Eötvös Loránd University (ELTE)
Faculty of Informatics (IK)
Pázmány Péter sétány 1/c
1117 Budapest, Hungary

DATA ANALYSIS SOLUTIONS: MAPREDUCE

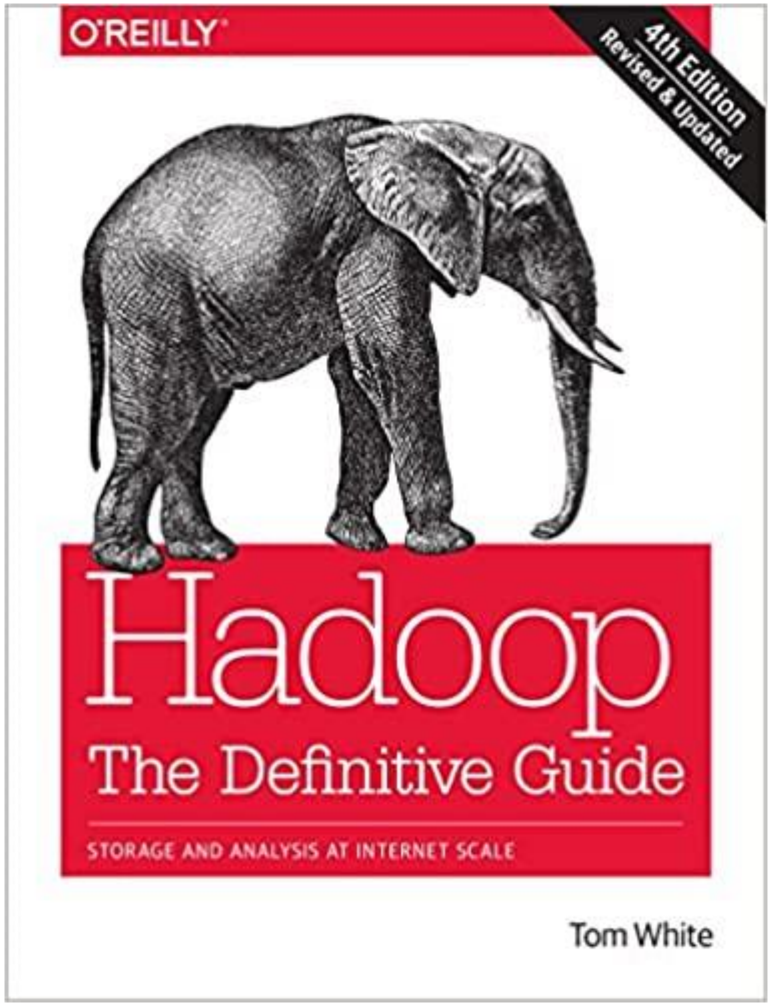
Open-source Technologies for Real-Time Data Analytics

Imre Lendák, PhD, Associate Professor

2020
Budapest, Hungary

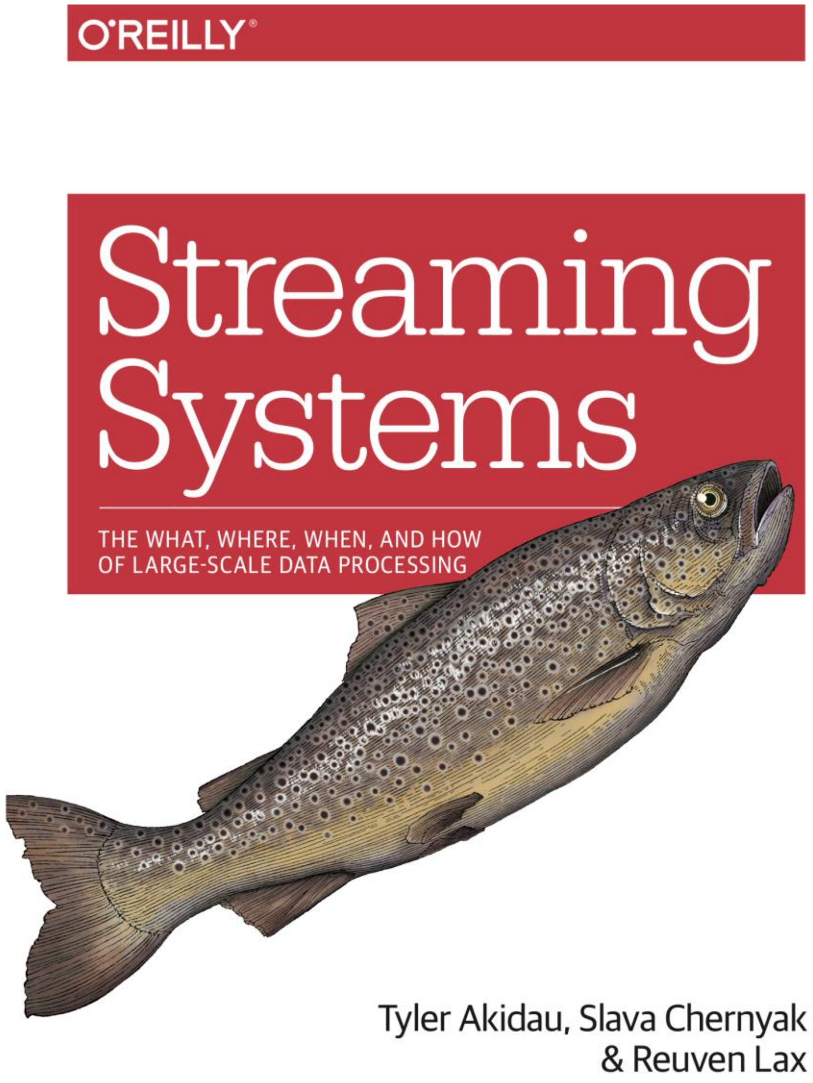
Chosen data analytics topics

- **MapReduce** implements the map & reduce paradigm known from functional programming
  - Discussed in this lecture!
  - Mostly based on the Hadoop book by T.White
- **Spark** is an in-memory batch processing pipeline
  - Discussed later!
- **ElasticSearch** search & analytics engine



Tom White, Hadoop – The definitive guide, O'Reilly, 4th Edition, 2015

Additional resource


O'REILLY
Streaming Systems
THE WHAT, WHERE, WHEN, AND HOW OF LARGE-SCALE DATA PROCESSING
Tyler Akidau, Slava Chernyak & Reuven Lax

- **Q:** Why the Streaming systems book if there is absolutely no streaming in this lecture?
- **A:** Akidau et al work at Google and have local knowledge about

- **Note:** MapReduce was initially a proprietary Google tech

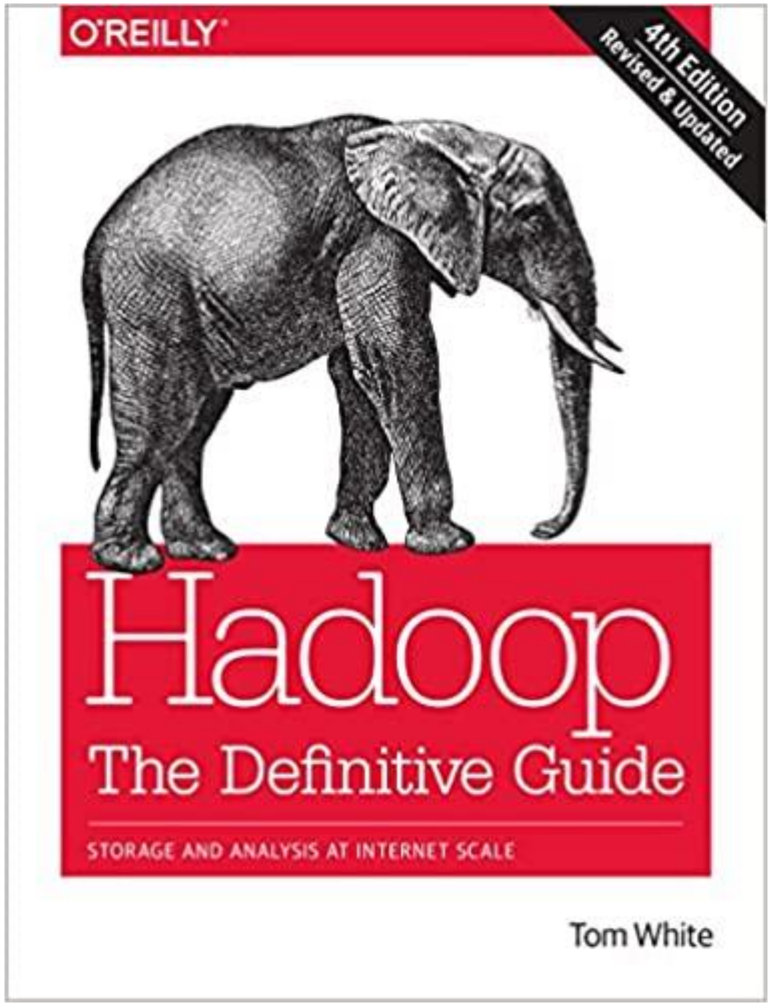
Data analysis timeline

	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019	2020
MapReduce	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦						
Hadoop*		🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Flume						🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪	🟪
Spark							🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
ElasticSearch								🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦
Cloud Dataflow											🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Beam														🟨	🟨	🟨	🟨	🟨

* Analysis elements of the Hadoop ecosystem

MapReduce lecture structure

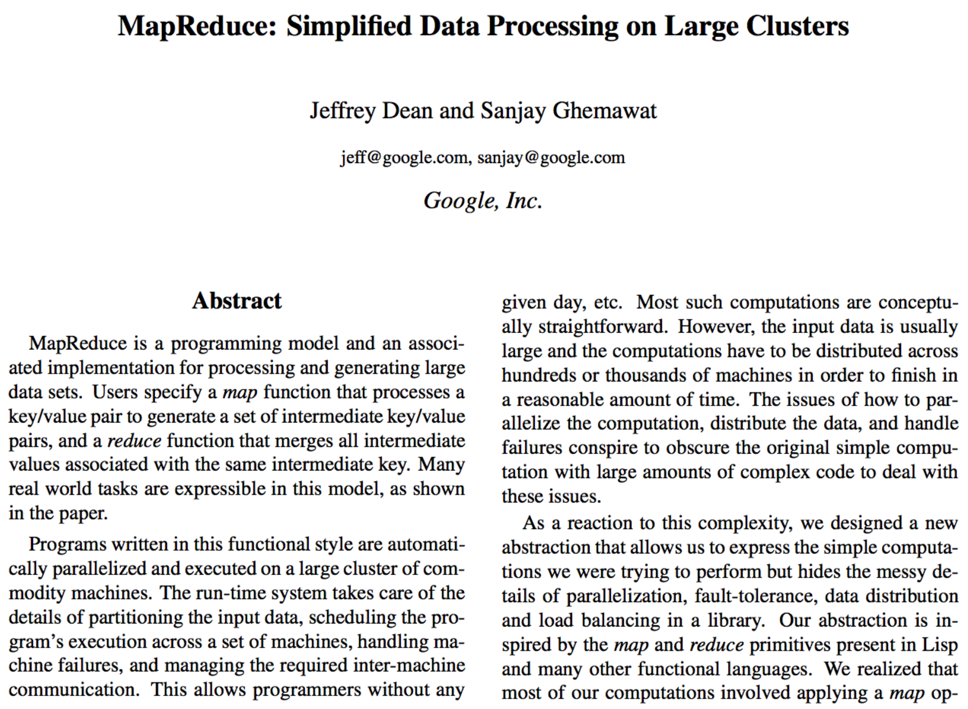
- A bit of history
- Architecture
- Processes → map & reduce + job execution
- Scheduling & sync
- Monitoring & fault tolerance



Tom White, Hadoop – The definitive guide, O'Reilly, 4th Edition, 2015

History

The beginnings…



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* op-

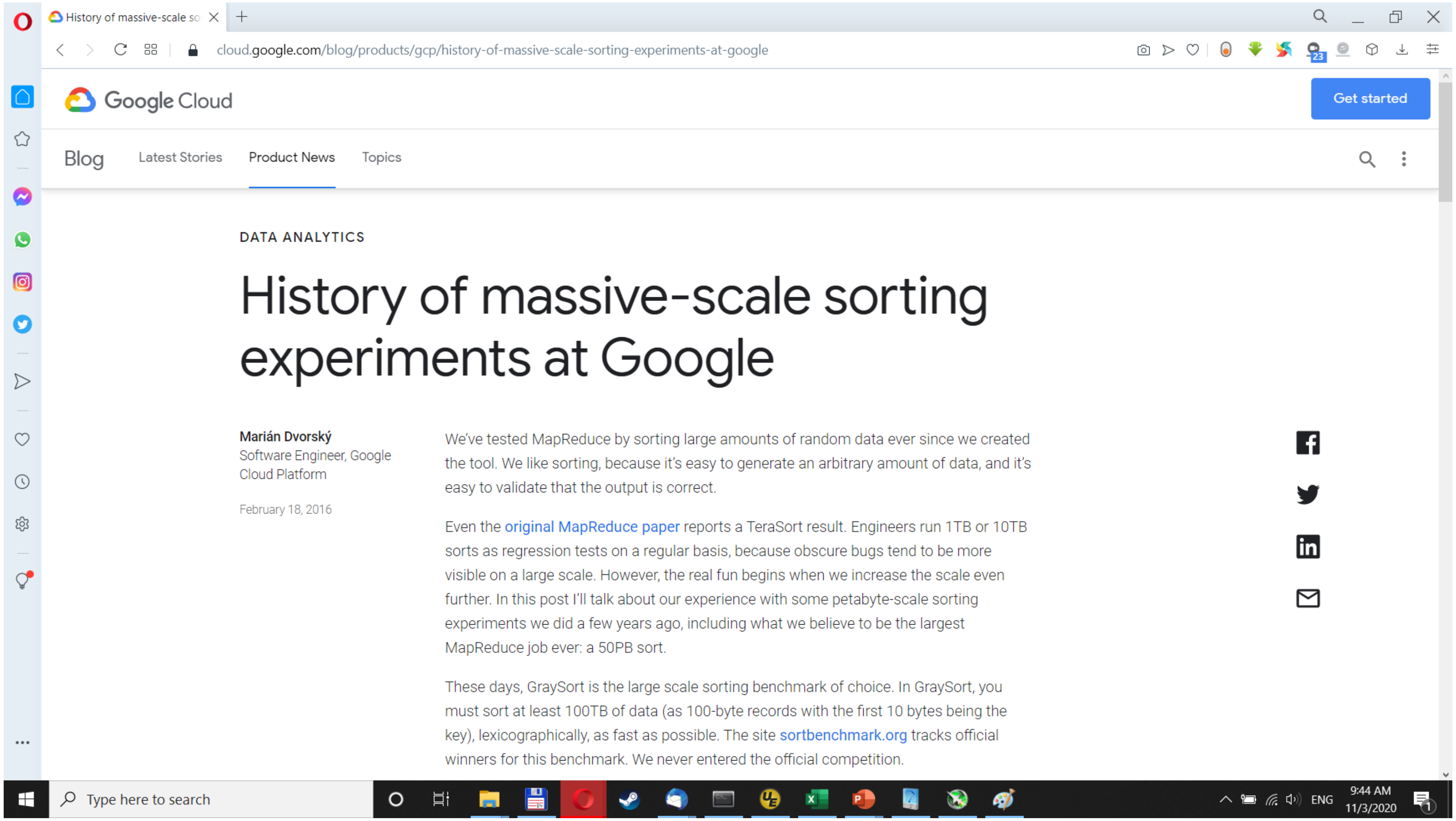
Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters.

- Built on the long history of distributed systems R&D
- Regarded as the first truly 'big data' platform capable to tackle data analyses on the web-scale
- Developed at Google
  - Originally proprietary Google technology
  - Open-sourced later
- Operational in Google as early as 2003
- MapReduce paper published in 2004

Motivation

- **Data processing is hard** → it is always challenging to extract useful information from data
- **Scalability is harder** → it is especially difficult to extract useful information from large-scale (e.g. web scale) data, e.g. crawled content from the web for Google searches
- **Fault-tolerance is even harder** → it is even more challenging to extract useful information from large-scale data in a fault-tolerant & correct way on commodity hardware, i.e. regular, low-cost server hardware
- The developers of MapReduce were motivated to solve the scalability & fault-tolerance challenges → ease large-scale data analysis for generations of data engineers & scientists
  - Developers can focus on data processing instead of solving above distributed system challenges

Additional reading


History of massive-scale so
cloud.google.com/blog/products/gcp/history-of-massive-scale-sorting-experiments-at-google
Google Cloud
Get started
Blog
Latest Stories
Product News
Topics
Type here to search
ENG
9:44 AM
11/3/2020

DATA ANALYTICS

History of massive-scale sorting experiments at Google

Marián Dvorský
Software Engineer, Google Cloud Platform

February 18, 2016

We've tested MapReduce by sorting large amounts of random data ever since we created the tool. We like sorting, because it's easy to generate an arbitrary amount of data, and it's easy to validate that the output is correct.

Even the original MapReduce paper reports a TeraSort result. Engineers run 1TB or 10TB sorts as regression tests on a regular basis, because obscure bugs tend to be more visible on a large scale. However, the real fun begins when we increase the scale even further. In this post I'll talk about our experience with some petabyte-scale sorting experiments we did a few years ago, including what we believe to be the largest MapReduce job ever: a 50PB sort.

These days, GraySort is the large scale sorting benchmark of choice. In GraySort, you must sort at least 100TB of data (as 100-byte records with the first 10 bytes being the key), lexicographically, as fast as possible. The site sortbenchmark.org tracks official winners for this benchmark. We never entered the official competition.

https://cloud.google.com/blog/products/gcp/history-of-massive-scale-sorting-experiments-at-google

ARCHITECTURE

Introduction

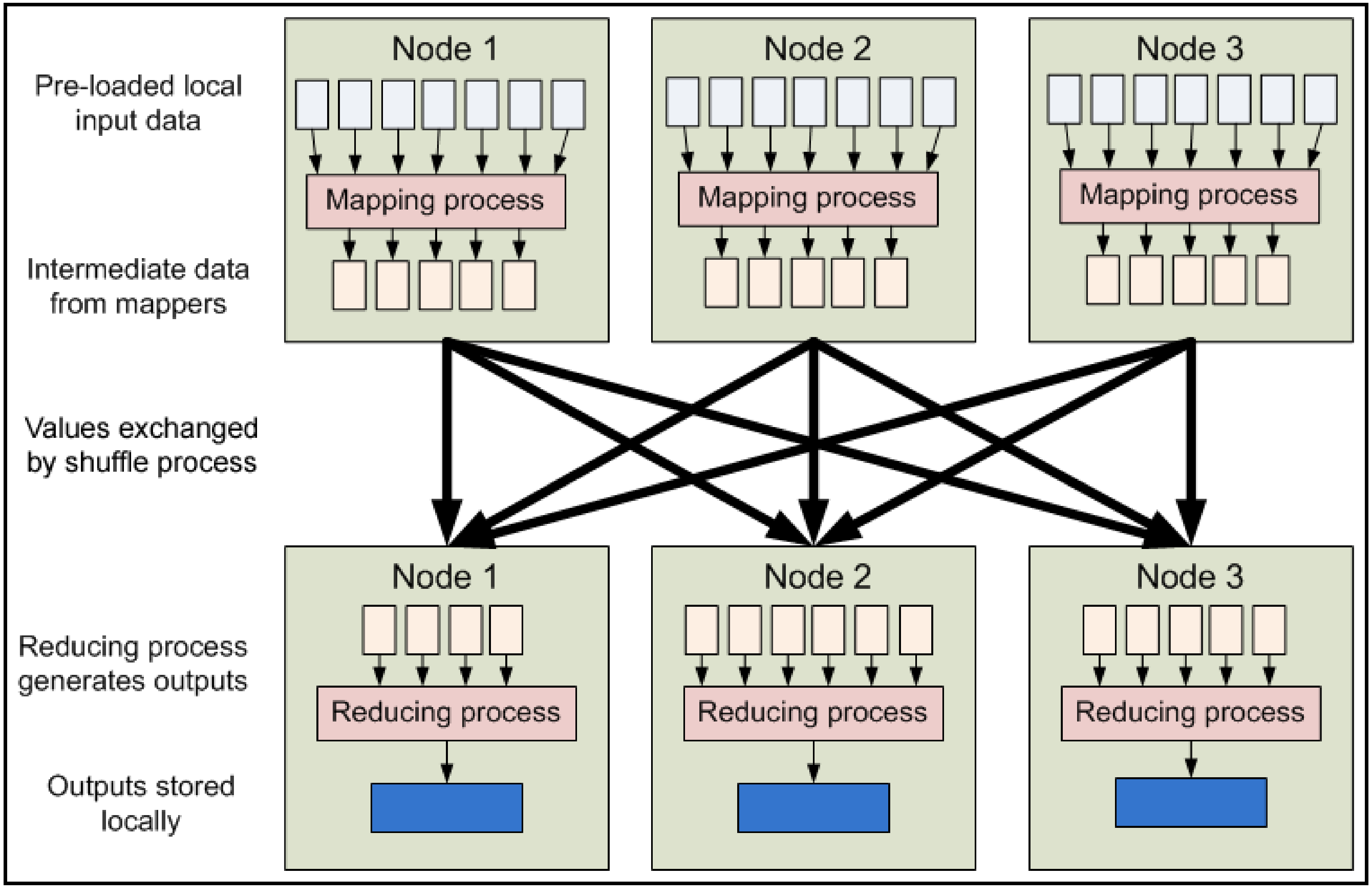
Definitions

- **DEF:** MapReduce is a programming model for efficient distributed computing
  - Presented in 2004 in a paper authored by Google employees
- Each node processes the data that is stored at that node
- Consists of two main phases: (1) Map & (2) Reduce

2-stage data processing

- Map
  - Read (large-scale) key-value data from a distributed filesystem
  - Transform and output different key-value pairs
- Reduce
  - Called once for each unique key
  - The reducer outputs zero or more final key/value pairs

MapReduce dataflow

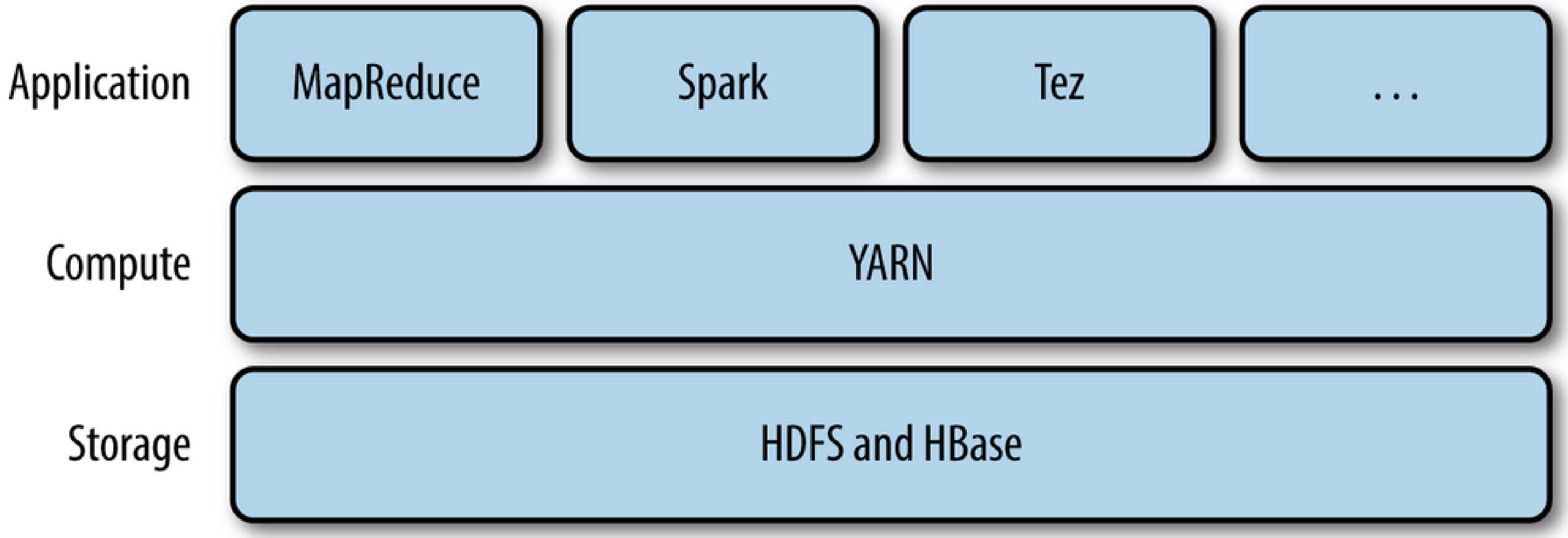

Pre-loaded local input data
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Outputs stored locally
Node 1
Node 2
Node 3
Mapping process
Reducing process

Input & output data types

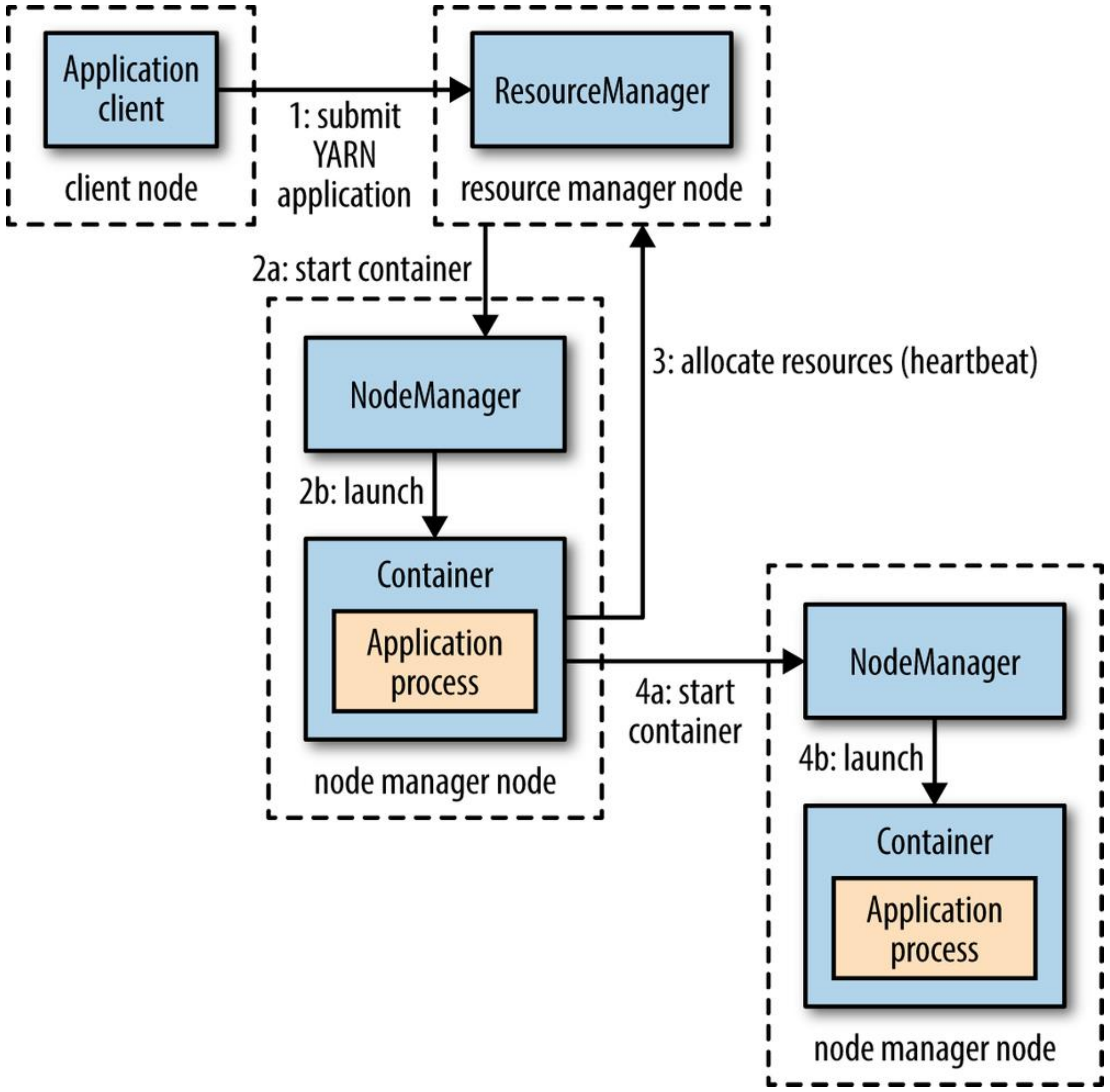
- MapReduce inputs are large-scale data consisting of (K, V) key-value pairs
- A split is a chunk of input data processed by a single map process
  - Entries inside an input split are ‘records’
  - MapReduce is optimized for large-sized inputs
- Common file input/output formats
  - **Text:** lines, key-value pairs, XML
  - **Binary:** sequences of binary key-value pairs
  - **Database input:** read data from an RDBMS using JDBC
  - **Multiple inputs:** common solution for multiple proprietary versions of the same data source

YARN

- **DEF:** Yet Another Resource Negotiator (YARN) is Hadoop’s cluster resource management system
- Introduced in Hadoop 2
- Supports other distributed computing paradigms



YARN application execution



Processes #1: Map & Reduce

Applications, jobs and tasks

Job-related definitions

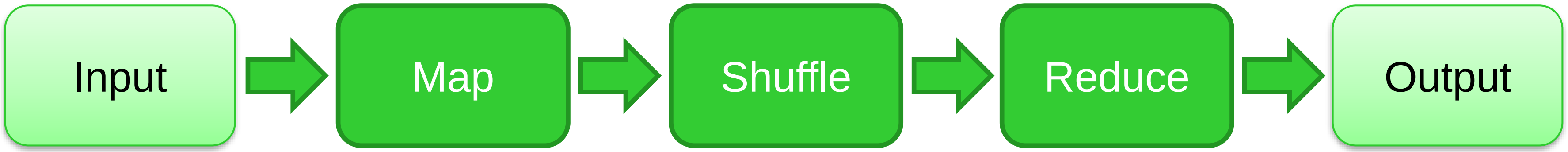
- **Applications** submit **jobs**
- The job's input dataset is organized into (input) **splits**
- Each split is assigned a map **task**
  - Tasks are executed 'near' the (input data) split
  - Tasks execute in parallel

Job lifecycle

1. Job submission
   - Compute input splits
   - Copy resources needed to run
2. Job initialization
3. Task assignment
4. Task execution
5. Progress monitoring
6. Job completion

Data processing phases

- I/O: Key-value (K1, V1) input is loaded and split (usually HDFS)
- Process: The **map** process transforms (K1, V1) and generates intermediate results on the local disk
- Process: The **shuffle** process sorts, copies and merges the intermediate outputs on the reduce compute nodes
- Process: The **reduce** process transforms the outputs of the shuffle phase
- I/O: The reduce phase writes the outputs to storage (usually HDFS)



Map

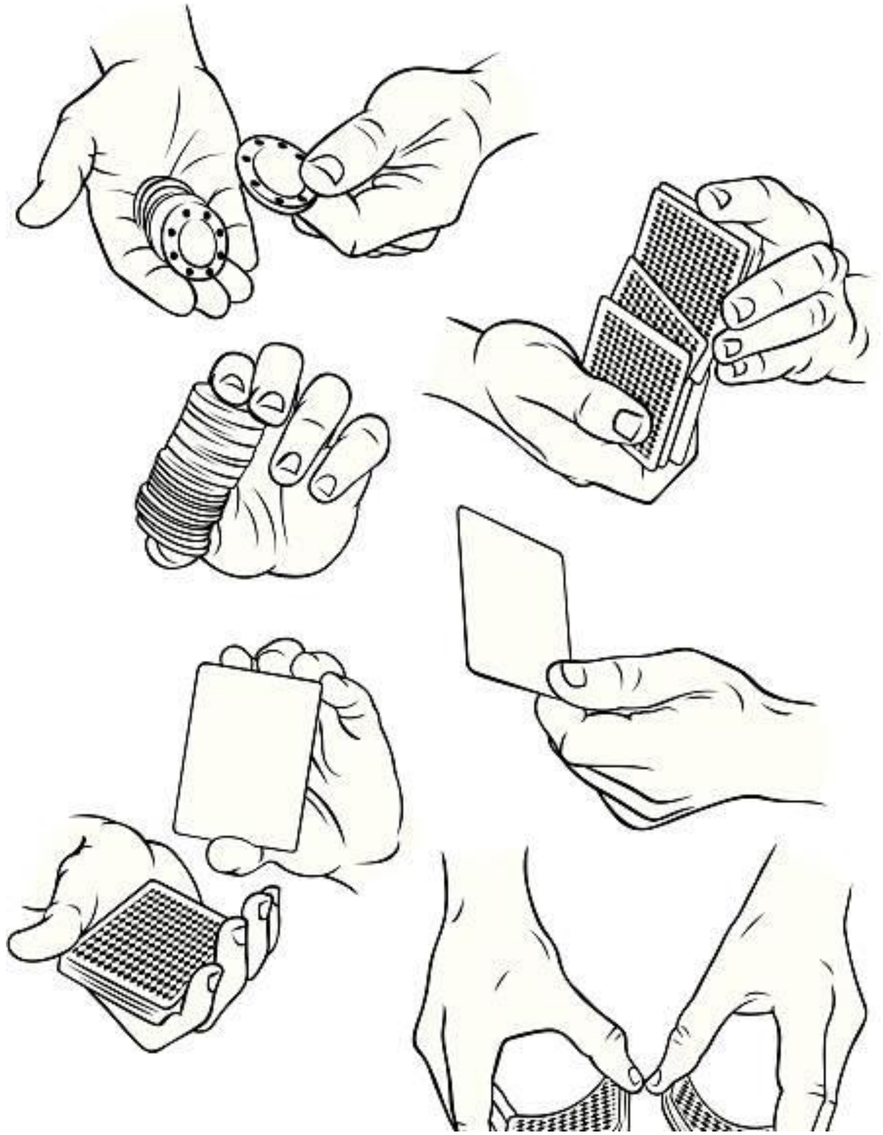
- Map transformation:

$$map(k, v) \rightarrow list(k_1, v_1)$$

- **Implemented by:** the user submitting the job(s)
- **Input:** Map is performed on a(n input) split of the dataset
  - Large datasets are usually stored in a distributed filesystem, e.g. HDFS
- **Execution:** run on multiple compute nodes with data locality optimization
- **Output:** Intermediate values are persisted only in local storage only → does not consume valuable cluster (bandwidth) resources
  - Spill to disk → when local (output) buffer is full, its contents are persisted to (local) disk
  - Map outputs can be compressed (ZIP) for optimized spill
  - Map outputs are reported via the heartbeat to the Application Master

Shuffle

- **Goal:** collect intermediate map outputs, sort by key and prepare for the reduce tasks
- **Implemented by:** the computing cluster itself, i.e. automatically executed and the client submitting the job is not burdened with the implementation of this phase
- **Input:** intermediate outputs generated by map tasks and stored locally
- **Output:** sorted key-value pairs collected on the compute nodes running the reduce tasks



Reduce

- Reduce transformation:

$$reduce(k_1, list(v_1)) \rightarrow (k_2, v_2)$$

- **Implemented by:** the user submitting the job(s)
- **Input:** collected by the shuffle phase, sorted and prepared for the reduce phase
  - Reduce executes multiple threads which fetch map outputs → reduce queries the Application Master for map process status
  - Inputs copied to memory if they fit, otherwise to disk
- **Execution:** one or more reduce tasks are executed to produce the output key-value pairs
  - Usually one reduce process per key k_1 in large-scale data
- **Output:** Reduce outputs key-value pairs (k_2, v_2)
  - Reduce outputs are persisted in persistent storage (HDFS)
  - Input & output keys (i.e. k_1 and k_2) might not be the same

Map & reduce in action

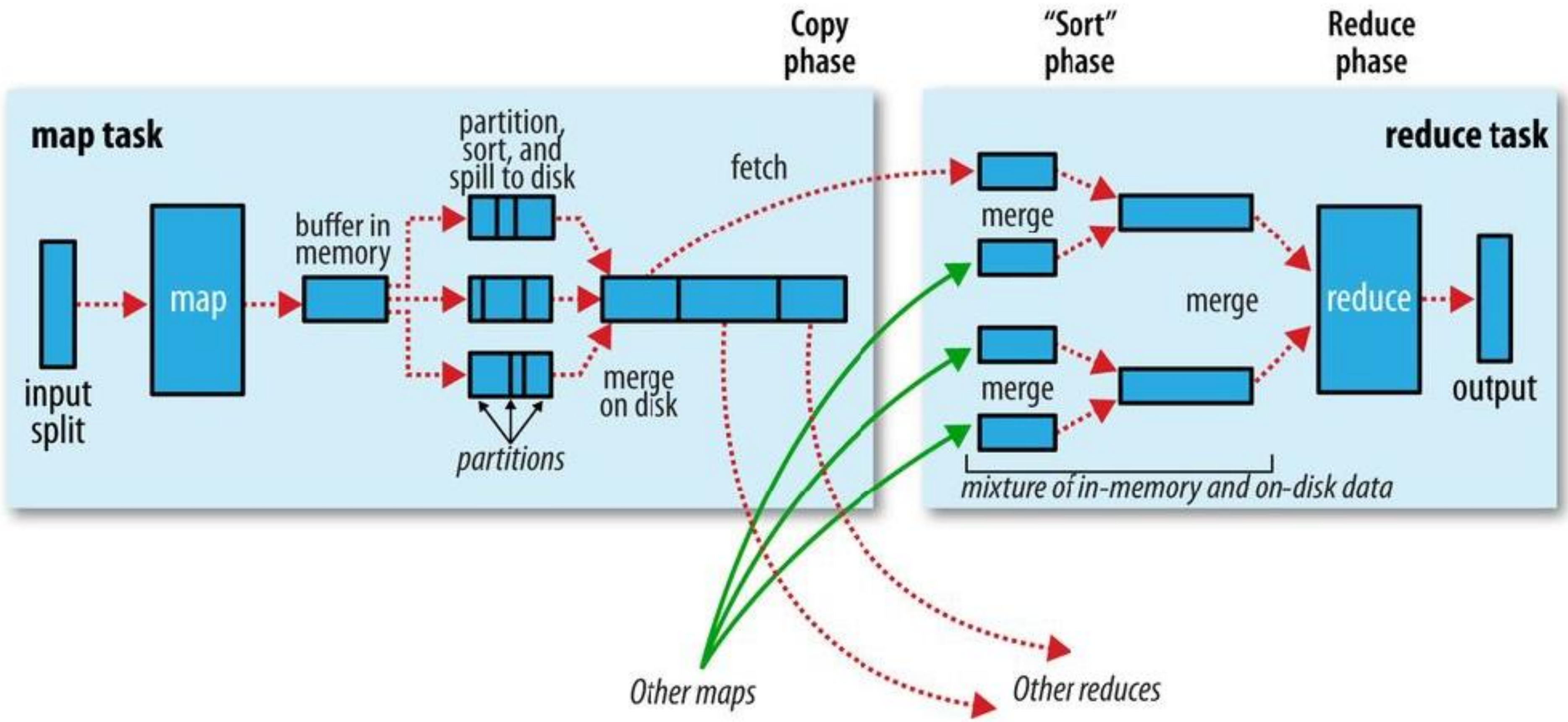
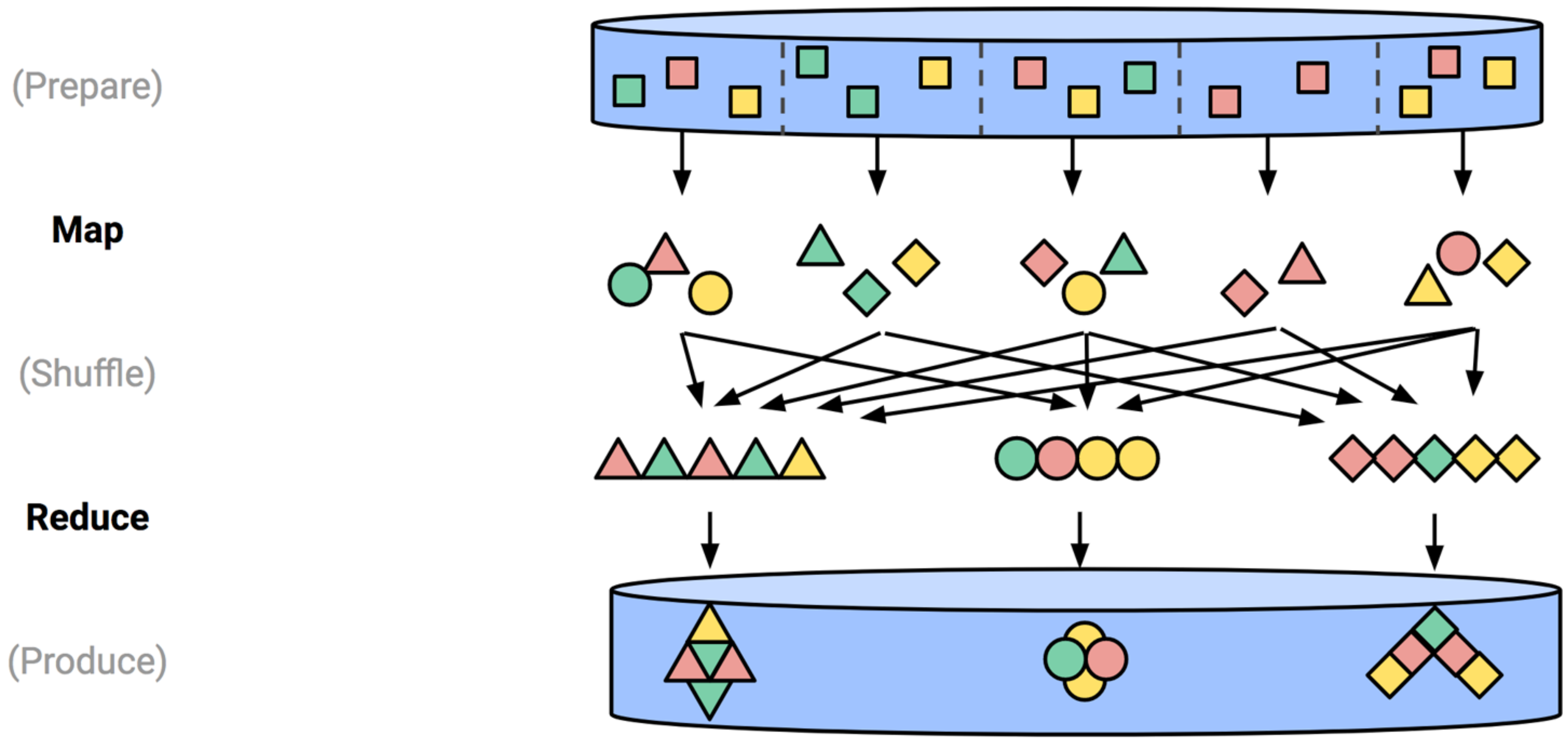


Figure 7-4. Shuffle and sort in MapReduce

Tom White, Hadoop – The definitive guide, O'Reilly, 4th Edition, 2015

Data-centered view

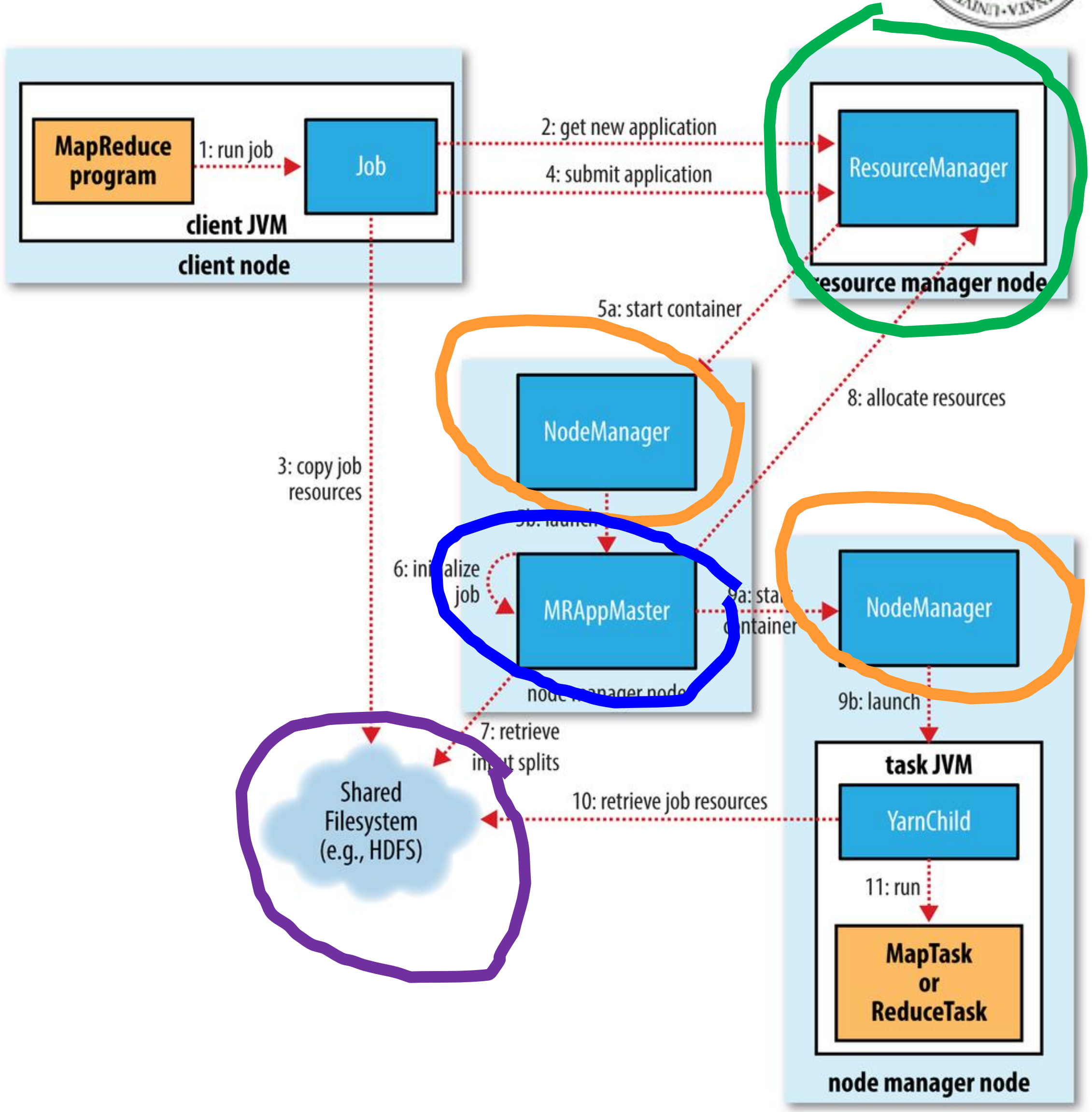


Akidau T., Chernyak S., Lax R. Streaming Systems: The What, Where, When, and how of Large-scale Data Processing, O'Reilly Media, 2018.

Processes #2: Job execution

Processes in job execution

1. Client submits a job
2. Resource manager
3. Node manager
4. Application master
5. Distributed filesystem, i.e. HDFS discussed earlier



Resource Manager

- The (YARN) **Resource Manager (RM)** coordinates the allocation of compute resources in the cluster
  - The RM is a long-running process → its lifecycle can be as long as complete cluster uptime
  - The RM has two main components: Scheduler and Application Manager
- The **Scheduler** allocates resources to running jobs
  - Does not monitor or track the progress of jobs
  - Does not offer guarantees in case of failures
- The **Application Manager** accepts job submissions
  - Assigns the 1st container to run the app-specific Application Master
  - Restarts the Application Master on failure

Node Manager

https://www.cleanpng.com/free/middle-management.html

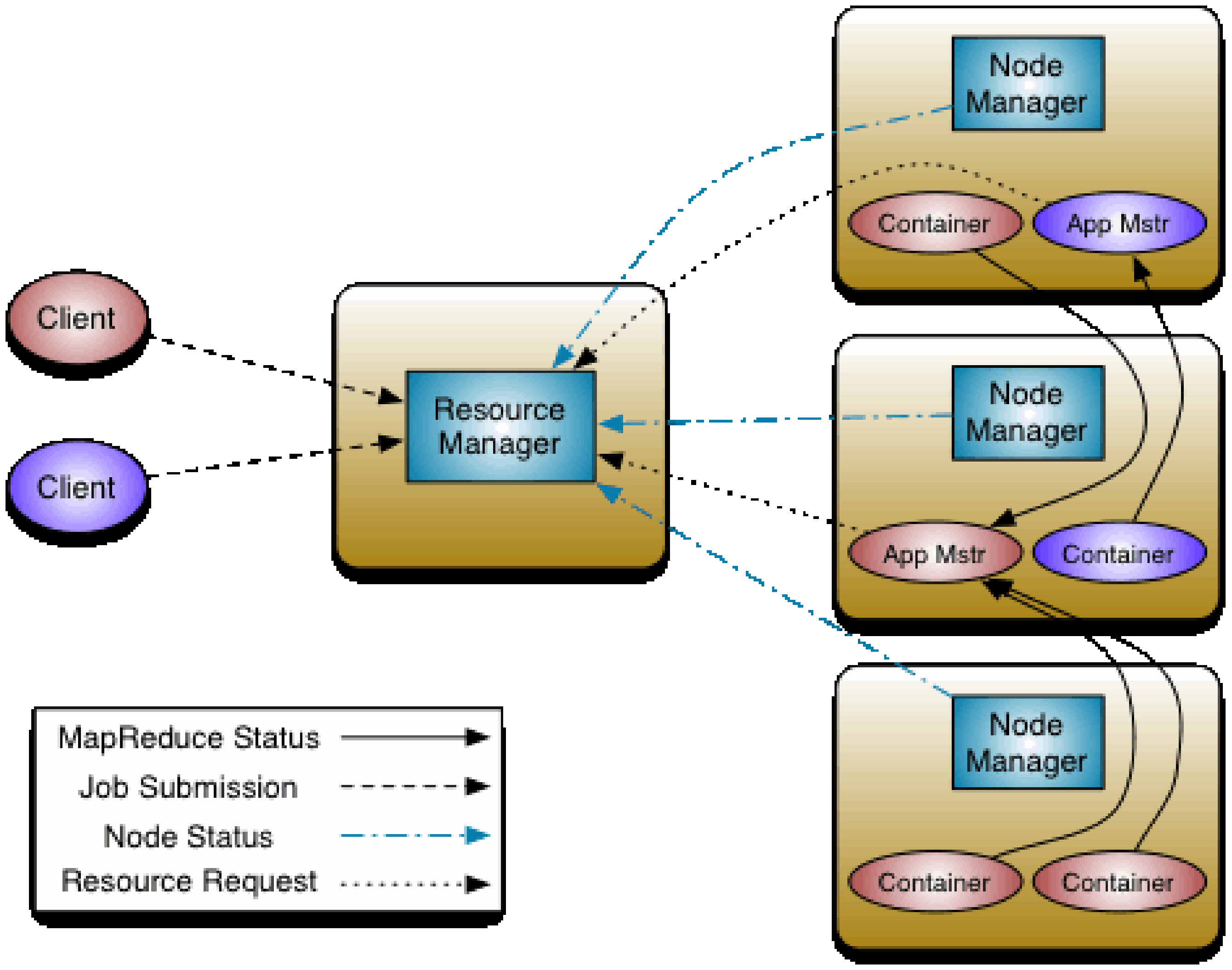
- Node Manager (NM) processes are executed on compute nodes, e.g. physical server computers in the computing cluster
- NMs report node status to the Resource Manager and/or Scheduler
  - Node resource usage (cpu, memory, disk, network)
  - Active containers/tasks

Application master

- A single Application Master (AM) is run **per application**
- Requests containers from the Scheduler
- Tracks the status of containers received and running the tasks of the current job, i.e. monitors container (task) progress

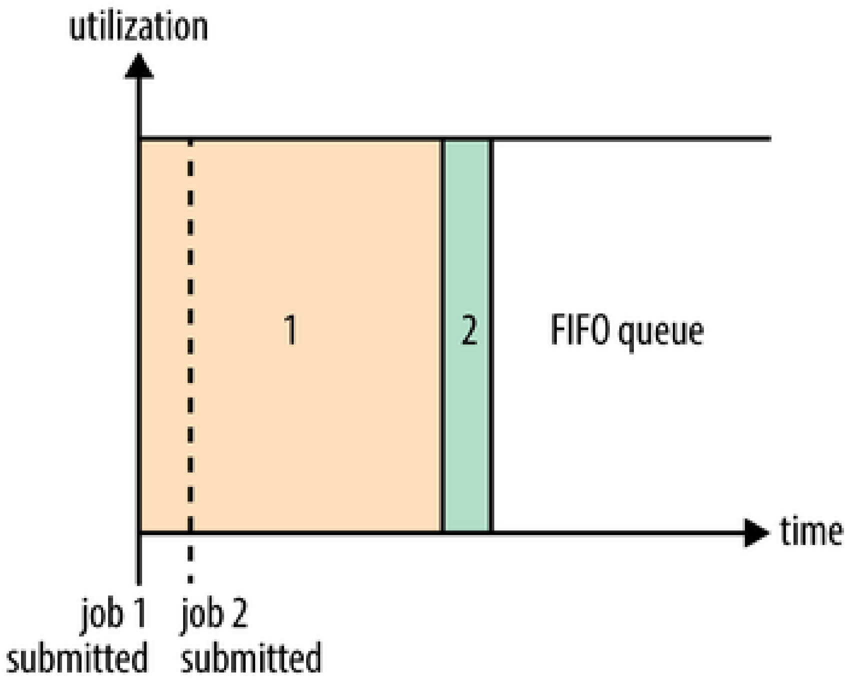
Note: Master Yoda is not the best fit, as he is the Grand Master in Star Wars, while the AM is only short-lived process

Processes in context



SYNCHRONIZATION & SCHEDULING

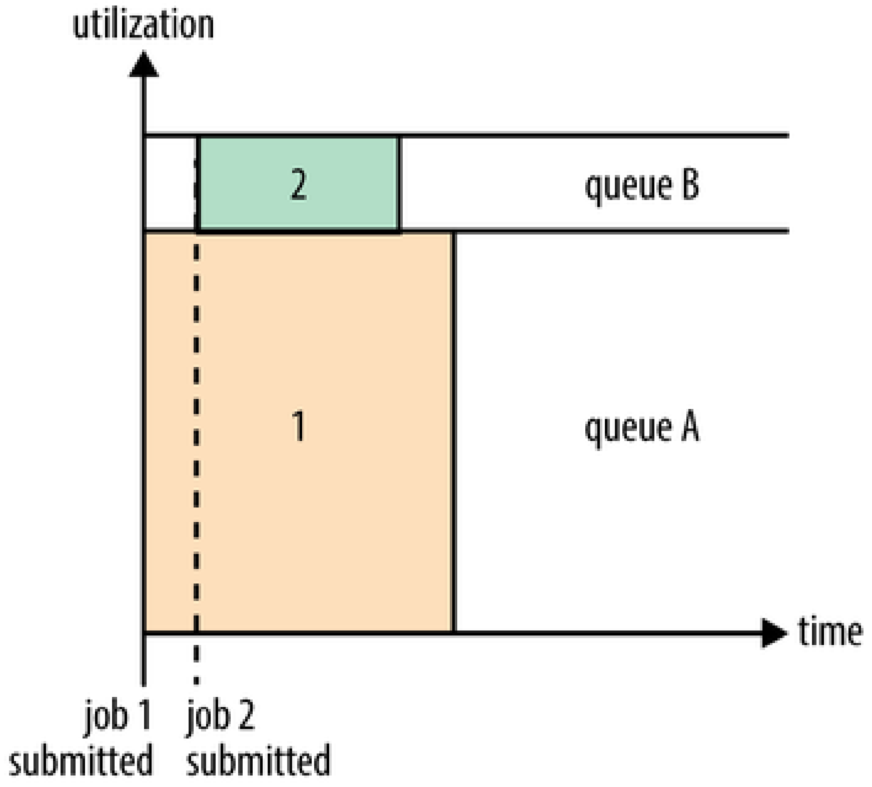
YARN scheduler: FIFO



https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/ch04.html

- First in, first out (FIFO)
- Simple (to understand and implement)
- Long-lasting jobs use 100% resources
- Other tasks wait until large, long-lasting jobs finish
- Not suitable for shared clusters with large numbers of jobs

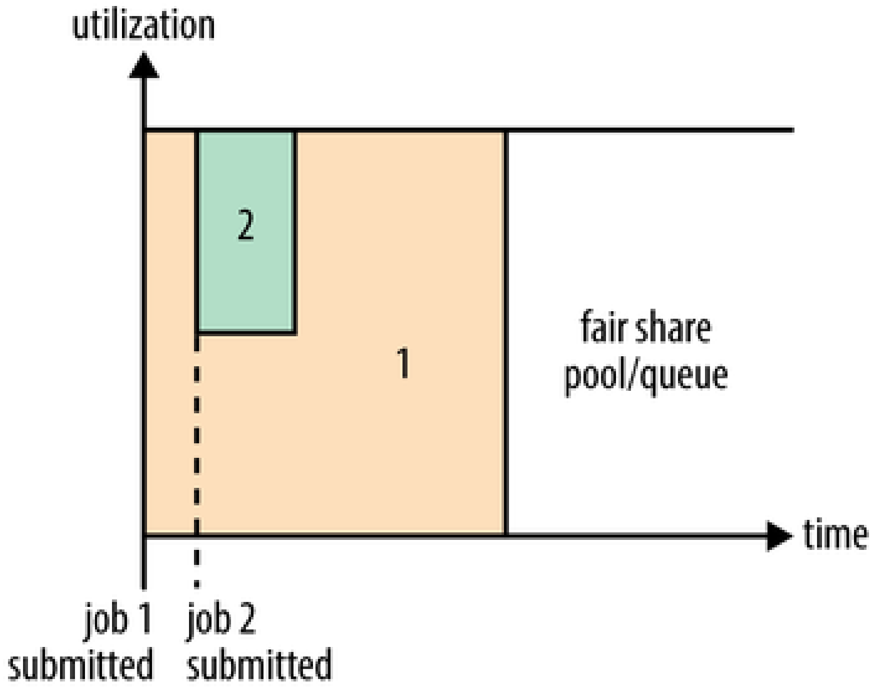
YARN scheduler: Capacity



- The Capacity Scheduler allows cluster admin to configure additional queues
- The large job (1) finishes later compared to the FIFO Scheduler
- When job 2 finishes, the resources allocated to queue B are not freed → the cluster is not 100% utilized

https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/ch04.html

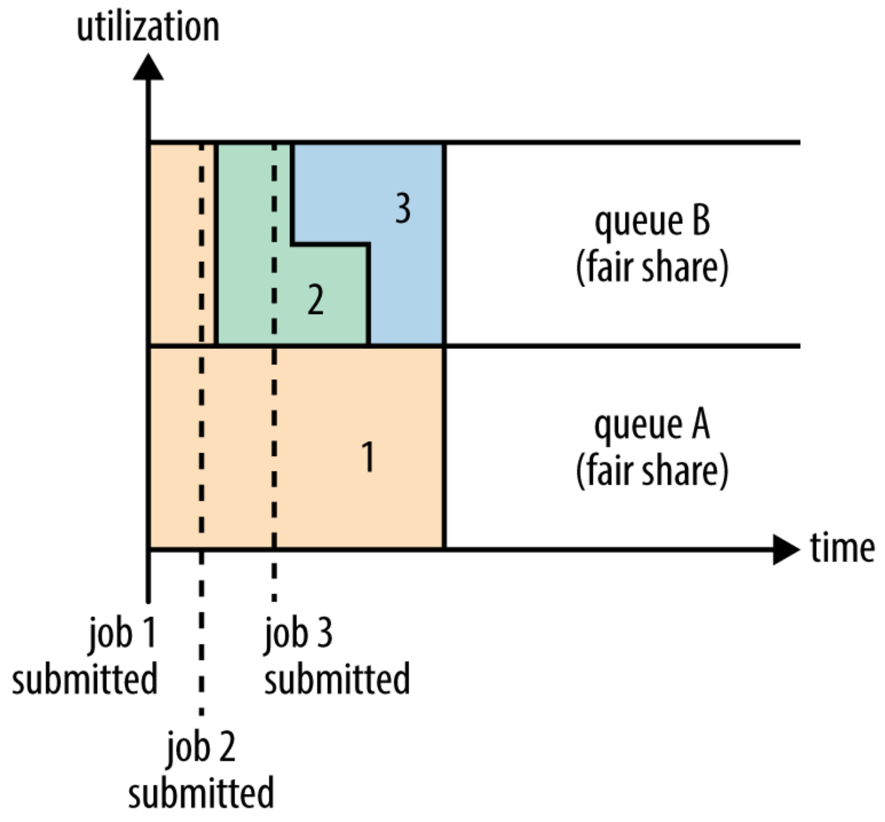
YARN scheduler: Fair



https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/ch04.html

- The Fair Scheduler does not rely on resource reservations
- Job 2 starts as soon as containers are freed by job 1
- Job 2 is allocated 50% of the cluster resources
- When job 2 finishes, the long-lasting job resumes to use 100% of cluster resources

YARN Fair Scheduler: 2 users



https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/ch04.html

- **Users:** A, B
- **Queues:** A & B, each with up to 50% of cluster resources
- **Phase #1** – A single job by use A: At the start the large job 1 of user A uses 100% of cluster resources
- **Phase #2** – Additional jobs by user B: When the first job of user B is submitted, queue B is allocated 50% of cluster resources

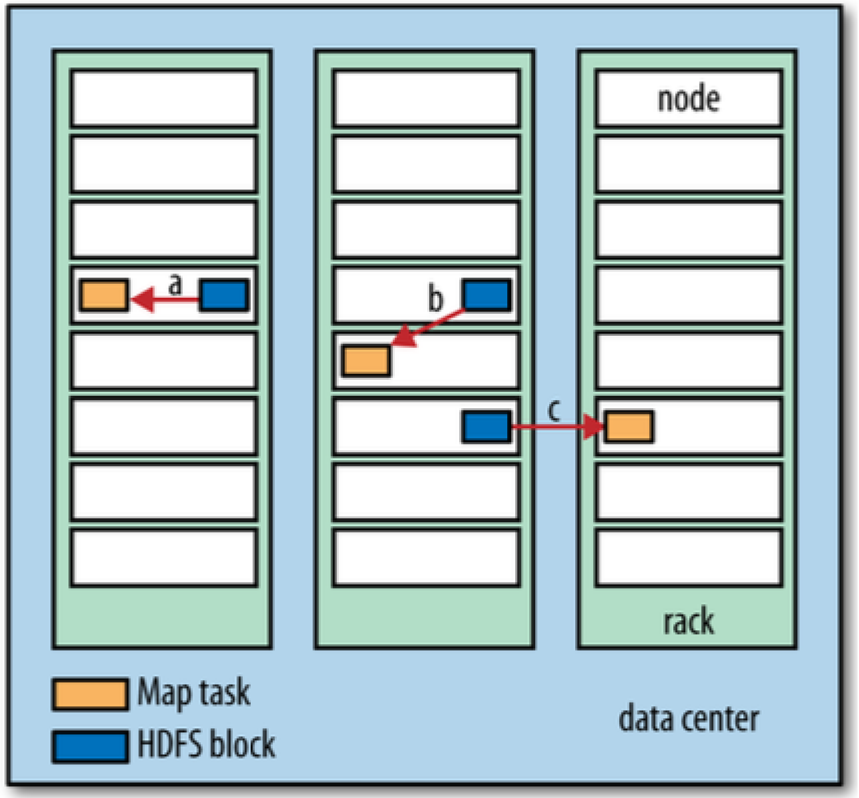
Preemption

1. Cluster utilization: 100%
2. A job J1 is submitted to an empty queue
3. J1 will wait for job-to-start time T1, i.e. until some containers complete their tasks

- **DEF:** Preemption is optional and allows the scheduler to kill tasks for queues which are running with more than their fair shares of cluster resources
- **Pro:** preemption allows job-to-start times to be more predictable
- **Contra:** preemption negatively impacts overall cluster efficiency, as the killed tasks are executed again

Data locality optimization

- **DEF:** **Data locality** optimization aims to run (map) tasks on the same compute node where the input data (split) is located
- Levels of data locality in map task execution:
  - Data-local
  - Rack-local
  - Off-rack
- Data locality results in lower cluster bandwidth used



Delay scheduling

- Delay scheduling is done if the compute node requested and used to store the (input data) split is 100% busy with running other tasks
- Delay scheduling allows the resource manager to wait a short period of time before deciding to schedule the task on a different (compute) node in the same, or another rack

- **Pro:** Delay scheduling was shown to be superior to the resource manager immediately deciding to move the task to a different node and/or rack
  - Less cluster bandwidth is used for copying the split
- **Contra:** there is a slight delay in task execution

Dominant resource fairness

- Different MapReduce jobs/tasks can have different CPU and memory requirements
  - Job A might have high CPU and low memory load, while job B might have both high CPU and memory
- Dominant Resource Fairness (DRF) allows the resource scheduler to allocate resources based on dominant resource use (measured as a % of cluster capacity)
- DRF example:
  - Cluster capacity: 100 CPUs, 10 TB memory
  - Task A: 4 CPUs, 500 GB memory
  - Task B: 8 CPUs, 300 GB memory
  - Dominant resource: ?   4+8=12 → 12%, 500+300=800 → 8%
  - Amount of CPUs/memory allocated to A and B: ?

Task A gets 2x more CPUs than B

Monitoring & control

Progress & status updates

- MapReduce jobs can take from tens of seconds to hours to complete
- Each job and its tasks have **status** (running, completed, failed)
- Jobs also have **counters** of two varieties:
  - Built in counters, e.g. number of records written
  - User-defined counters
- User code can set the **status message or description**

- Map and reduce tasks measure **task progress** as
  - Map task progress is measured as the amount (%) of input processed
  - Reduce task progress is more challenging to measure → estimated proportion (%) of reduce input processed
- Tasks report task progress to the Application Master

Progress in MapReduce

- **Read** an input record (map/reduce)
- **Write** an output record (map/reduce)
- **Set status description** (from inside the user’s map/reduce code)
- **Increment a counter**

https://www.vhv.rs/viewpic/hJoToxb_funny-work-in-progress-clipart-png-download-work/

Built-in job counters (selection)

Counter	Description
Launched map tasks	Number of map tasks launched, inclusive of speculative execution tasks
Launched reduce tasks	Number of reduce tasks launched
Failed map tasks	Number of map tasks failed
Failed reduce tasks	Number of killed reduce tasks
Killed map tasks	Number of killed map tasks
Killed reduce tasks	Number of killed reduce tasks
Data-local map tasks	Number of tasks ran on the same node with split*
Rack-local map tasks	Number of tasks ran in the same rack with split*
Other-local map tasks	Number of tasks ran in other racks*
Total time in map tasks	The total time taken to run map tasks*

* Note: the same counters exist for reduce tasks as well !

Built-in task counters (selection)

Counter	Description
Map input records	Number of records consumed by map tasks*
Map output records	Number of output records produced*
Split raw bytes	Number of input split bytes read by map tasks
Spilled records	Number of records spilled to disk
Reduce input groups	Number of distinct key groups consumed by reduce tasks
CPU milliseconds	Cumulative CPU time for a task

* Note: the same counters exist for reduce tasks as well !

Speculative execution

INVESTMENTS
SCHWADRON
"YOUR BEST HEDGE AGAINST A DOWN MARKET IS HAVING LOTS OF MONEY."
CARTOONSTOCK
Search ID: hsc5781

https://www.cartoonstock.com/directory/h/hedged.asp

- Speculative execution (SE) → create a duplicate task
- SE is done when the RM detects slow-running tasks
- SE re-runs the same task on a different node → duplicate
  - The first to finish task's outputs are consumed
  - The other (late-comer) task is killed

Fault tolerance

Fault intro

https://www.dreamstime.com/illustration/fault.html

- Errors and faults can occur at multiple locations
- We discuss the following fault locations
  - Tasks
  - Node Manager
  - Application Master
  - Resource Manager
- **Note:** we are not discussing HDFS failures

Task failure

- Both map and reduce tasks might fail, usually due to a software bug
- **Caught exceptions** are reported to the Application Master by the JVM → AM marks task failed and frees container resources
- **Hanging tasks** are noticed by the AM based on delayed heartbeats → AM kills hanging tasks after a default timeout of 10 min
- Failed tasks are **re-scheduled** on different nodes up to 4 times → # failures > 4 → job fails
- **Partial job success** is allowed if only a configured amount of tasks fails

https://midliferesolution.com/tag/failure/

Application Master failure

https://www.123rf.com/photo_49076207_the-3d-guy-and-a-sign-of-error.html

- The Application Master (AM) sends **periodic heartbeats** to the Resource Manager (RM)
- The **client polls** the AM for status updates
- When RM detects AM failure, it creates a **new AM process** in a different container
  - The running AM persists job history & task progress
  - New AM re-constructs job history → not all tasks re-run
- YARN **retries** failed Application Masters
  - Default retries: 2

Node manager failure

- Node Manager (NM) instances send **heartbeats** to the Resource Manager (RM) → detect failure/slowdown
- Tasks belonging to incomplete jobs on the failed node are re-run on different nodes
  - Intermediate outputs might not be accessible
- An Application Manager (AM) might **blacklist nodes** with high failure counts

https://www.dreamstime.com/stock-illustration-error-d-people-upset-metaphor-image43976249

Resource Manager failure

https://www.123rf.com/photo_76996200_3d-man-presses-the-button-with-error-symbol-.html

- The active Resource Manager (RM) stores running application information in stable storage
  - Note: Node Manager state is not stored → it is re-constructed based on NM heartbeats
- When the RM fails, no jobs and tasks can run
- A pair of **active-standby** RMs are run
- When a new RM is run, it reads **application state** from stable storage and restarts the Application Manager processes

MapReduce Summary

- A bit of history
- Architecture
- Processes
  - Map & reduce
  - Job execution
- Synchronization & scheduling
- Replication
- Monitoring
- Fault-tolerance

Thank you for your attention!