# Introduction to Data Analytics

## Hadoop

András Varga
*IBM Consulting*

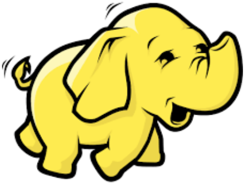
*Bratislava*, 2022

# Contents

# Business Card

**Definition**

Apache Hadoop is Hadoop is one of the first big data processing frameworks



Details:

- Provides a distributed filesystem
- A distributed execution framework
- Built on top of commodity hardware
- Open source

# Motivation, History and Trends

| | | |
|---|---|---|
| File System | Data Processing Framework | Job Scheduling Framework |
| Being replaced (e.g. S3) | Replaced by Spark | Superseded (e.g. Kubernetes, Mesos) |

# Processing Big Volumes of Data

- An abstraction of the processing:
  - fetch input from storage
  - load the data into memory
  - process the data
  - write back the results
- Memory is cheap, compared to CPU time
- In big volumes of data the network becomes a bottleneck
- Fast network devices are expensive
- It is cheaper to use commodity hardware, so failures are anticipated

# The Distributed File System

- A solution that addresses the challenges from the previous slide
- Google File System
- Hadoop Distributed File System

# Terminology

- Basic computation entities (computers, servers) are called *nodes*
- *Cluster* is a set of connected nodes working together and can be viewed as a single system
- *Data center* is a facility hosting computer systems
- File operations are sometimes called *mutations*

# Properties I

- The parts of the file system is built from cheap commodity parts (usually on some linux OS)
- As there may be several hundreds of nodes, failures are a expected
- Multi GB files are common, small files are rare
- Possibly millions of files (smaller numbers of big files is encouraged)

# Properties II

- Appending files is more common than rewriting them
- Sequential read is more efficient than seeks (searching specific position in the file)
- Multiple users can use the same system, concurrent file changes can happen
- Neither HDFS nor GFS present a general POSIX-compliant API
- File permissions in HDFS are only meant to prevent unintended operations

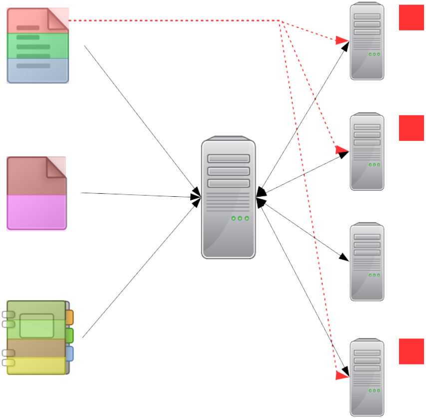
# Google File System

- A single cluster consists of
  - Single *GFS master* node
  - Multiple *chunkservers*
- *Clients* are accessing the cluster
- Clients and chunkservers can run on the same nodes
- Files are divided into fixed-size *chunks* (also called as *blocks*) with a globally unique 64 bit *chunk handle* assigned by the master (upon creation)
- By default each chunk is stored in 3 replicas
- Chunks are much bigger than usual OS block sizes (128 MB default)

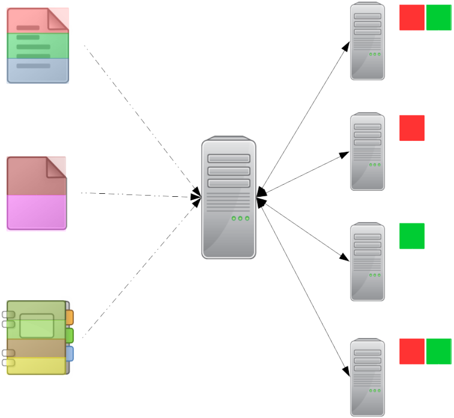
# GFS Node Roles

- GFS master
  - maintains all file system metadata (Access control, etc.)
  - garbage collection
  - chunk migration between chunkservers
  - *HeartBeat* communication between master and chunkserver to collect its state
- Client
  - implements the file system API and communicates with the master and chunkservers to read or write data
  - interact with the master for metadata
  - data-bearing communication goes directly to the chunkservers

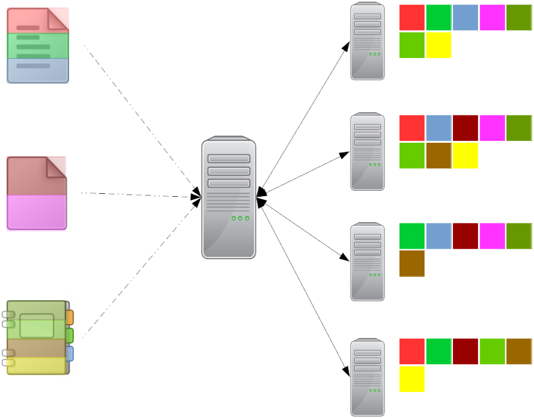
# Overview of GFS



# Overview of GFS



# Overview of GFS



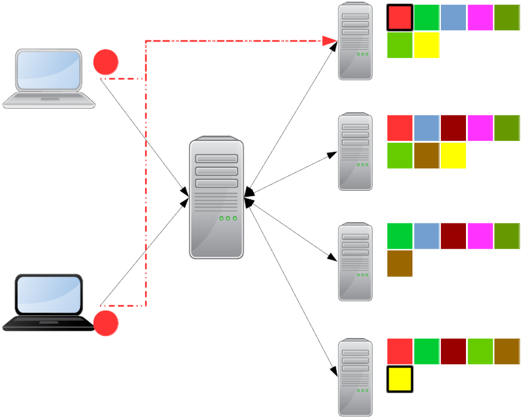
# Metadata

- Is kept in the memory of the master
- Metadata contains:
  - the file and chunk namespaces
  - the mapping from files to chunks
  - the locations of each chunk's replicas
- The first two are stored on the hard drive with logging mutations to an *operation log*
- The master does not store chunk location
- The master asks the chunkserver for chunk locations on startup (or when new chunkserver is added)

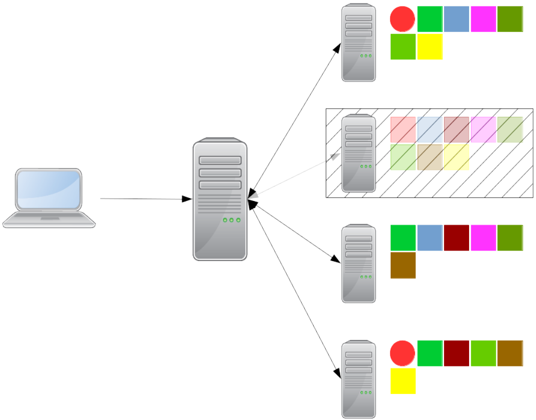
# Mutations

- File namespace mutations (e.g., file creation) are atomic
- Each (data) mutation is executed on all chunk replicas
- The master grants a chunk *lease* to one of the replicas (*primary*)
- The primary is responsible for the serialization of multiple concurrent mutations
- *Chunk version number* is used by the master to distinguish between up-to-date replicas and outdated ones (e.g. chunkserver failure)

# Mutation Example



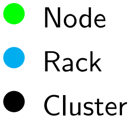
# Mutation Example With Unavailable Node



# HDFS

- Namenode $\leftrightsquigarrow$ GFS Master
- Datanode $\leftrightsquigarrow$ chunkserver
- In HDFS the clusters are built from racks, which in turn are built from nodes

# HDFS Cluster



# Failures of GFS Master/HDFS Namenode

- Brewer's CAP Theorem - in large-scale distributed systems, simultaneously providing consistency, availability, and partition tolerance is impossible
- In this case partitioning is unavoidable
- Real trade-off is between consistency and availability
- In a single master setup the consistency is provided but availability can not be guaranteed
- The chunks are stored on multiple nodes, but the master is not replaceable
- A Master/Namenode is a single point of failure
- Multiple Master nodes provide high availability

# Basic Concept of MapReduce

- A *divide and conquer* approach
- If sub-problems are independent, they can be processed in parallel
- Independent sub-problems can be assigned to different workers (nodes, processors, etc.)
- The result of each independent worker needs to be combined into the final result

# Interaction with DFS

- MapReduce does not necessarily require a distributed file system, but it provides many advantages
- DFS was not created solely for MapReduce Frameworks, the basic concept is unrelated
- DFS enables efficient speculative execution approach

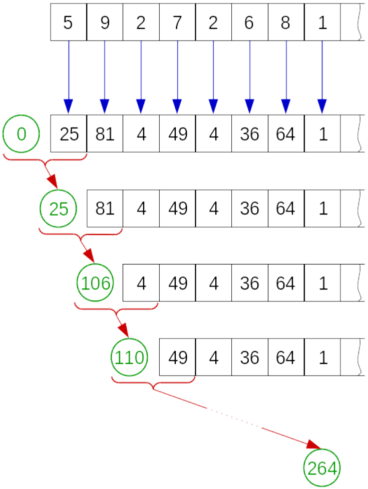
# Inspiring MapReduce

- *High-order functions* - Functions which can take other functions as arguments or return them as result
- Two examples, both working on a list of values:
  - *map*
    - Takes a function $f$ with one parameter as an argument
    - Applies $f$ to all elements in a given list
  - *fold*
    - Takes a function $g$ with two parameters as an argument + an initial value
    - Applies $g$ to the initial value and the first element on the list
    - Iteratively applies $g$ to the last intermediate result and the next element of the list

# map-fold Example

- Compute the sum of squares from the list
- map function takes parameter $\lambda x.x^2$
- fold function takes parameter $\lambda x \lambda y.x + y$
- fold function has an initial value 0

# Visualisation of map-fold




map $\lambda x.x^2$

fold (step 1) $\lambda x \lambda y.x+y$

fold (step 2) $\lambda x \lambda y.x+y$

fold (step 3) $\lambda x \lambda y.x+y$

fold (step n) $\lambda x \lambda y.x+y$

# High-level Overview of MapReduce

- Consists of 2 steps over large datasets
- First step: apply computation on datasets separately/in parallel
- Second step: apply aggregation over all precomputed intermediate results
- In MapReduce Framework programmers have to define the user-specific computation and the user-specific aggregation (like $f$ and $g$ from the map-fold example)

# Basic Data Structures

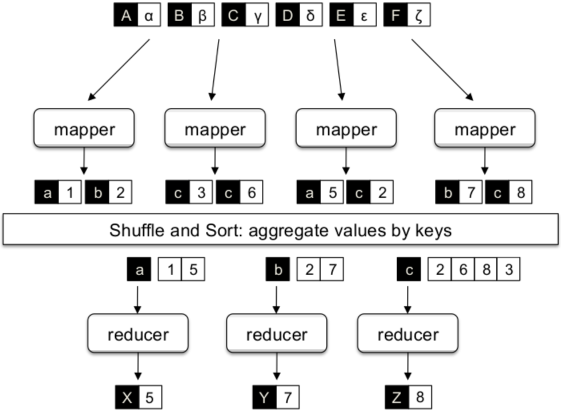
- Basic data types
  - primitives: integers, floating points, strings, raw data, ...
  - complex structures: tuples, lists, arrays, ...
- **Key-value pairs** built from basic data types
- Examples:

| | | |
|---|---|---|
| Web pages | *Key* - URLs | *Value* - HTML content |
| Files | *Key* - Filename | *Value* - content |
| Graphs | *Key* - Vertex | *Value* - list of neighbors |

# Mappers and Reducers

- The programmer defines a mapper and a reducer:
  - map: $(k_1, v_1) \to [(k_2, v_2)]$
  - reduce: $(k_2, [v_2]) \to [(k_3, v_3)]$
  - where $[\ldots]$ denotes a list
- Semantics:
  - The mapper is applied to every input key-value pair (split across an arbitrary number of files) to generate an arbitrary number of intermediate key-value pairs
  - The reducer is applied to all values associated with the same intermediate key to generate output key-value pairs

# MapReduce Schema



# Technical Details

- Reduce can be imagined as distributed "group by"
- Intermediate data arrives to each reducer in order, sorted by the key
- Intermediate key-value pairs are not preserved after the end of the MapReduce job
- Output key-value pairs are written persistently onto the file system
- The output usually appears as $r$ files, where $r$ is the number of reducers

# The Execution Framework

- The MapReduce Framework separates the code from *distributed processing* (*the execution framework*)
- The developer submits the job to the submission node of a cluster
- The execution framework (sometimes called the "runtime") takes care of everything else

# Scheduling - Motivation

- Each MapReduce job is divided into *tasks* (Map task, Reduce task,...)
- In large jobs, the total number of tasks may exceed the number of tasks that can run on the cluster concurrently
- Therefore a *task queue* is needed
- Coordination among tasks belonging to different jobs (and users) is mandatory

# Scheduling - Speculative Execution

- The Map phase of a job is as long as the slowest map task
- Similarly the reduce phase is as long as the slowest reduce task
- These slowest tasks are the so called *stragglers*

# Speculative Execution - Handling the Stragglers

- Identical copy of the same task is executed on different machines, and the framework uses the result of the fastest instance
- More efficient with Map tasks as Reduce needs data from the network
- Resolves problem with insufficient hardware
- Does not solve problems, when **data is not distributed properly amongst the nodes**

# Data-Code Co-location

- Basic idea: **move the code, not the data**
- The scheduler will start the code on a node that holds the data
- This is not always possible (e.g. already too big workload on a given node)
- Solution is to start the code on a different node and stream the data there

# Synchronization

- There is a "barrier" between Map and Reduce phases
- "shuffle and sort" - distributed sort of intermediate key-value pairs, which involves copying intermediate data over the network
- $m$ mappers and $r$ reducers involves up to $m \times r$ distinct copy operations

# Partitioners and Combiners

- The above is a simplified view
- In reality there are 2 additional elements: *partitioners* and *combiners*

# Partitioners

- Are responsible for splitting up the intermediate key space and assigning intermediate key-value pairs to reducers
- Specifies the (reduce) task to which an intermediate key-value pair must be copied
- Keys are processed in sorted order

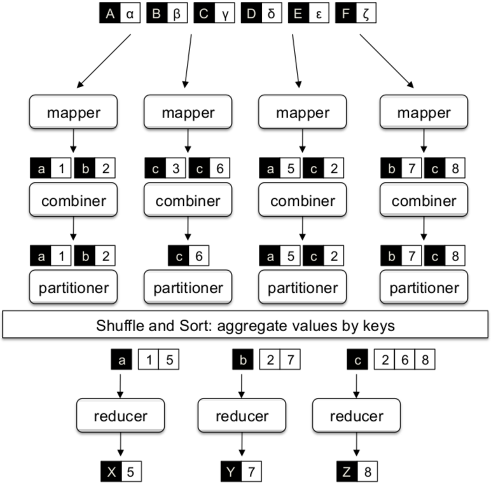
# Default Partitioner Method

- Simplest/Default method: compute the hash value of the key mod by number of reducers
- Copies the key-value pair to the reducer with ID computed as above
- Ignores the value of the key value pair $\rightsquigarrow$ may yield *large differences* in the number of key-values pairs assigned to the reducer nodes

# Combiners

- Are an optimization in MapReduce
- A local aggregation before the shuffle and sort phase
- Motivation: Once Mapper is finished intermediate key-value pairs are copied across the network
- Solution: Local aggregation of the result emitted by a specific Mapper can reduce the size of the data
- Operates in isolation, reading only the output of the assigned mapper (running on the same node)
- Not necessarily have the opportunity to process *all* values associated with the same key
- Therefore the *correctness* of the Job *can not rely* on Combiners

# Full MapReduce Schema



# Translating Algorithms into MapReduce jobs

- Some algorithms cannot be implemented as a single MapReduce job
- Solution: Decomposition into a sequence of MapReduce jobs executed consecutively

# One's First Hadoop Program

**Problem Statement (WordCount)**

*Count the number of occurrences of each word from a file/set of files.*

- The "Hello World" of Hadoop
- Technical details of Hadoop are highlighted on this example
- Basic optimization techniques can be easily displayed

# Naive Implementation - Mapper

```
public static class TokenizerMapper extends
          Mapper<Object, Text, Text, IntWritable>{

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

  public void map(Object key, Text value, Context context)
          throws IOException, InterruptedException {
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      context.write(word, one);
    }
  }
}
```

# Naive Implementation - Reducer

```
public static class IntSumReducer extends
           Reducer<Text, IntWritable, Text, IntWritable> {
  private IntWritable result = new IntWritable();

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

# Submitting a Job

- When the Job is submitted:
  - The job's jar is copied into the distributed filesystem
  - The input is "prepared"
- Some of the additional options:
  - Jobs can be submitted into queues
  - Jobs can be chained
  - Monitoring settings can be configured
- Technical details shall be presented during lab sessions

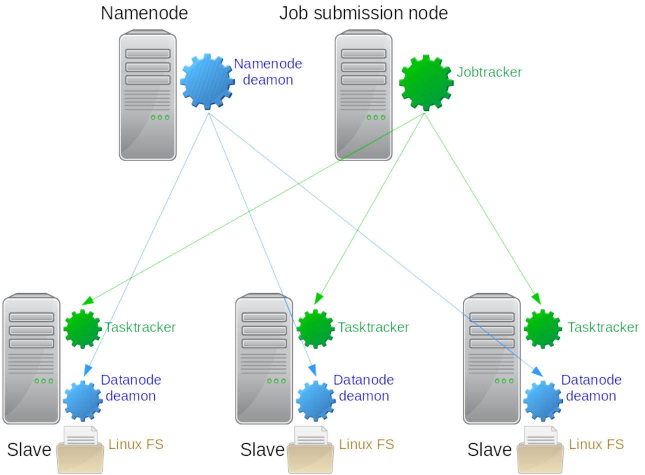
# MapReduce Version 1 I

- An older execution framework for Hadoop
- Consists of a single JobTracker and several TaskTrackers
- Both trackers are persistent, not related to any specific job or task
- JobTracker:
    - Primary user interface to a MapReduce cluster ("MapReduce master")
    - Handles the distribution and management of tasks
    - Often paired with the Namenode (hosted on the same machine)
    - Sends out heartbeats to all TaskTrackers to maintain an up to date table of available TaskTrackers

# MapReduce Version 1 II

- Jobs are broken down into tasks: Map task and Reduce task
- Each task is assigned by the JobTracker to a TaskTracker, handling the execution of the task
- TaskTrackers:
  - Provides execution services for the submitted jobs ("MapReduce worker/slave")
  - Manages the execution of tasks on an individual computation node
  - One instance of this server is running on each computation node (usually) paired with the HDFS Datanodes

# Execution in MapReduce Version 1



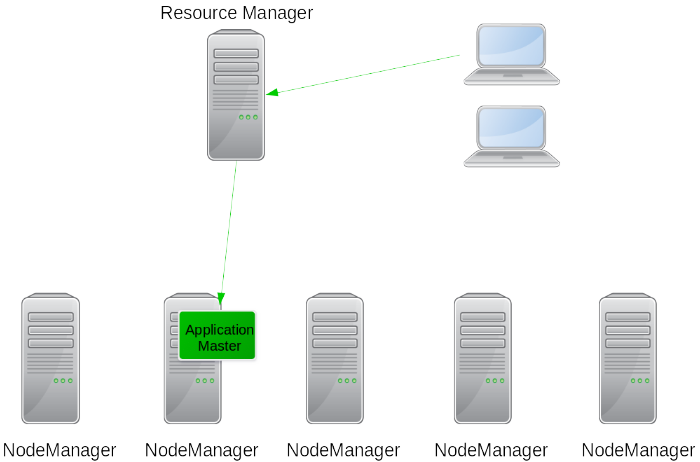
# Limitations of MapReduce (v1)

- Only one JobTracker $\rightsquigarrow$ scalability
- JobTracker has two responsibilities
  - Management of computational resources
  - Coordination of all tasks running on a cluster
- Supporting different kind of workload as MapReduce
- Solution: Yet Another Resource Negotiator [YARN]

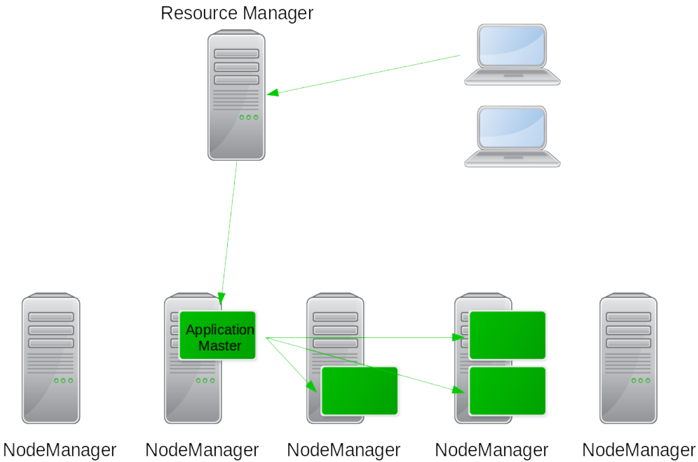
# MapReduce Version 2 - YARN

- Idea: splitting up JobTracker
- Resource manager
  - Global as a master daemon
  - Tracks available nodes and resources
- Application manager
  - Started when an application/job is submitted
  - Coordinates execution of tasks, speculative executions
  - Handles failures of tasks
  - Each job has its own application manager instance
- Nodemanager
  - More generic than TaskTracker
  - Works using dynamically created resource containers

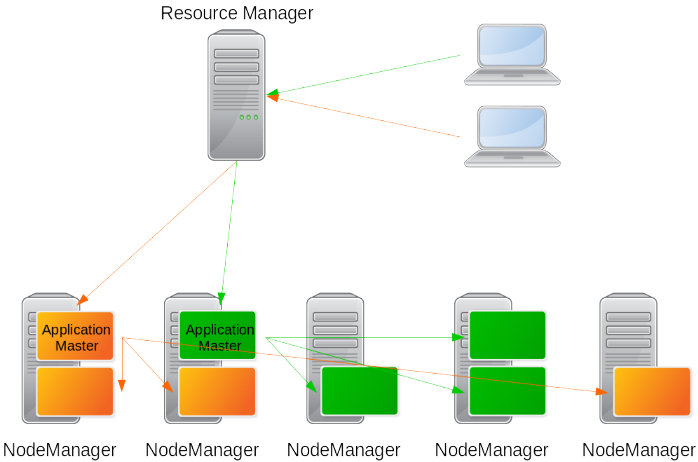
# Execution with YARN



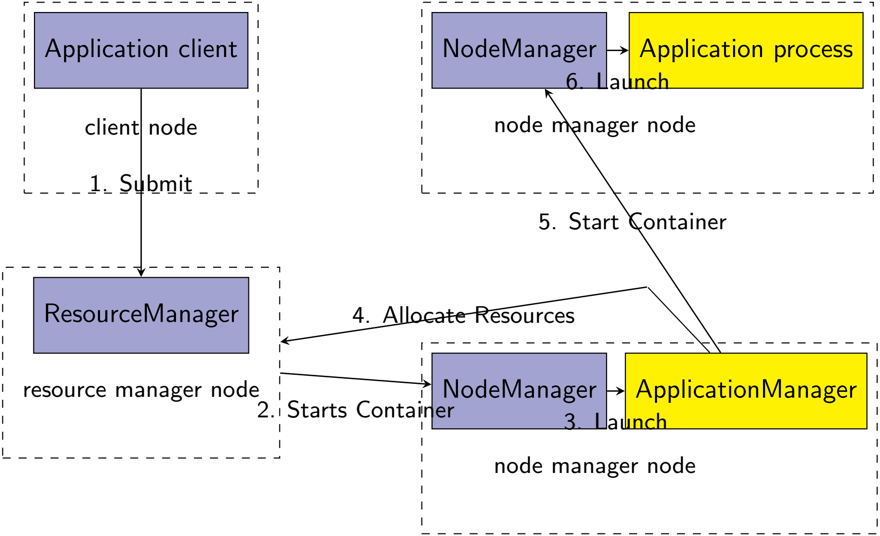
# Execution with YARN



# Execution with YARN



# Diagram - Start of YARN Application



# MapReduce Version 1 vs YARN

| MRv1 | YARN |
|---|---|
| Cluster Manager | Resource Manager |
| JobTracker | ApplicationMaster<br>(dedicated and short lived) |
| TaskTracker | NodeManager |
| MapReduce Job | Distributed Application |
| Slot | Container |

# YARN Properties

- Application manager is no longer a bottleneck
- Containers are general purpose - in fact Application managers run in them
- Resource Manager is a bottleneck
- True High Availability can be achieved using Apache Mesos

# Basics of Reading Data

- Input data is usually stored in HDFS, hence split into chunks
- Data is not read directly
- Data is tokenized - split into words using whitespace by default
- Tokens are provided to Mapper for computation
- Shortcoming of the WordCount program
  - ⇝ Words **elephant** and **elephant.** are considered to be different
- During initialization the data is split into so called *input splits*, which are being sent to dedicated Mappers

# Local Aggregation

Motivation:

- During the shuffle and sort stage the intermediate results are often transferred via network
- Network latencies are relatively expensive compared to other operations
- In Hadoop, intermediate results are written to local disk before being sent over the network
- Reductions in the amount of intermediate data translate into increases in algorithmic efficiency
- Effective technique for dealing with reduce stragglers (As counting some words can be much slower than other words)

# Possible Local Aggregations

- Use of combiners
- In-Mapper aggregation
  - It is not a supported part of the system
  - In-mapper aggregation drawback: Needs memory to store intermediate results

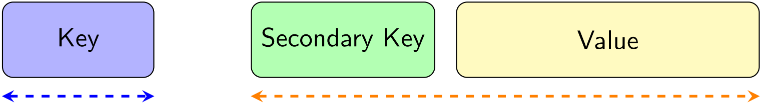
# Secondary Sorting

- Shuffle and sort phase - is very convenient if computations inside the reducer rely on an ordering of keys
- But: How can we sort by value?
- Google's MapReduce provides a built in option for secondary sorting

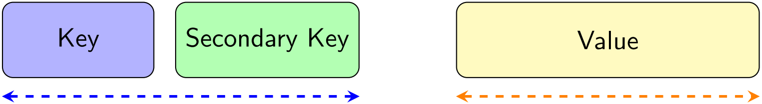
# Secondary Sorting Other Solutions

- In memory buffering and sorting is a scalability bottleneck
- *Value-to-key conversion* - a general design pattern for secondary sorting
  - Move part of the value into the intermediate key to form a composite key
  - Let the sorting to MapReduce, with a correctly defined order
  - Custom partitioner is needed so the real key from the emitted complex key is taken into account when shuffling to reducers
- This approach can be generalized to any number of secondary sorting

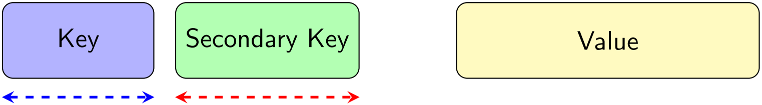
# Secondary Sorting




In the mapper the part of value is moved to the key:




Using the partitioner assign each pair in accordance with the original key, secondary key is ignored:



# Graphs

- Different problem than text processing
- Documents may exist in the context of some underlying network
- Examples:
  - Social Graphs (Twitter, Facebook, etc.)
  - Transportation networks
  - Graphs created by transactions (money transfers, etc.)
- The main goal is to create *scalable* algorithms for graph processing

# Graph Representations

- Usual graph representations:
  - Adjacency matrix
  - Incidence matrix
  - Edge lists
  - Adjacency lists
- Common algorithms are based on adjacency matrix

# Adjacency matrix

- A square matrix $M$, $m_{ij}$ represents the edge from node $n_i$ to node $n_j$
- A handy representation for linear algebra
- Can be too huge to store in memory
- Inefficient for *sparse* graphs, holding several 0s as most of the edges do not exist
- Social and web graphs are usually sparse
- Solution for big data: *Adjacency list*

## Incidence matrix

- For a graph with $V = \{v_1, \ldots, v_n\}$ vertices and $E = \{e_1, \ldots, e_m\}$ edges
- The incidence matrix is an $n \times m$ matrix, where $x_{ij}$ represents vertex $v_i$ being incidental with edge $e_j$
- Orientated graph can be represented by enabling more than 2 values (True, False)
- Too huge, rarely used

# Edge lists

- For a graph with list of edges $E$ the edges are split into 2 groups:
  - Oriented edges
  - Unoriented edges
- Edge lists are a representation where each edge is represented as a pair of vertices incidental with it $(v_1, v_2)$ given as two lists, one for oriented one for unoriented edges
- Unoriented edges may be split into two oriented once
- Weights can be added as a third "column" into the list
- Compact for sparse graphs
- Time consuming to find all edges related to a given vertex

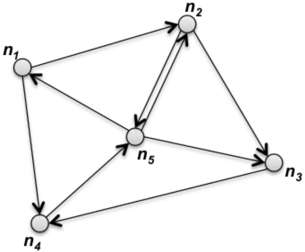
# Adjacency List

- For each node $n$ from the graph there is a list containing all nodes $n_i$, such that an edge $(n, n_i)$ exists
- May be directed or undirected
- There are two possibilities to encode undirected graphs:
  - Each undirected edge can be stored as a pair of directed edges
  - Or the edges can be ordered in some order and each edge will be stored once in the adjacency list of the vertex with smaller label

# Adjacency Matrix vs Adjacency List

- Using adjacency lists it is a more complex problem to find the list of incoming edges for a given vertex, whereas it can be done easily using the adjacency matrix
- For sparse graphs the list representation is more efficient
- For dense graphs the matrix is more compact

# Example Graph




| | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ |
|---|---|---|---|---|---|
| $n_1$ | 0 | 1 | 0 | 1 | 0 |
| $n_2$ | 0 | 0 | 1 | 0 | 1 |
| $n_3$ | 0 | 0 | 0 | 1 | 0 |
| $n_4$ | 0 | 0 | 0 | 0 | 1 |
| $n_5$ | 1 | 1 | 1 | 0 | 0 |

adjacency matrix

| | |
|---|---|
| $n_1$ | $[n_2, n_4]$ |
| $n_2$ | $[n_3, n_5]$ |
| $n_3$ | $[n_4]$ |
| $n_4$ | $[n_5]$ |
| $n_5$ | $[n_1, n_2, n_3]$ |

adjacency lists

# References I


J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*.
Morgan & Claypool Publishers, 2010.

"Apache$^{TM}$ Hadoop® Official Web Page."

"Big Data University."

"Hadoop official tutorial."

J. Plesník, *Grafové Algoritmy*.
SAV, 1983.

"Apache Giraph Official Web," 2018.

S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *ACM*, 2003.