Data Intensive Systems (DIS)
KBH-SW7 E25

4. MapReduce

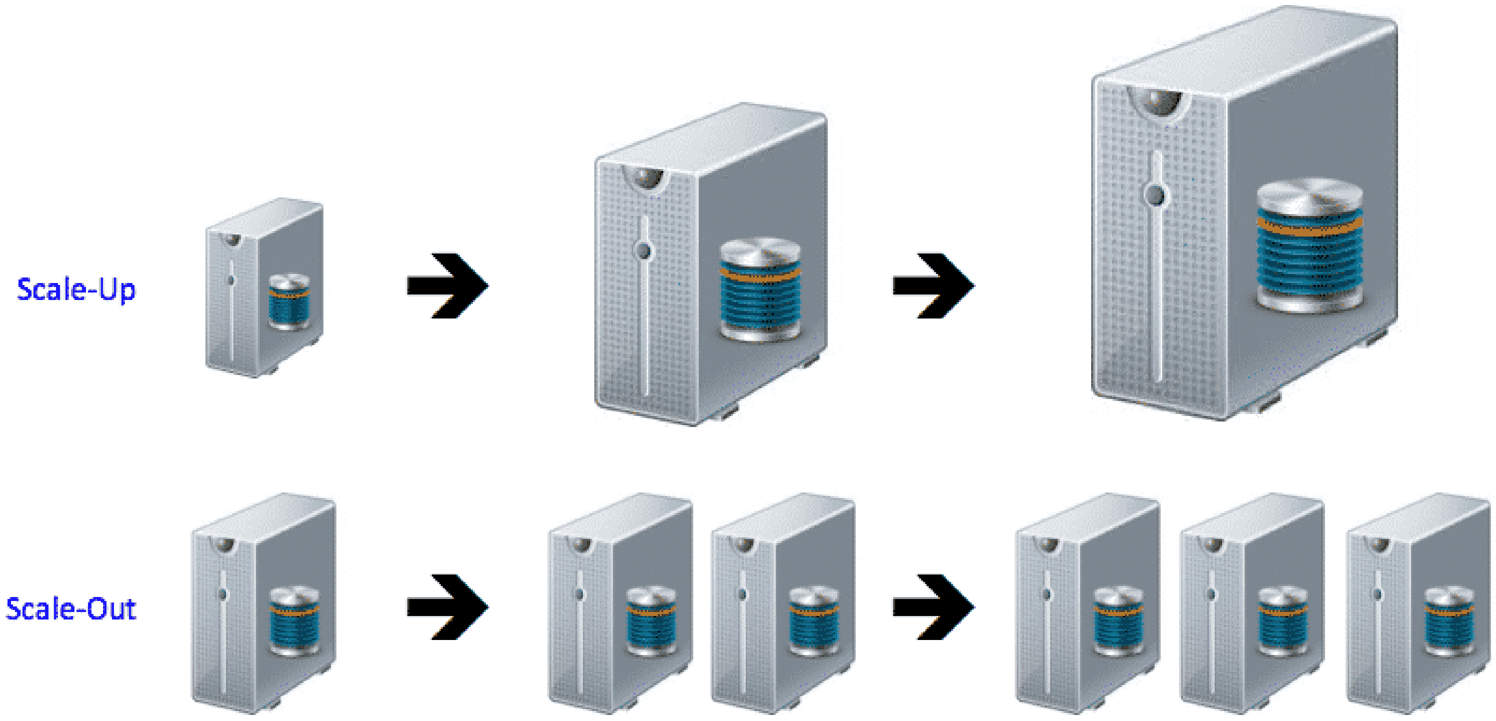
AALBORG UNIVERSITY

Agenda

- Background
    - Scalability in computation
    - Big ideas behind MapReduce
- Programming model
- Execution framework
- Software implementation
- MapReduce and databases

Scale-up vs. Scale-out

- Demand for efficiency is the drive



https://www.lebigdata.fr/stockage-scale-out-nas-definition

Scale-up vs. Scale-out

- Scalability is a system’s ability to swiftly enlarge or reduce the power or size of computing, storage, or networking infrastructure.
- In a data system, scalability means its ability to process larger amounts of data and maybe also at a faster pace.
- To achieve scalability, we need to add capacity to the infrastructure.
- **Scale-up**: adding more (virtual) resources, e.g., equipping a computer with more/better CPUs, more RAMs, and larger hard disks.
- **Scale-out**: adding more computers to spread the workload across more machines. This form *clusters* of machine.

Scale-up (Vertical Scaling)

- When to scale up
  - When there’s a performance impact, e.g., limited I/O and CPU capacity increase latency and cause performance bottleneck.
  - When storage optimization does not work, i.e., no room for better on the current resources.
- Pros
  - Relative speed up. E.g., a dual processor, a faster DRAM
  - Simplicity. System architecture remains the same.
  - Cost-effectiveness: cheaper than scale-out with many machines
  - Limited energy consumption
- Cons
  - Latency
  - Labor and risks
  - Aging hardware

Scale-out (Horizontal Scaling)

- When to scale out
  - When a long-term scaling strategy is needed.
  - When upgrades need to be flexible.
  - When storage workloads need to be distributed.
- Pros
  - To use newer server technologies
  - Adaptability to demand changes
  - Cost management (incremental, predictable):
    - We can use commodity low-end servers. This is significantly cheaper than buying high-end servers.
- Cons
  - Limited rack space
  - Increased operational costs
  - Higher upfront costs

A cluster of
low-end machines

Programming

- Programming for parallel & distributed systems is very difficult
  - Concurrency control: Race conditions, deadlocks, etc.
  - A programmer needs to spend considerable efforts on ‘infrastructural details’ than on the real problem.
- Ideally
  - Programmers should be liberated from such (system-level) infrastructural details shared by a large class of, if not all, problems.
  - A solution should scale with input data.
    - It should take twice long, if the data size doubles.
    - It should take the same time on a twice big cluster, if the data size doubles.

Big Ideas behind MapReduce

- Scale out, not up
  - Make use of large number of commodity, low-end machines
- Fault-tolerance
  - Assume failures are common
- Move processing to the data
  - Not the other way around, which is time-consuming
- Process data *sequentially* and avoid random access
  - Suitable for some particular types of applications, not all
- Hide system-level details from the application developers
- Seamless scalability

3 Concepts of MapReduce

- The term of MapReduce can refer to three distinct but related concepts.
- Programming model
  - How are developers supposed to program?
- Execution framework
  - Runtime that coordinates the execution of programs written in the MapReduce paradigm
- Software implementation
  - The implementation of the previous two: Google, Hadoop and others.

Agenda

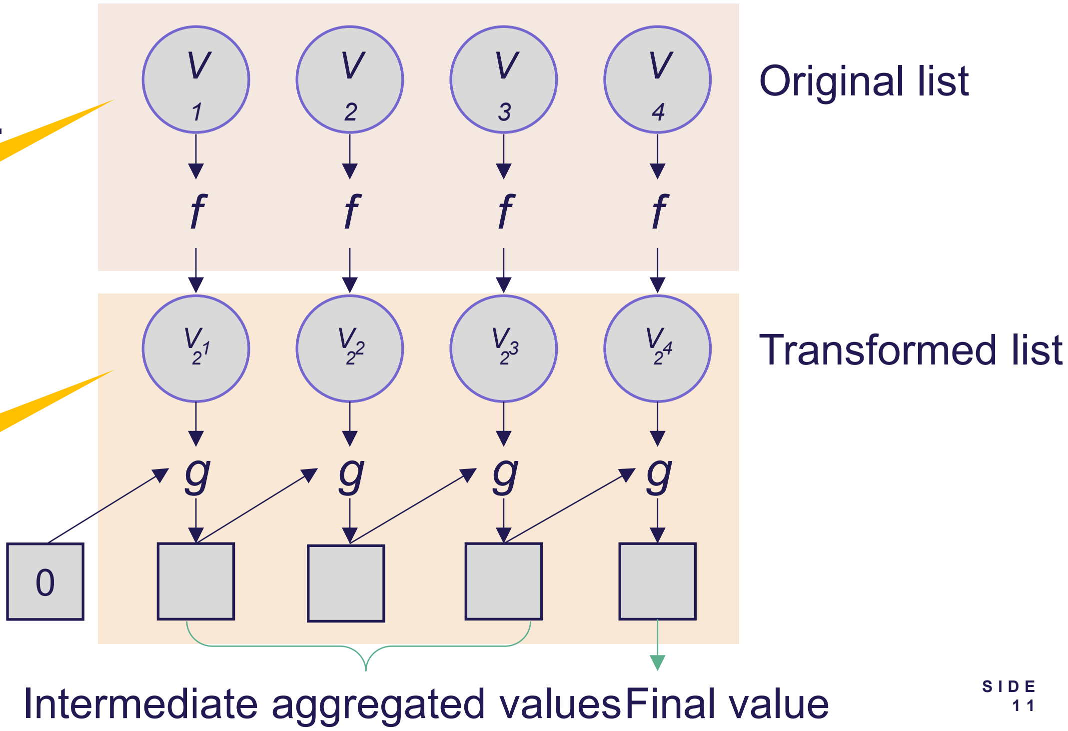
- Background
- Programming model
- Execution framework
- Software implementation
- MapReduce and databases

Functional Programming Roots

- Higher-order functions, functions that accept other functions as arguments
  - Two common built-in functions *map* and *fold*
- Consider a 4d vector $\boldsymbol{v}(v_1, v_2, v_3, v_4)$. To get $|\boldsymbol{v}|$, we need to compute Σv_i^2.
  - **Transformation**: *map* applies $f(x)=x^2$ to each v_i.
  - **Aggregation**: *fold* applies $g(x_1, x_2)=x_1+x_2$ to v_i^2s.

map phase ⟺ Transformation can always be parallelized.

reduce phase ⟺ Aggregation may enjoy *group-based* parallelism.



MapReduce **Functional Programming**

MapReduce Programming Model

- **Input**: A set of key/value pairs
- **Output**: Another set of key/value pairs
- Keys and values can be *primitives* or *complex types*
- A progammer implements two functions: **map** and **reduce**
- **map:** `(k1, v1) → list(k2, v2)`
  - Takes an input pair and produces a set of intermediate key/value pairs.
  - MapReduce groups all intermediate pairs with the same key and gives them to *reduce.*
- **reduce:** `(k2,list(v2)) → list(k3,v3)(Hadoop)`
  `list(v2)(Google)`
  - Takes an intermediate key and the set of all values for that key.
  - Merges the values to form a smaller set (typically empty or with a single value)

Example: WordCount

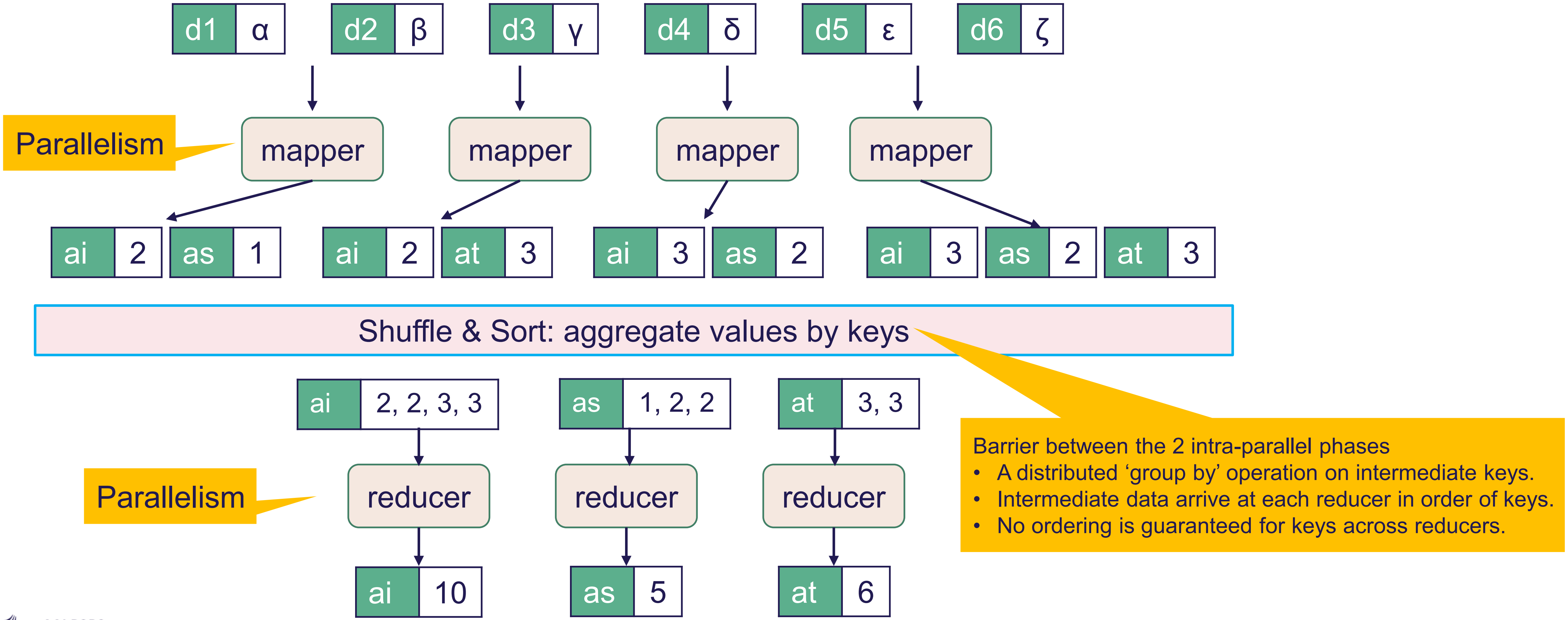
- Data: A large collection of documents
- Expected output: *#occurrence* of *each* word across *all* documents.
- Parallelism could be easily achieved.
  - A number of mappers can work on different documents in parallel.
    - For each word they see, they output an intermediate count (of 1) for the word.
  - When the mappers all finish, a number of reducers can work in parallel.
    - A reducer gets all counts for a certain word and ‘reduces’ the list of counts (i.e., 1’s) to a single value.

Map and Reduce for WordCount

```
map(String key, String value):
  // key: document id; value: document contents
  for each word w in value:
     emit(w, count)

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

A Simplified Conceptual View



Other Examples

- Counting URL accesses
  - Very similar to the WordCount example
  - The map function processes log files and outputs (URL, 1) for each access to URL
  - The reduce function adds all intermediate 1’s for each URL
- Reverse web-link graph
  - The map function is given a document’s URL as key, and the content as value. For each referenced target, it outputs (target, URL)
  - The reduce functions just outputs the list of URLs that reference a given target

Demo in Python

- Problem
  - Given a large list of strings, return the longest string
- StringLengthMR.py
  - This file implements four versions of MapReduce based functions
- DIS-E23-Lecture4_MapReduce_strings.ipynb
  - This is the 'main' file that calls these functions
- This format of modularity is needed to use multiprocessing in Jupyter Notebook
  - To 'simulate' or utilize multiple processors for MapReduce

Agenda

- Background
- Programming model
- Execution framework
- Software implementation
- MapReduce and databases

Execution Framework

- MapReduce separates the *what* of distributed processing from the *how*.
- A MapReduce program (a job) consists of
  - Code for mappers and reducers
    - And combiners and partitioners to be covered shortly
  - Configuration parameters
    - E.g., where the input is and where the output should go
- The developer submits the job to the *submission node* of a cluster
  - It's called jobtracker in Hadoop
- **Execution framework** (a.k.a. runtime) takes care of everything else *transparently*:
  - All other aspects of distributed code execution
  - On clusters ranging from a single node to thousands of nodes

MapReduce Runtime Responsibilities

- Scheduling
  - Each job is divided into smaller units called tasks.
    - For map: splits of input key-value pairs
    - For reducer: division of the intermediate key space.
  - Tasks are assigned to nodes in the cluster.
  - Coordination may be needed among tasks and/or nodes.
- Data/code co-location
  - For data locality, the scheduler starts tasks on the node having the needed data in its local storage.
  - If it's not possible, new tasks will be started elsewhere with the data streamed into via the network. Intra-rack transmission is preferred over inter-rack transmission.

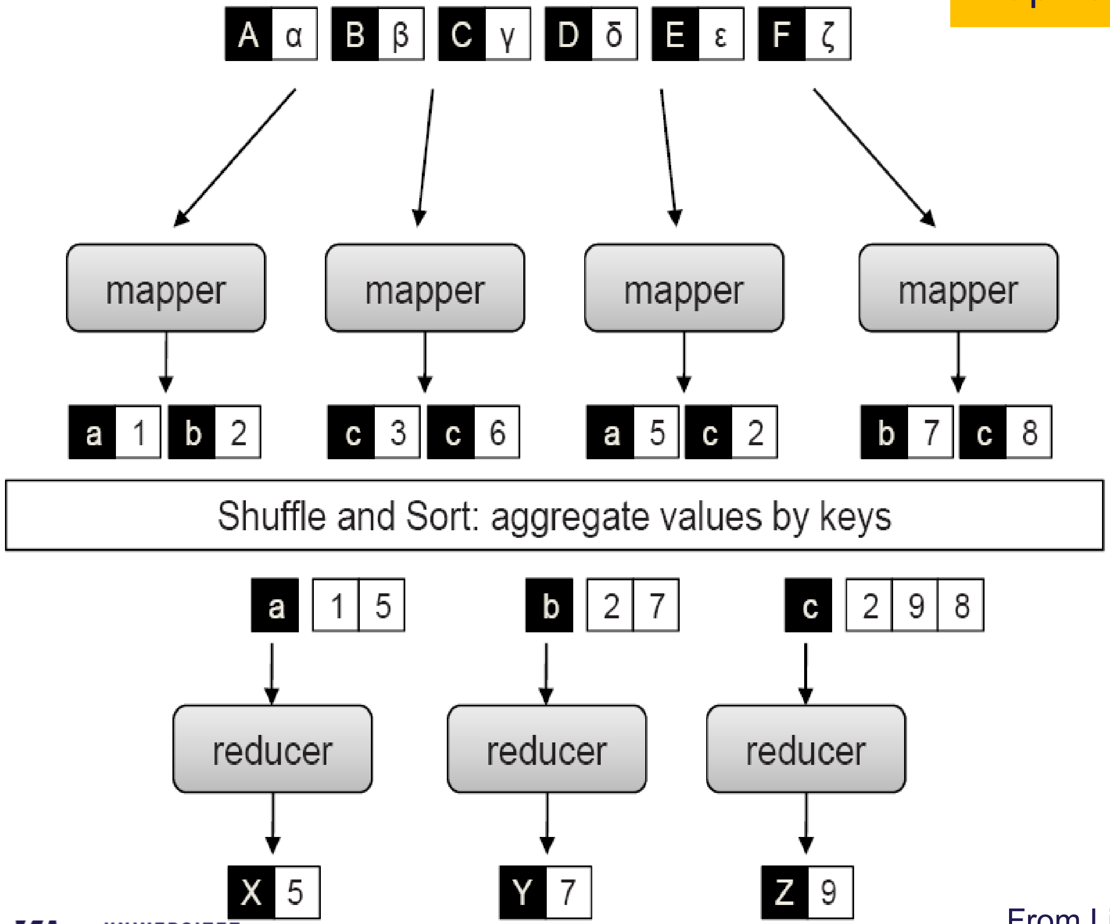
MapReduce Runtime Responsibilities, cont.

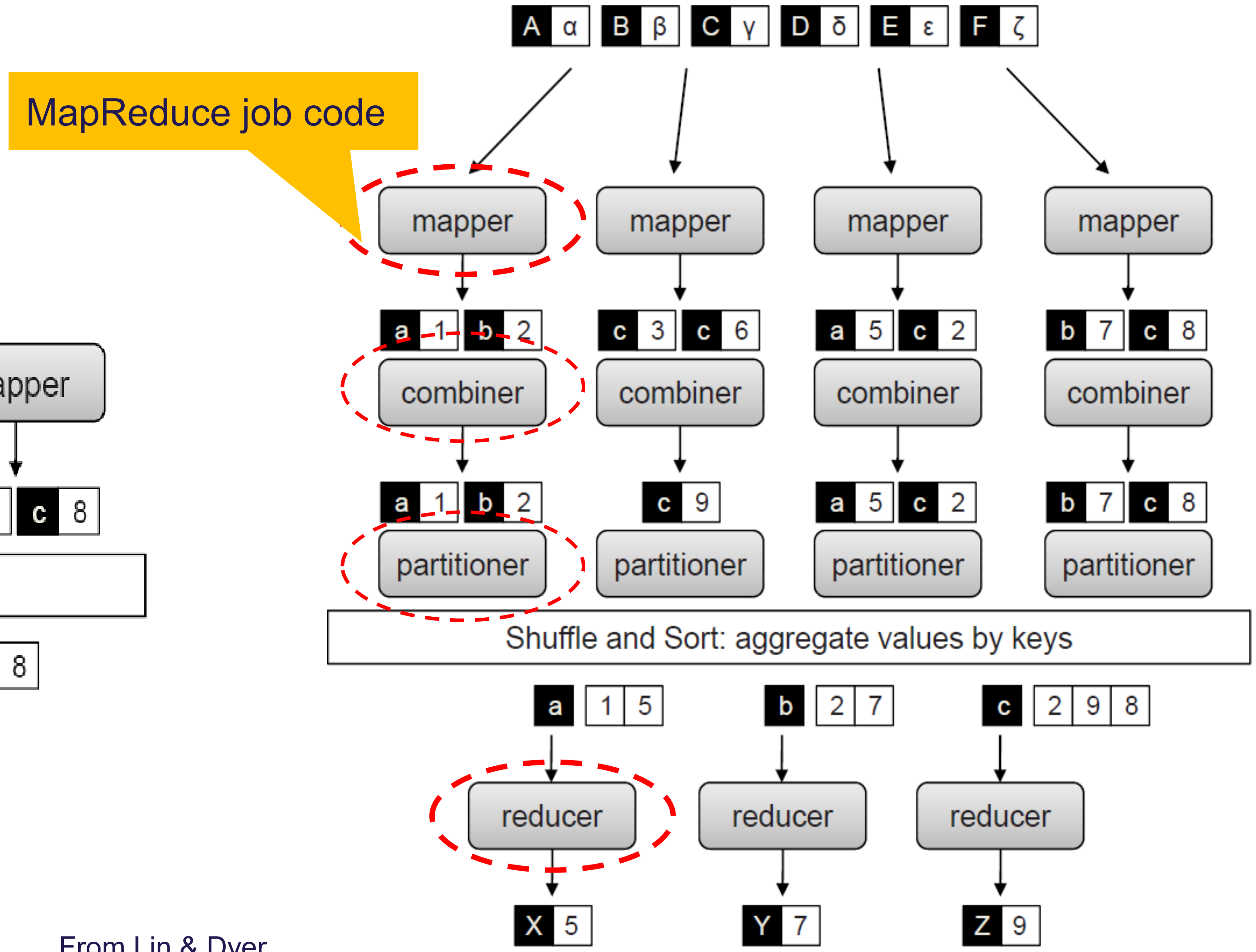
- Synchronization
  - ‘Shuffle and sort’ is a barrier between the map and reduce phases.
  - It involves copying immediate data over the network.
  - A job with *m* mappers and *r* reducers involves up to $m \times r$ distinct copy operations.
  - Unlike the *fold* operation, the reduce computation cannot start until
    1. all the mappers have finished emitting key-value pairs *and*
    2. all intermediate key-value pairs have been shuffled and sorted.
- Error and fault handling
  - All responsibilities above are in an error- and fault-prone environment.
    - Think of hardware failures of low-end commodity servers.
    - Software bugs and data errors

Combiners and Partitioners

- Combiners are an optimization that allows for *local aggregation* before Shuffle and Sort.
  - This helps reduce the number of intermediate key-value pairs.
  - Can be regarded as ‘mini-reducers’.
  - Recall the WordCount example
- Partitioners divide up the intermediate key space and assign intermediate key-value pairs to reducers.
  - A partitioner specifies the task (and thus the node to be determined by the runtime) to which an intermediate key-value pair must be copied.
    - The information is for the Shuffle and Sort phase to copy the data to the right place.
  - Hashing is the default partitioning function on the intermediate keys.

The Conceptual View Again





From Lin & Dyer

Agenda

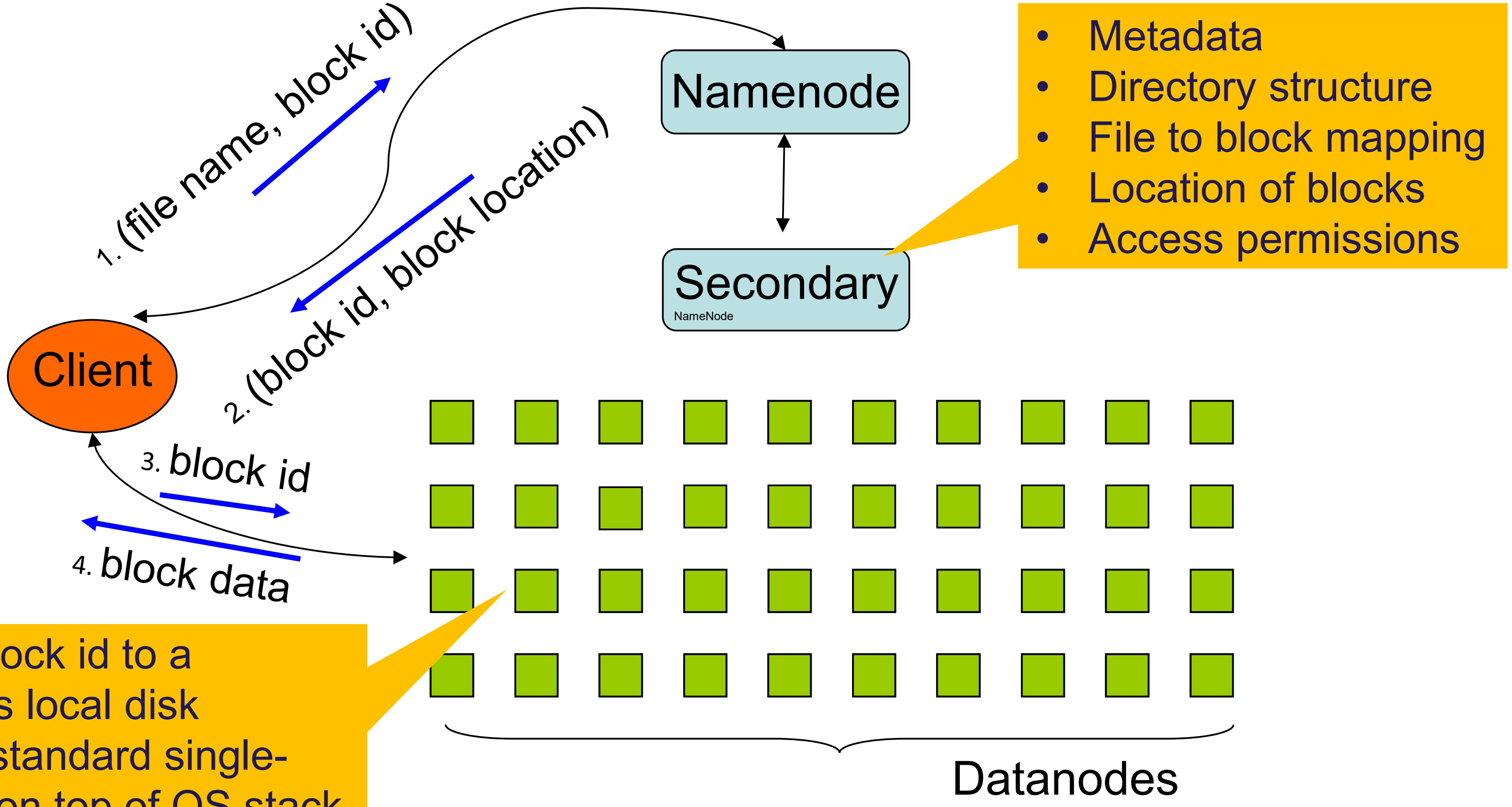
- Background
- Programming model
- Execution framework
- Software implementation
- MapReduce and databases

The Distributed File System (DFS)

- In traditional clusters, computation and storage are separated as distinct components.
- DFS abandons the separation and makes data considerably closer to computation, which helps reduce overall processing time.
  - DFS divides user data into blocks and replicates them across the local disks of nodes in the cluster.
    - DFS uses significantly large block sizes.
  - DFS adopts a **master-slave** architecture
    - The master maintains the file namespace
    - The slaves manage the actual data blocks.
- MapReduce can work without DFS, but many advantages will disappear without an underlying DFS or the like.
  - Google File System (GFS)
  - Hadoop Distributed File System (HDFS)

HDFS Architecture

- Master
  - Namenode
- Slaves
  - Datanodes



- A datanode maps a block id to a physical location on its local disk
- Blocks are stored on standard single-machine file systems on top of OS stack
- Data is never moved through namenode

Figure from S. Sudarshan, IIT Bombay

More Details

- By default, HDFS stores *three* separate copies of each data block to ensure reliability, availability and performance.
  - In large clusters, the three replicas are spread across different physical racks.
- HDFS files are immutable---they only accept appends.
  - WORM (Write Once Read Many)
- HDFS namenode's responsibility
  - Namespace management
  - Coordination file operations
  - Maintaining overall health of the file system
- The namenode forms a single point of failure. So, a warm standby namenode is often used in case that the primary namenode fails.

Concurrent Clients of HDFS

- Read from a replic
- Write to all replicat

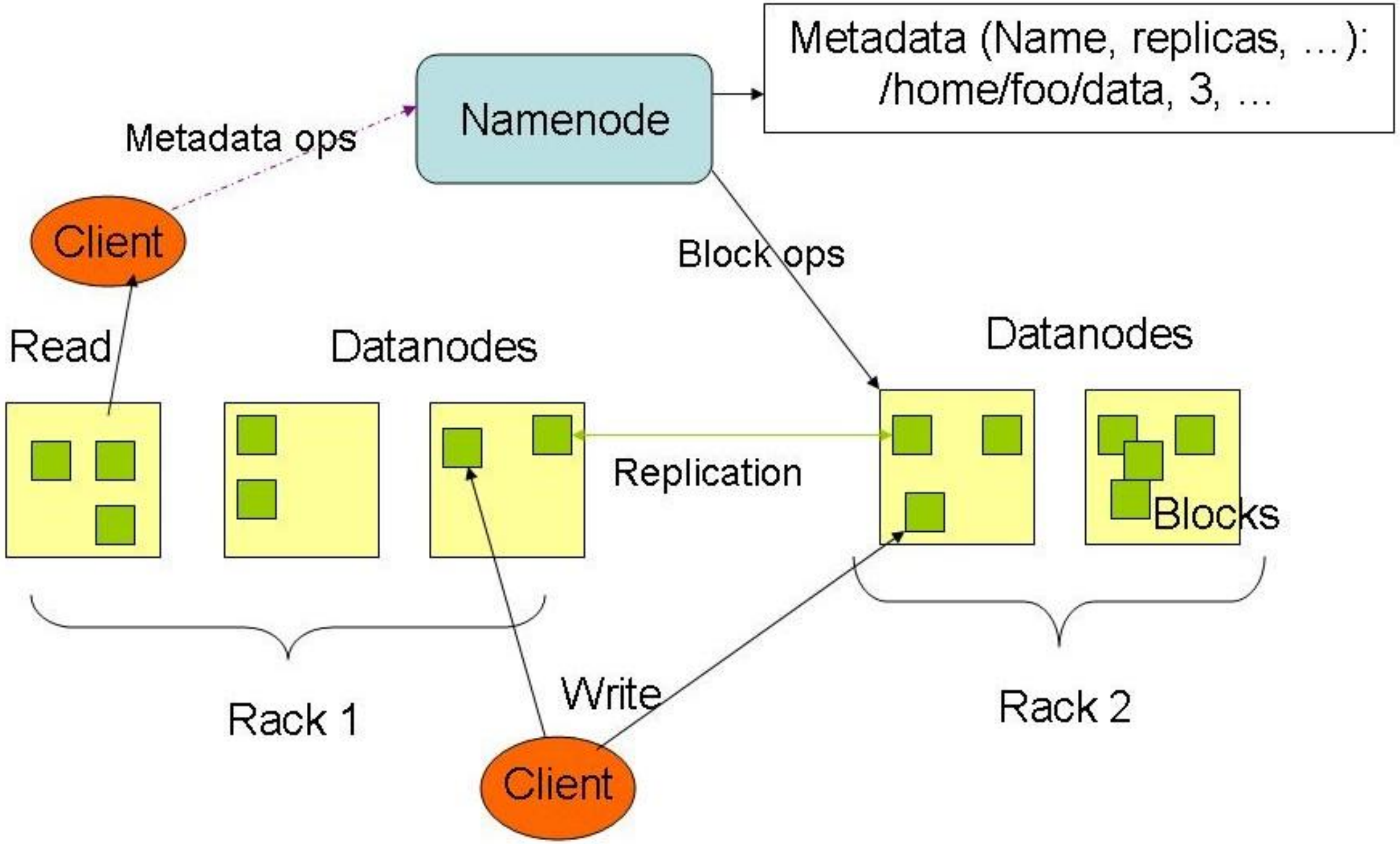


Figure from S. Sudarshan, IIT Bombay

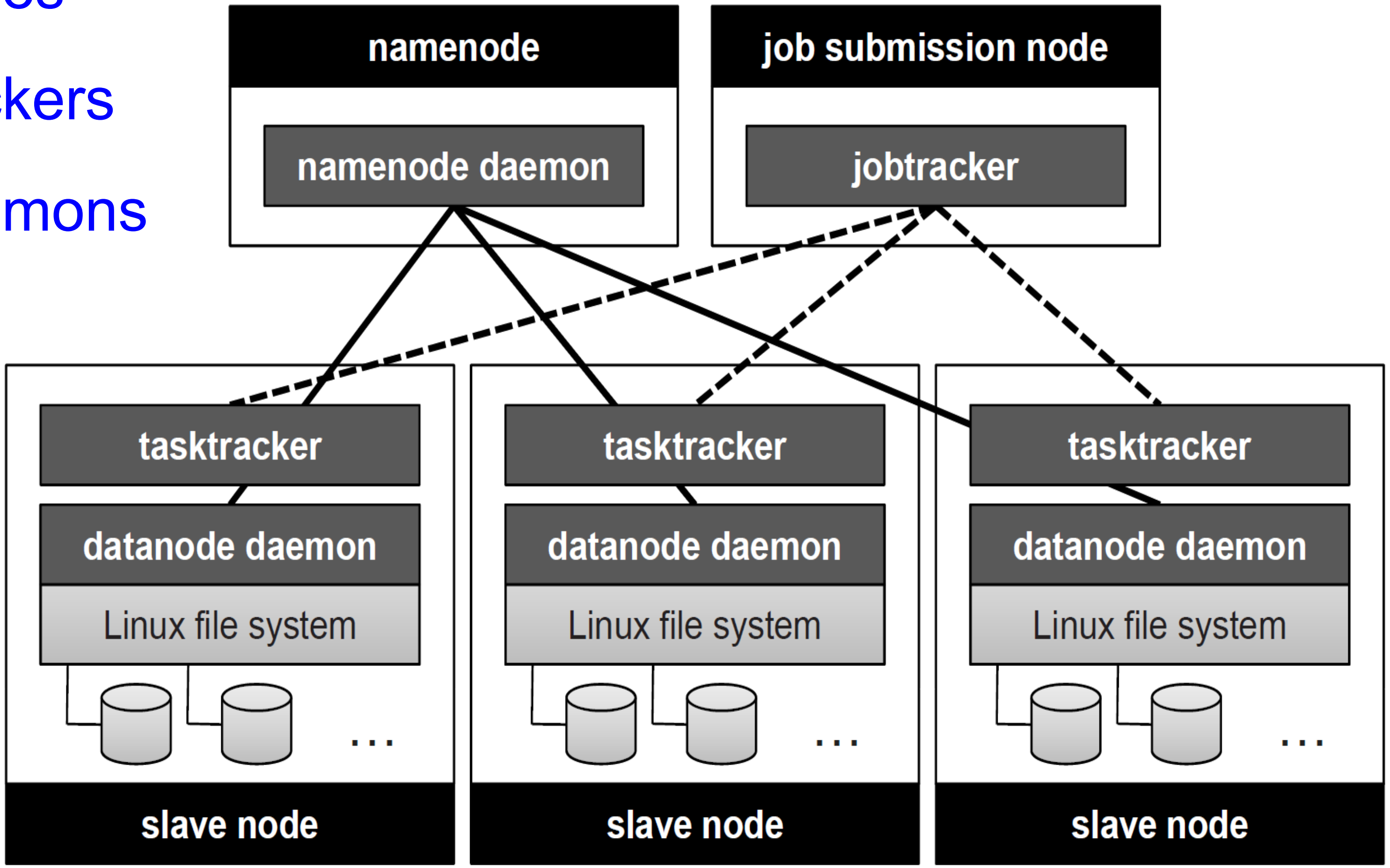
Assumptions behind GFS and HDFS

- The file system stores a relatively modest number of large files.
  - Large multi-block files are favored than too many small files
- Workloads are *batch* oriented, dominated by long streaming reads and large sequential writes.
  - Batch operations on large chunks of data without data caching.
- Applications are aware of the characteristics of the distributed files system.
  - Data management is pushed onto the end application.
- The file system is deployed in an environment of cooperative users.
  - Security is essentially not a concern in MapReduce.
- The system is built from unreliable but inexpensive commodity machines.
  - Self-monitoring and self-healing mechanisms are needed.

Hadoop Cluster Architecture

- 3 types of nodes
- 2 types of trackers
- 2 types of daemons

The two nodes can be co-located if the cluster is small.



From Lin & Dyer

MapReduce on Hadoop Cluster

- A simplified view

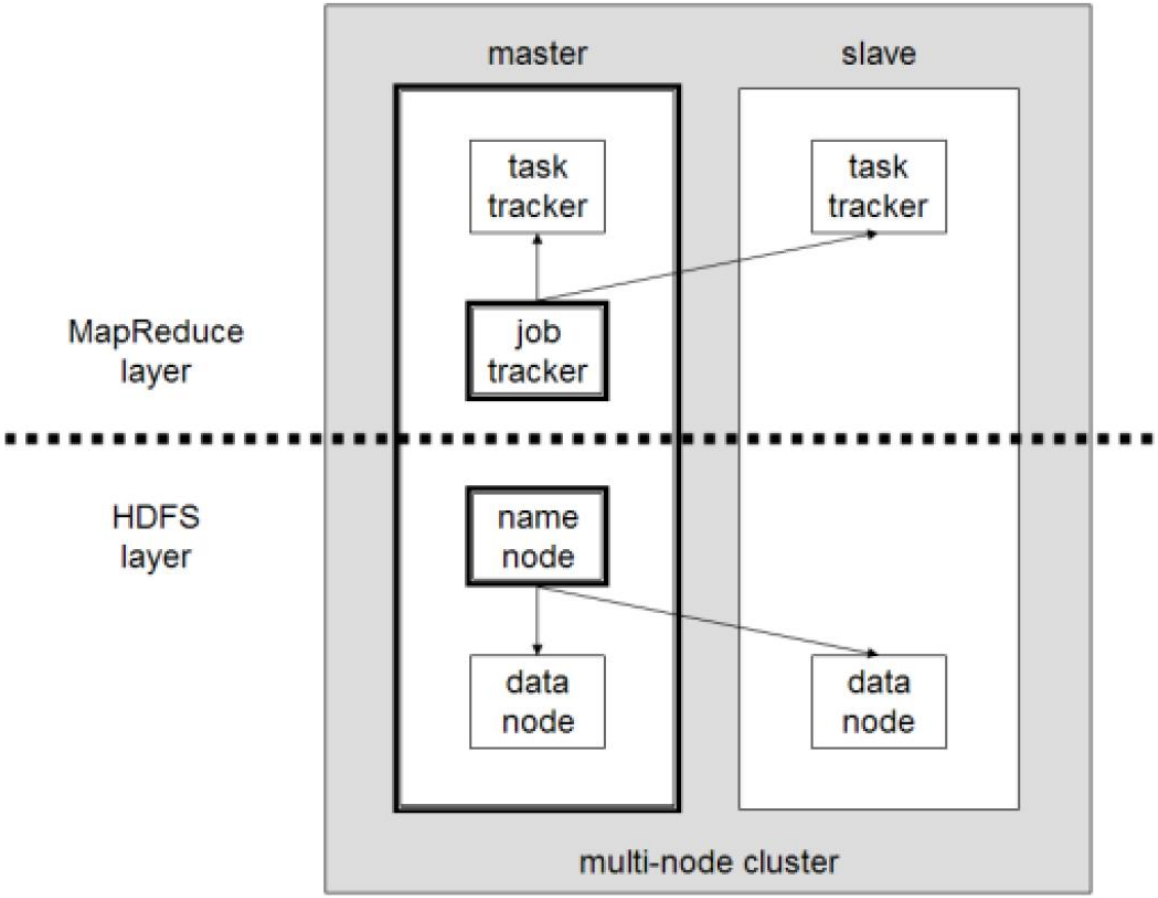


Figure from Henrik Bulskov

Execution Overview

- This is about Google MapReduce
- But the overall procedure is similar in Hadoop

The files can be local or distributed

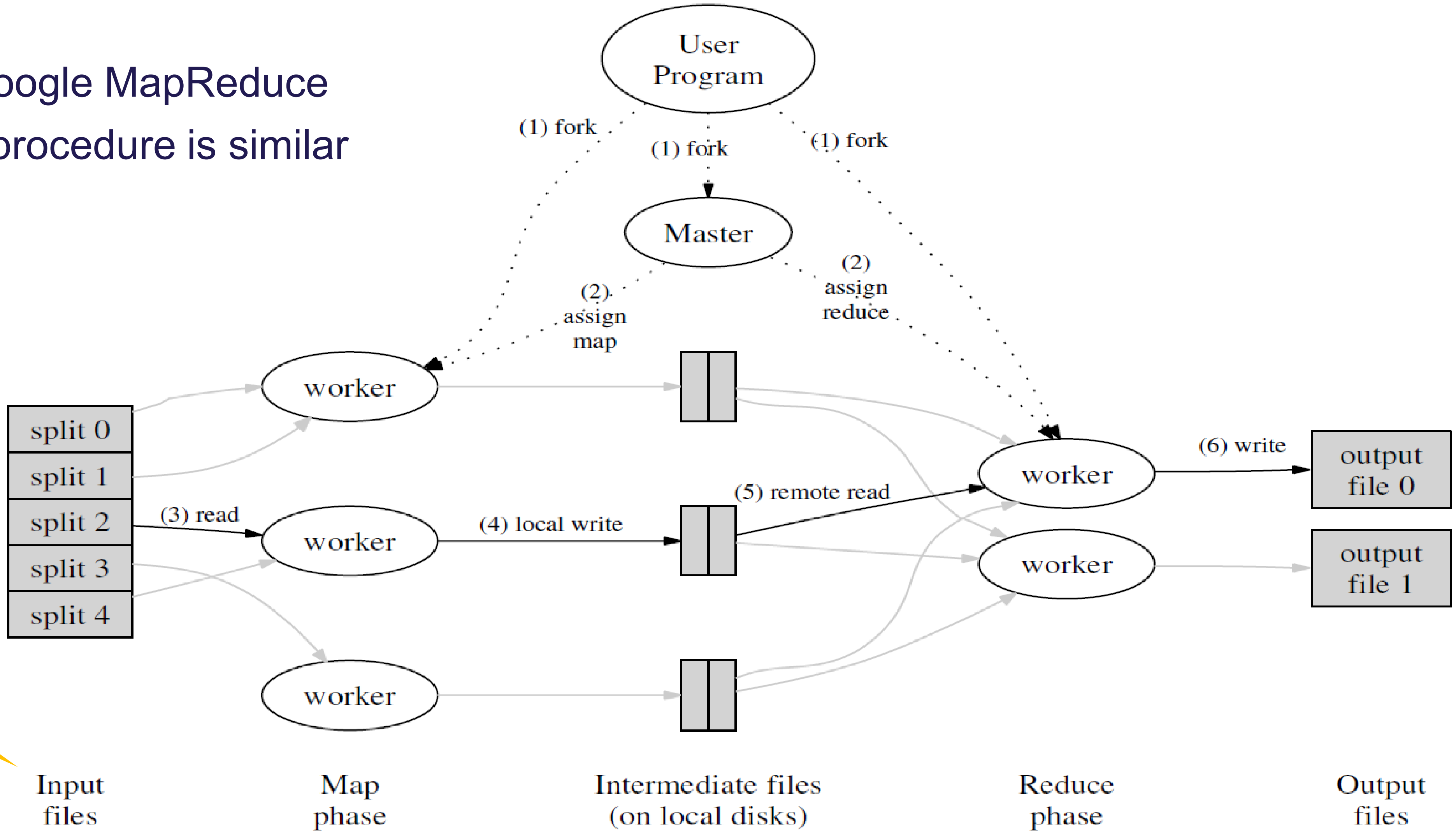


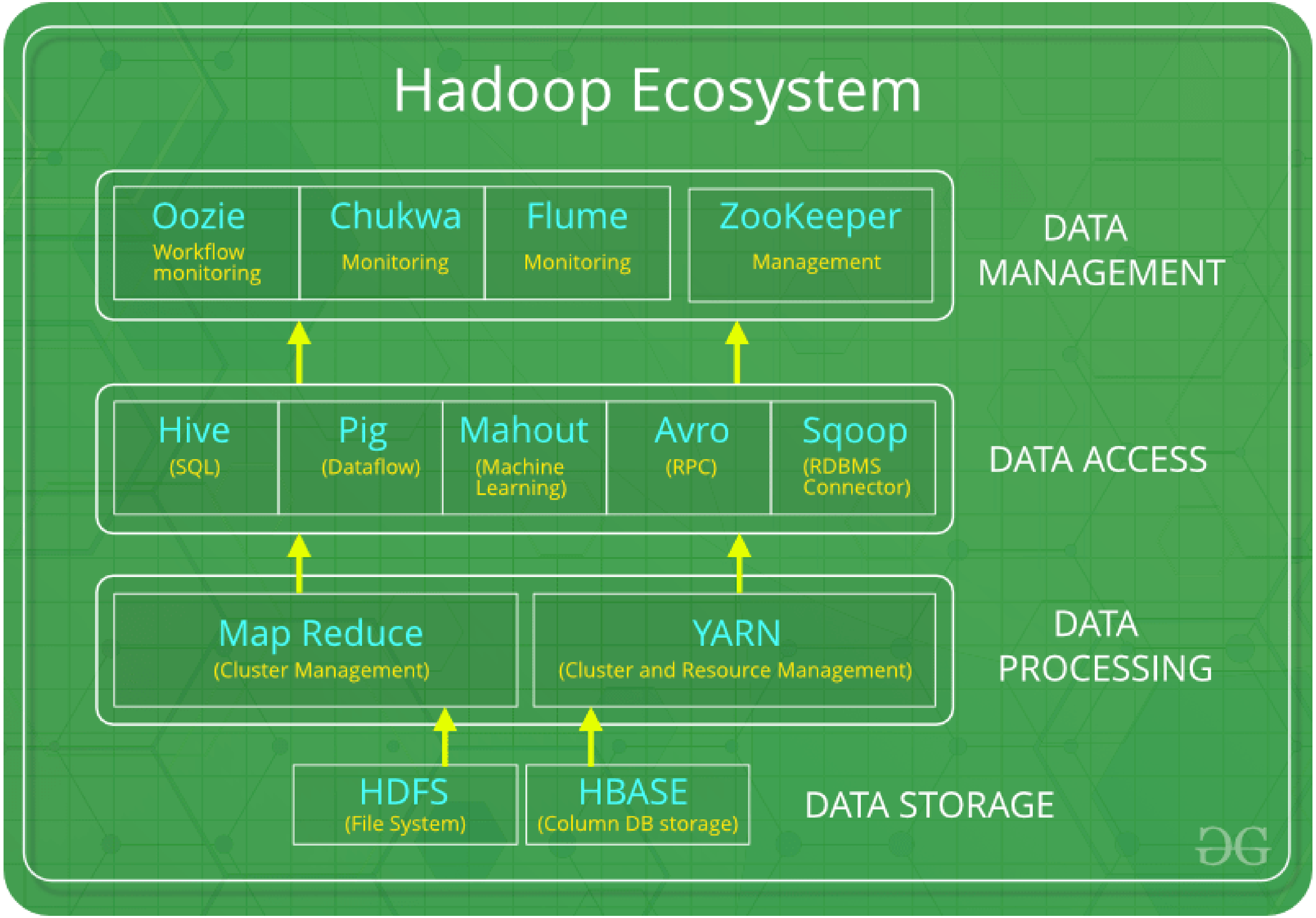
Figure from Jeffrey Dean & Sanjay Ghemawat

Agenda

- Background
- Programming model
- Execution framework
- Software implementation
- MapReduce and databases

Hadoop vs. Databases

- First of all, Hadoop is *not* a (relational) database system.
- Structured vs. Unstructured Data
  - Hadoop can handle both and is a perfect match for unstructured data.
  - Relational databases are designed for structured data only.
- Scalable analytics infrastructure
  - Constant and predictable workloads are good for databases.
  - Increasing data demands can take advantage of Hadoop.
- Cost-effective
  - Low-end commodity machines and open-source software for Hadoop
- Fast data analysis
  - Databases can do better for time-sensitive and general data analysis.
- It's possible/interesting to build hybrid systems with both Hadoop and databases.



https://www.geeksforgeeks.org/hadoop-ecosystem/

MapReduce vs. Parallel Databases

- Schema
  - The parallel RDBMS forces data into a schema (structured)
    - Parsing is only needed at load time and the system enforces constraints
    - Can improve compression and makes it easy to extract a given attribute
    - Helps when optimizing (the declarative) queries
  - MapReduce
    - No schema – very flexible (unstructured)
    - But the programmer must always parse the input (and check that the constraints are not violated). This takes time…
- Indexing
  - The parallel RDBMS has indexes.
  - MapReduce does not have built-in indexes.

MapReduce vs. Parallel Databases, cont.

- Programming model
  - SQL: Declarative programming. You specify *what* you want.
  - MapReduce: Overall declarative; but imperative inside the map/reduce functions.
- Getting started is easier in MapReduce. Maintenance is harder.
- Fault tolerance
  - MapReduce handles it if a node fails and only re-computes the lost part.
    - But MapReduce is typically slower and needs more nodes than an RDBMS. The risk of a failure is then bigger.
  - An RDBMS must restart the entire query execution.

Summary

- ‘MapReduce is the most successful abstraction over large-scale computational resources we have seen to date.’ – Lin and Dyer
- Like all other abstractions in Computer Science, MapReduce is imperfect
  - It makes certain large-scale data processing problems easier, but others still (even more) difficult
- In the data world, there is no ‘one-size-fits-all’ solution. One should choose between MapReduce and (parallel) databases according to the characteristics of her own data and workloads.

References

- Mandatory reading
  - Jimmy Lin and Chris Dyer: Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers, 2010. Chapter 2 *MapReduce Basics*.
- Further readings
  - Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150
  - Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Michael Stonebraker: A comparison of approaches to large-scale data analysis. SIGMOD Conference 2009: 165-178

- Acknowledgements
  - In addition to the authors above, parts of the slides are also adapted from or inspired by:
    - Christian Thomsen (AAU), S. Sudarshan (IIT Bombay), Henrik Bulskov (RUC)

Exercises

Write your code using MapReduce in Python and Jupyter Notebook to resolve the following two tasks

1. To count *each* hashtag that appears in the ID-Hashtag file (in Moodle).
2. To return the hashtag with the highest appearance frequency.

NB: Think how these two tasks are correlated but different, and how you may reuse your code as much as possible.