Chapter 3:

Map Reduce / Hadoop / HDFS

Overview

Outline

- **Distributed File Systems (re-visited)**
- **MapReduce**
  - Motivation
  - Programming Model
  - Example Applications
- **Big Data in Apache™ Hadoop®**
  - HDFS
  - MapReduce in Hadoop
  - YARN

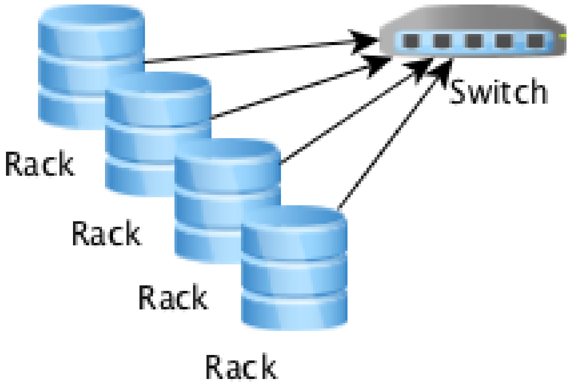
Outline

- **Distributed File Systems (re-visited)**
- **MapReduce**
  - Motivation
  - Programming Model
  - Example Applications

Today

- **Big Data in Apache™ Hadoop®**
  - HDFS
  - MapReduce in Hadoop
  - YARN

Next week



Distributed File Systems

- Difference to RDBMS
- Parallel Computing Architecture

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

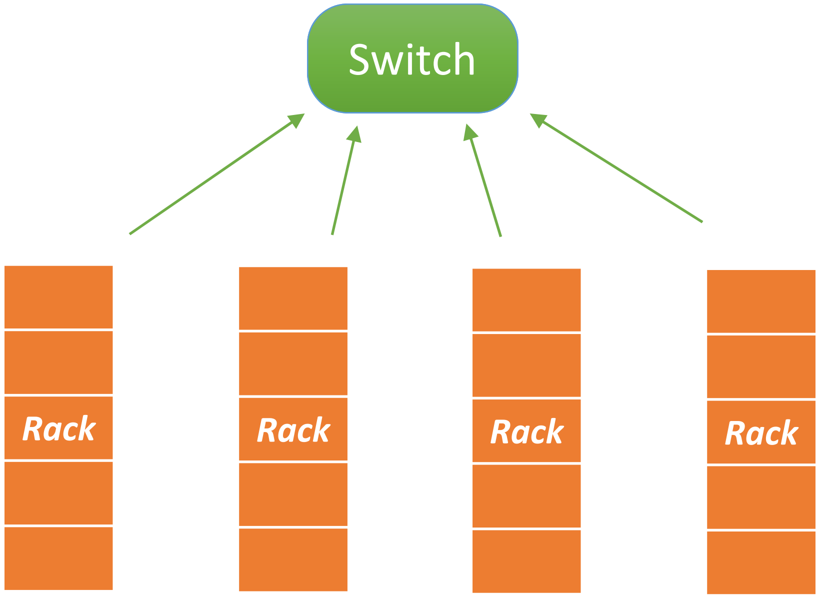
Distributed File Systems

- **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

Parallel computing architecture

- Referred as *cluster computing*
- Physical Organisation:
  - 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 Organisation:

- 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:

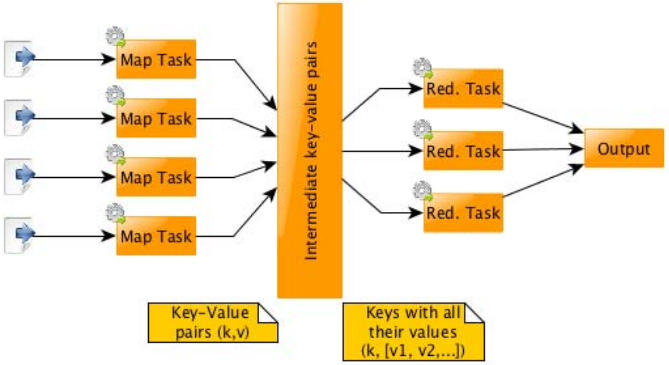
- Google File System (GFS)
- Hadoop Distributed File System (**HDFS**, by Apache)
- CloudStore
- HDF5
- S3 (Amazon EC2)
- …

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

Overview



MapReduce

- Motivation
- Programming Model
  - Recap Functional Programming
- Examples

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*

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

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), …

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)

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

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)]
```

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
```

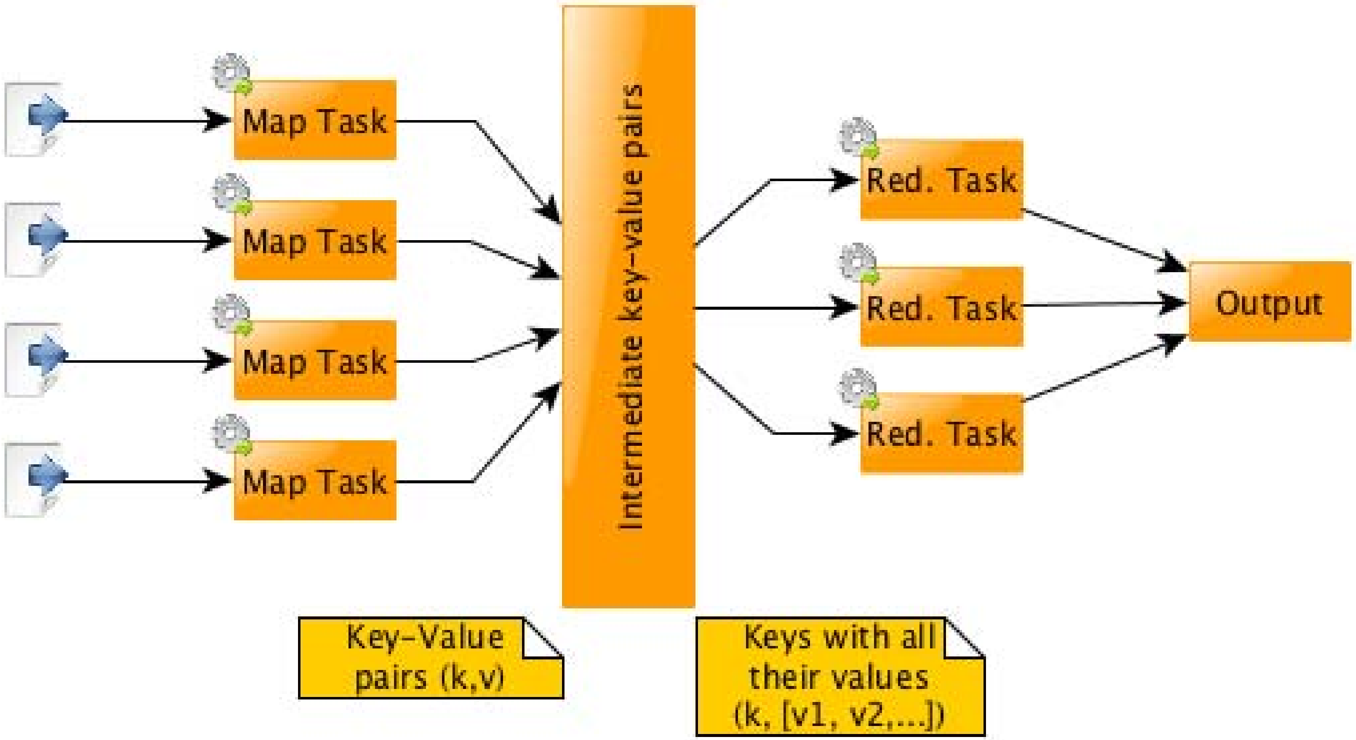
Programming Model - General Processing of MapReduce

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

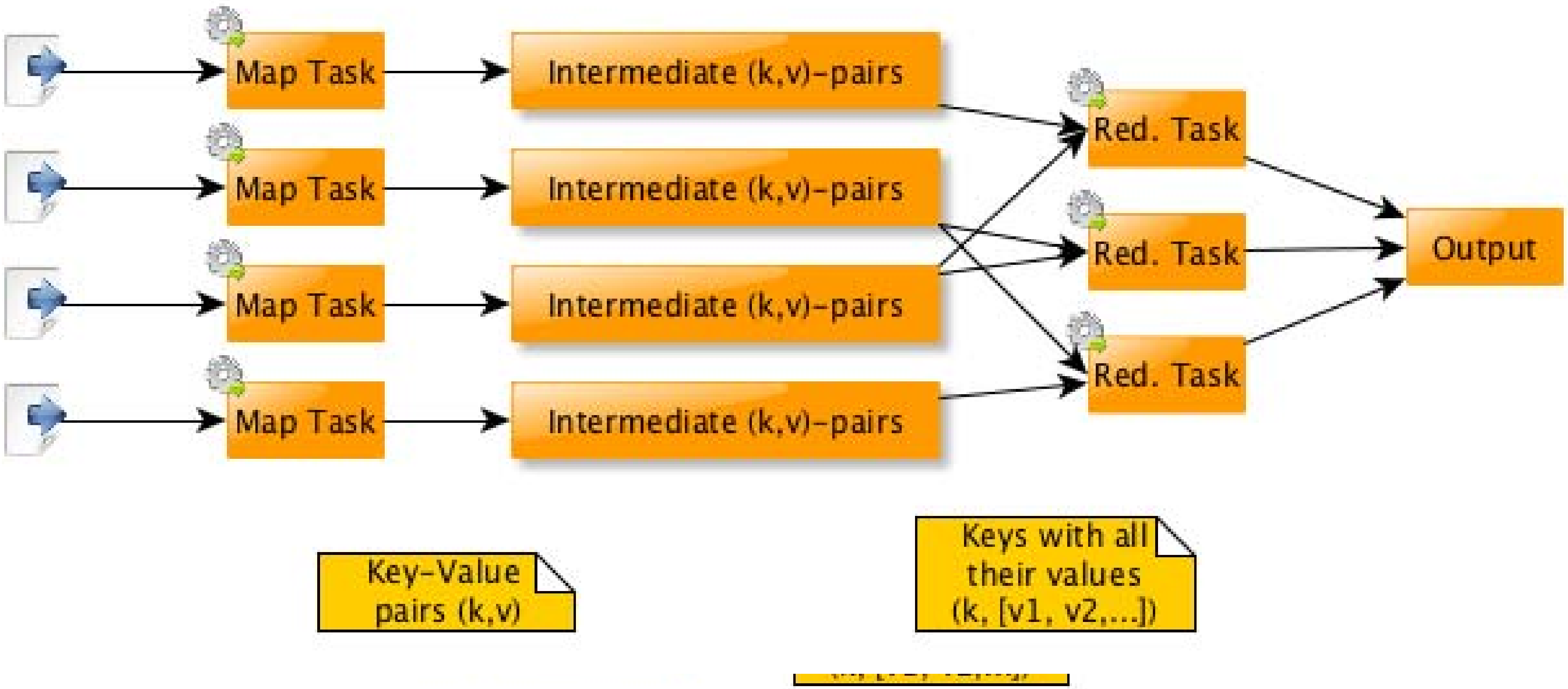
- 2. key-value pairs are collected by a master controller and sorted by key. The keys are divided among all Reduce tasks.

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

Programming Model - High-level MapReduce diagram



Programming Model - High-level MapReduce diagram



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**

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 storaged (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

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

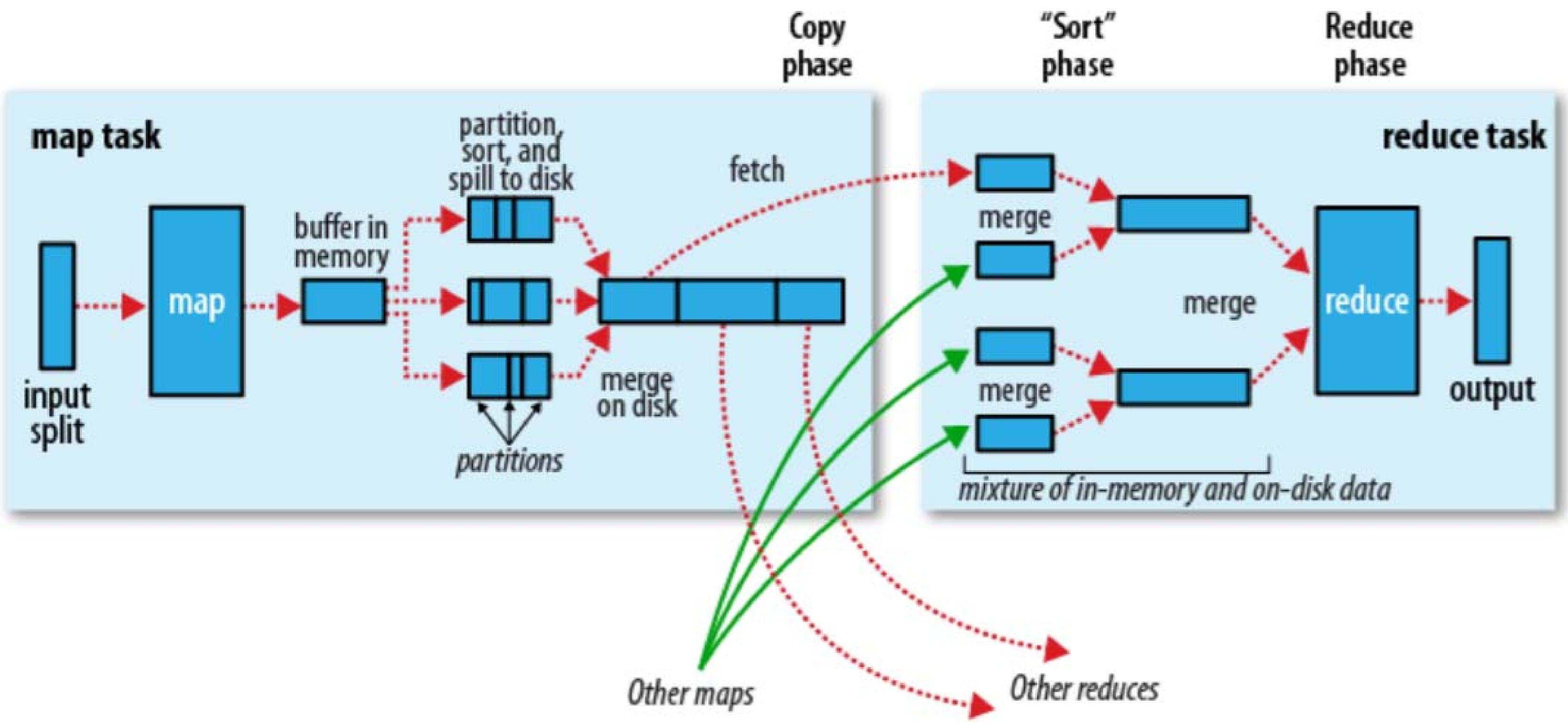
Programming Model - General Processing

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

Programming Model - General Processing

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



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

Programming Model - General Processing

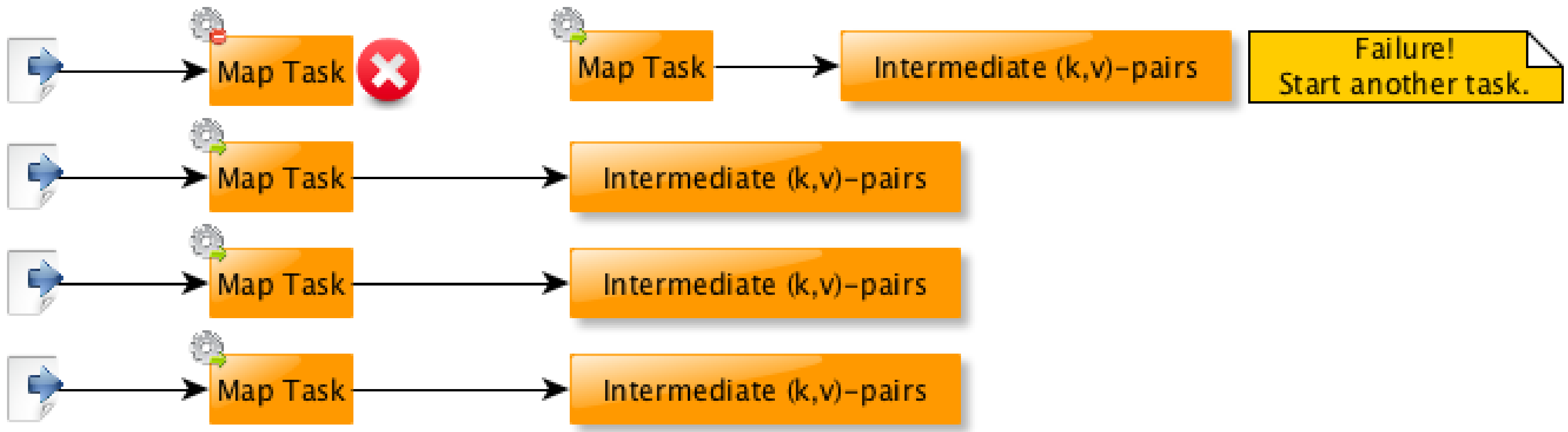
Handling machine **failures** and **stragglers**

- 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



Programming Model - General Processing

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

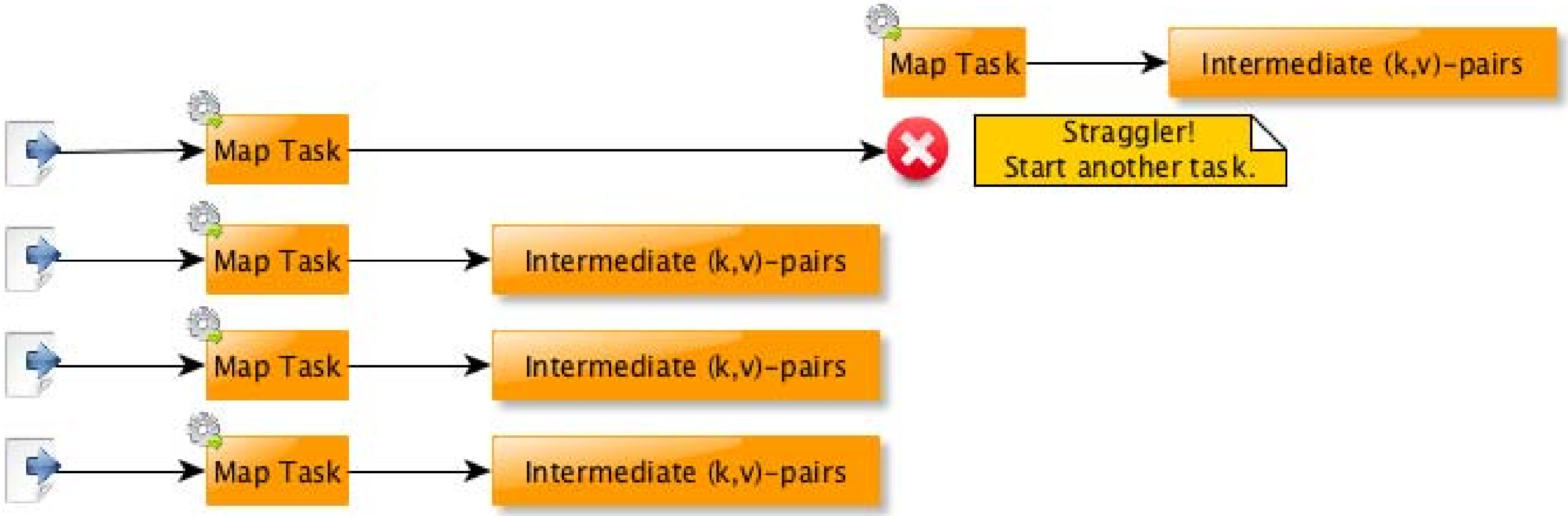
- **Stragglers**
    - slow workers lengthen the termination of a task
    - close to completion, backup copies of the remaining in-progress tasks are created
    - Causes: hardware degradation, software misconfiguration, …
    - if a task is running slower than expected, another equivalent task will 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**



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

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

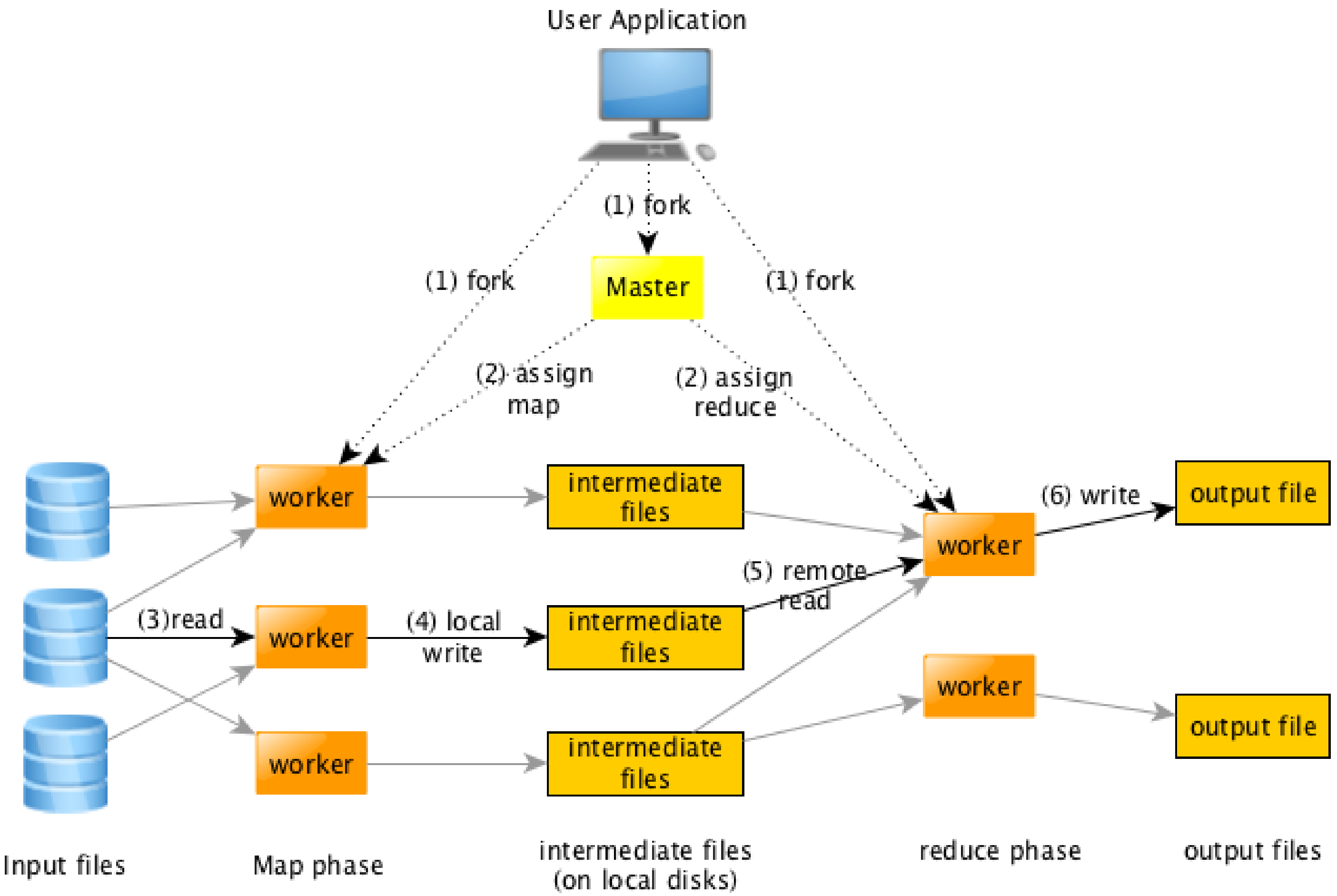
4. The buffered (key,value)-pairs are written to local disk. The location of these (intermediate) files are passed back to the master

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

—> On completion of all tasks, the master notifies the user program.

Programming Model - Low-level MapReduce diagram



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

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)

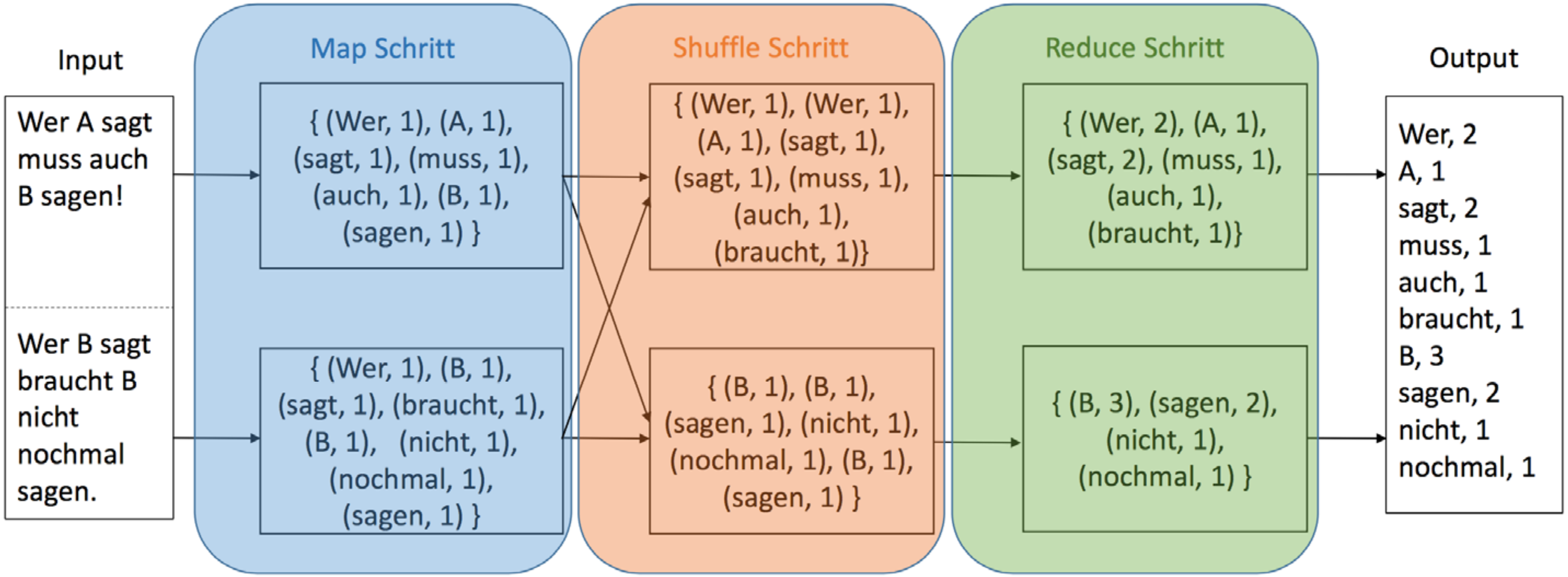
```
map (String key, String value):
    //key:document name
    //value: content of document
    for each word w in value do:
        emitIntermediate(w, "1")
```

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())
```

Example #1 WordCount

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



Example #2 k-Means

Randomly initialize k centers:

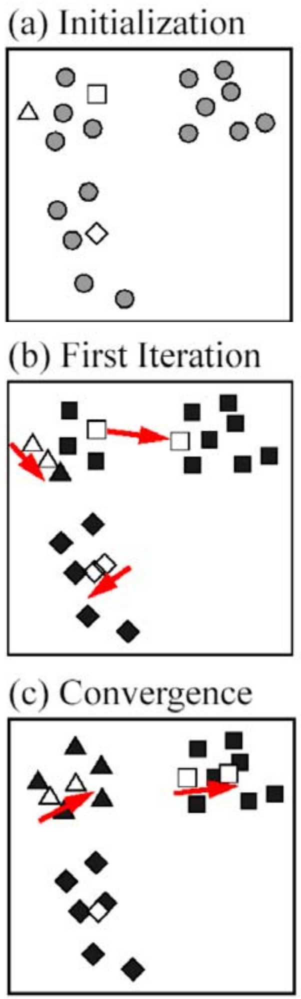
$$\mu^{(0)} = \mu_1^{(0)}, \ldots, \mu_k^{(0)}$$

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

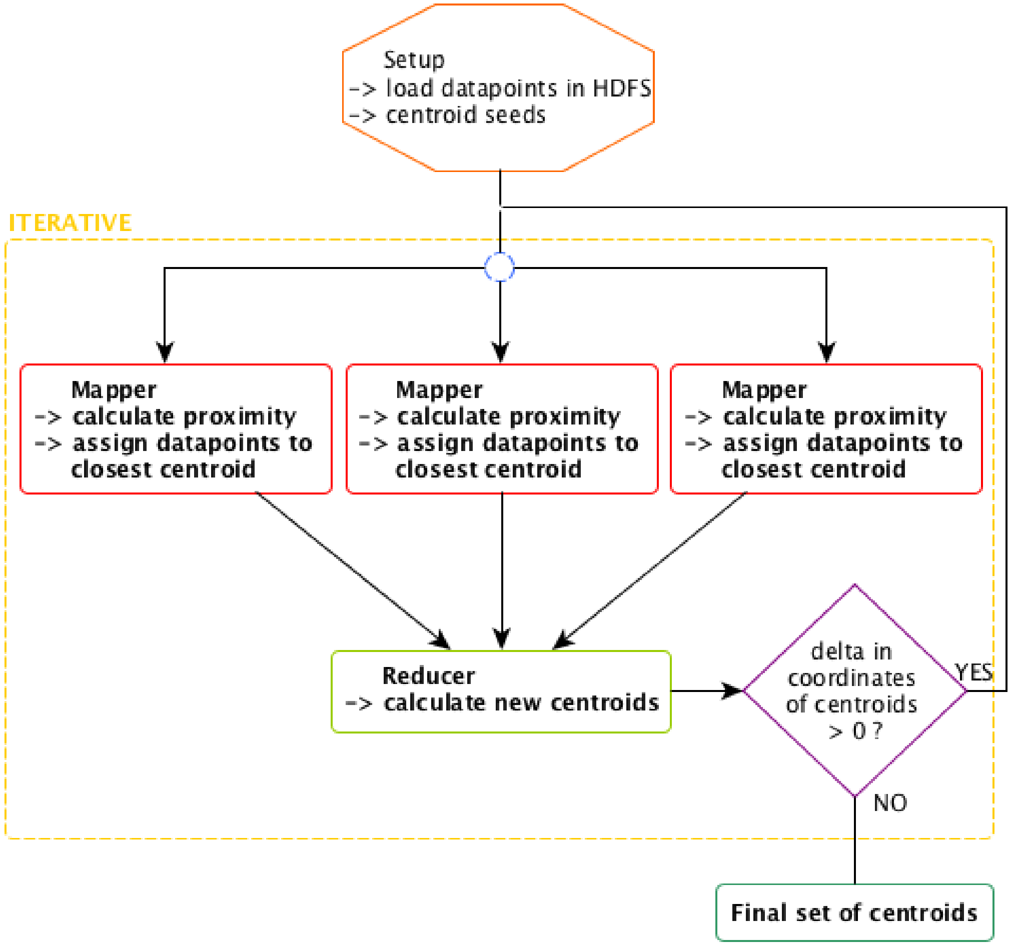
$$z^j \leftarrow \arg\min_i ||\mu_i - x^j||_2^2$$

Recenter: μ_i becomes centroid of its points:

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



Example #2 k-Means - MapReduce - Scheme



Example #2 k-Means - Classification Step As Map

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

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

Map:

Input:

- subset of d-dimensional objects of $M = \{x_1, \ldots, x_m\}$ 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

Example #2 k-Means - Classification Step As Map

Classify: Assign each point $j \in \{1, \ldots, m\}$ 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
```

Example #2 k-Means - Recenter Step as Reduce

Recenter: μ_i becomes centroid of its points:

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

Note: equivalent to averaging its 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

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