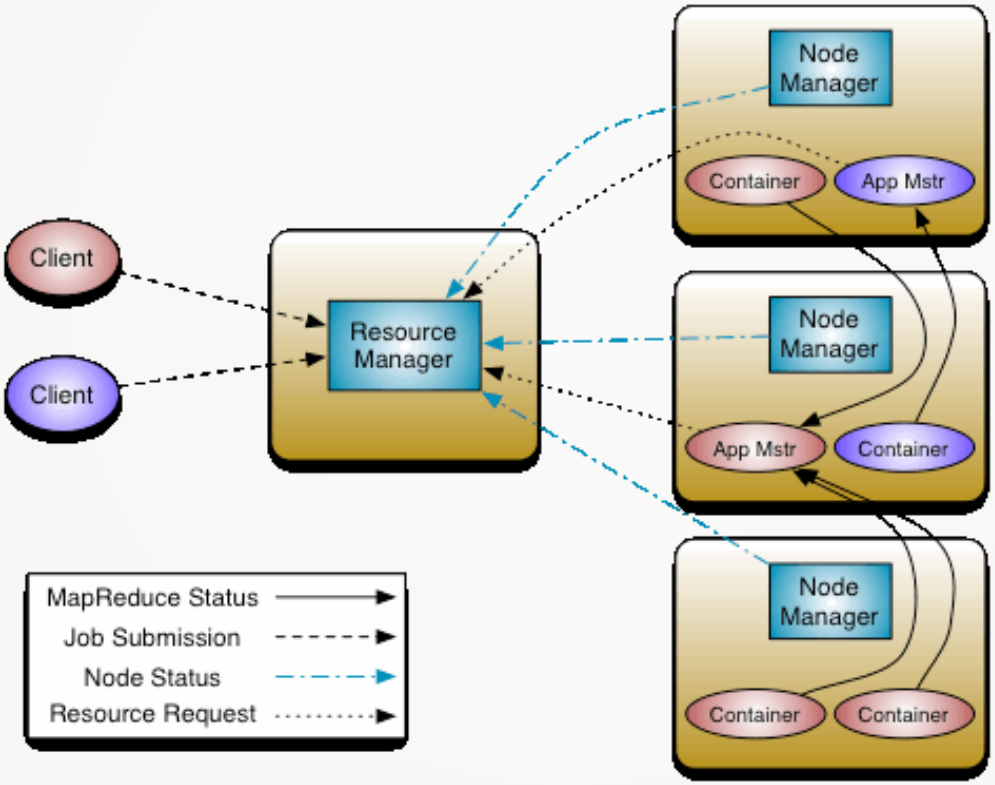
Apache Hadoop - Workflow of MapReduce 2.0 or YARN

General workflow of a YARN application:

1. Client submits application to Resource Manager
2. ResourceManager asks any NodeManager to create an ApplicationMaster which registers with the Resource Manager
3. ApplicationMaster determines how many resources are needed and requested the necessary resources from the ResourceManager
4. ResourceManager accepts the requests and queues it up
5. As the requests resources become available on slave nodes, the ResourceManager grants the ApplicationMaster requirements for containers on specific slave nodes

Big Data in Hadoop

Apache Hadoop - MapReduce 2.0 or YARN



Source: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html