

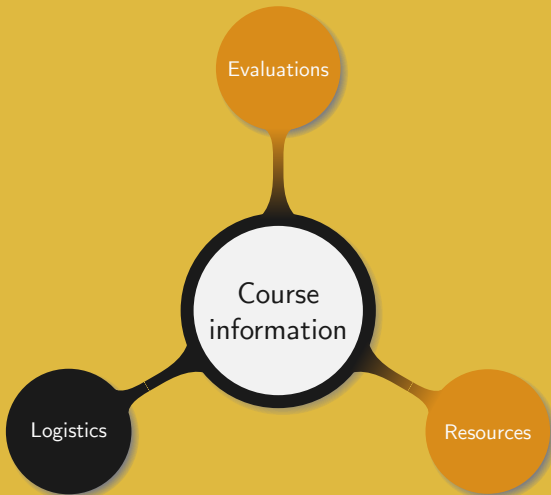
# Methods and tools for big data

Manuel – Summer 2022



## 0. Course information

---



### Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistants:
  - Yangyang (wangyangyang@sjtu.edu.cn)
  - TBD (TBD)

### Important rules:

- When contacting a TA for an important matter, CC the instructor
- Prepend [VE472] to the subject, e.g. Subject: [VE472] Grades
- Use SJTU jBox service to share large files ( $> 2$  MB)

Never send large files by email

## Course arrangements:

- Lectures:
  - Tuesday 16:00 – 17:40
  - Thursday 16:00 – 17:40
- Labs: Wednesday 18:20 – 20:40

## Office hours:

- Anytime on Piazza
- On appointment

## Primary goals:

- Understand how big data sets are analysed in practice
  - Be able to use Hadoop
  - Learn how to work in the Hadoop ecosystem
- Be able to performed advanced data analysis on large data sets
  - Get good foundations on big data analysis
  - Be able to design, implement, and use advanced algorithm in Spark

*Be able to analyse any given dataset, regardless of there size*

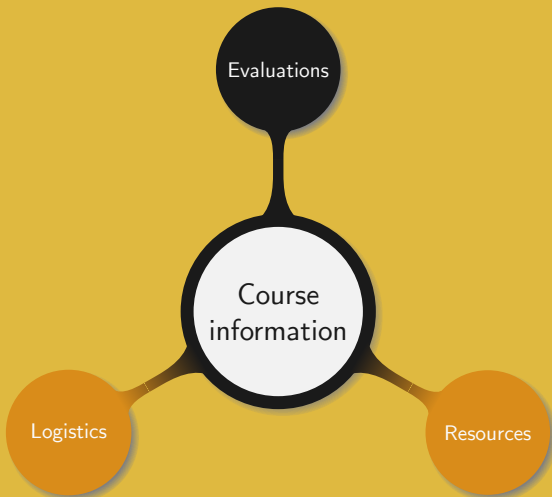
## Learning strategy:

- Course side:
  - ① Understand the new issues appearing as datasets grow
  - ② Be able to setup a Hadoop cluster and use it
  - ③ Understand why traditional algorithms fail on big data
  - ④ Be able to implement advanced algorithms for big data
- Personal side:
  - ① Derive algorithms for big data
  - ② Use and work “inside” Hadoop, Drill, and Spark
  - ③ Relate known strategies to new problems
  - ④ Perform extra research



### Detailed goals:

- Understand the basic logic behind Hadoop
- Have a general knowledge of the Hadoop ecosystem
- Be familiar with the basic Hadoop components: HDFS, YARN, and MapReduce
- Understand the structure of Drill and Spark
- Be able to work in Hadoop and “extend” its functionalities
- Know what tool to use for common specific purposes related to the study of big data
- Be familiar with common dimension reduction techniques
- Understand the limitations when facing “real” big data
- Be able to run basic data analysis on big data



### Homework:

- Total: 5 or 6
- Content: basic Hadoop, algorithms, Spark

### Labs:

- Total: 12
- Content: guided sessions to setup and work with Hadoop, and Spark

### Projects:

- Total: 1
- Content: analysis of some big dataset

### Challenge:

- Total: 1
- Content: compare theory and practice in Hadoop and Spark implementations

### Grade weighting:

- Midterm exam: 15%
- Final exam: 15%
- Quizzes: 20%
- Projects: 30%
- Homework: 10%
- Labs: 10%

Assignment submissions:  $-10\%$  per day, not accepted after 3 days

*Grades will be curved with the median in the range  $[[B, B+]]$*

## General rules:

- Not allowed:
  - Reuse the code or work from other students or groups
  - Reuse the code or work from the internet
  - Share too many details on how to complete a task
- Allowed:
  - Reuse part the course or textbooks and quoting the source
  - Share ideas and understandings on the course
  - Provide hints on where or how to find information

Documents allowed during the exams:

- Midterm: none
- Final: a single A4 paper sheet with original handwritten notes

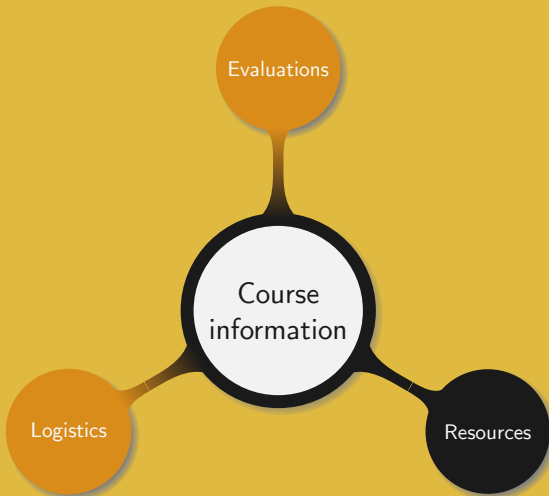
Group works:

- Every student in a group is responsible for his group's submission
- If a student breaks the Honor Code, the whole group is guilty

Contact us as early as possible when:

- Facing special circumstances, e.g. full time work, illness
- Feeling late in the course
- Feeling to work hard without any result

**Any late request will be rejected**





## Information and documents available on the Canvas platform:

- Course materials:
  - Syllabus
  - Lecture slides
  - Homework
  - Labs
  - Projects
- Course information:
  - Announcements
  - Notifications
  - Grades
  - Polls

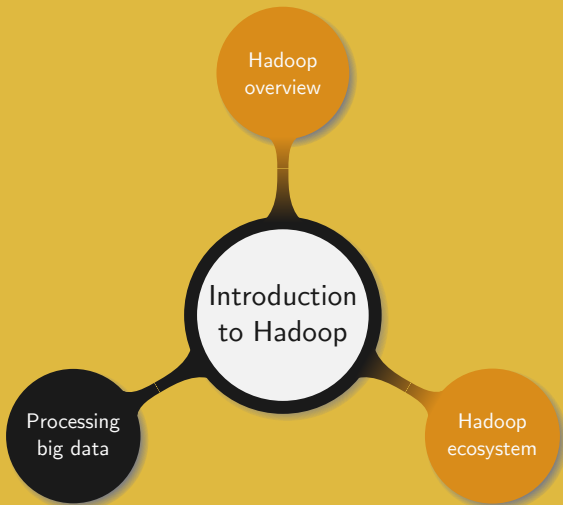
Useful places where to find information:

- *Hadoop the definitive guide*
- *Spark the definitive guide*
- *Machine learning, an algorithmic perspective*
- *Introduction to Data Mining*, by Tan et al..
- *Mining of Massive Datasets*, by Leskovec et al.. by White
- Search information online, i.e.  $\{\text{websites} \setminus \{\text{non-English websites}\}\}$



# 1. Introduction to Hadoop

---



Generated data is often:

- Stored, e.g. in databases
- Preprocessed, e.g. cleaned
- Analysed, e.g. machine learning

Most common advanced analytics:

- Supervised learning: predict a label based on some features
- Recommendation: suggest product based on users' behaviour
- Unsupervised learning: discover structure in the data
- Graph analytics: searching for patterns

Problem for a regular computer:

- Fast CPU
- Large memory
- Limited throughput

Mitigating the problem:

- Use caching
- Apply branch prediction
- Parallel read using RAID

Example. The speed of a disc read decreased relatively over time:

- 1990: 1.5 GB HDD at 4.4 MB/s
- Today: 1 TB HDD at 100 MB/s

*Scanning a whole disc in 1990 took 5 min, today it takes over 2.5 h!*

160

## A few numbers:

- 90% of the data was created in the past two years
- 40% of the data is generated by machines
- Over 26 billion IoT devices are activated

120

## Everyday:

- Google processes 3.5 billion search queries
- Facebook generates 4 petabytes of data
- 306 billion emails are sent

100

80

60

50

40

30

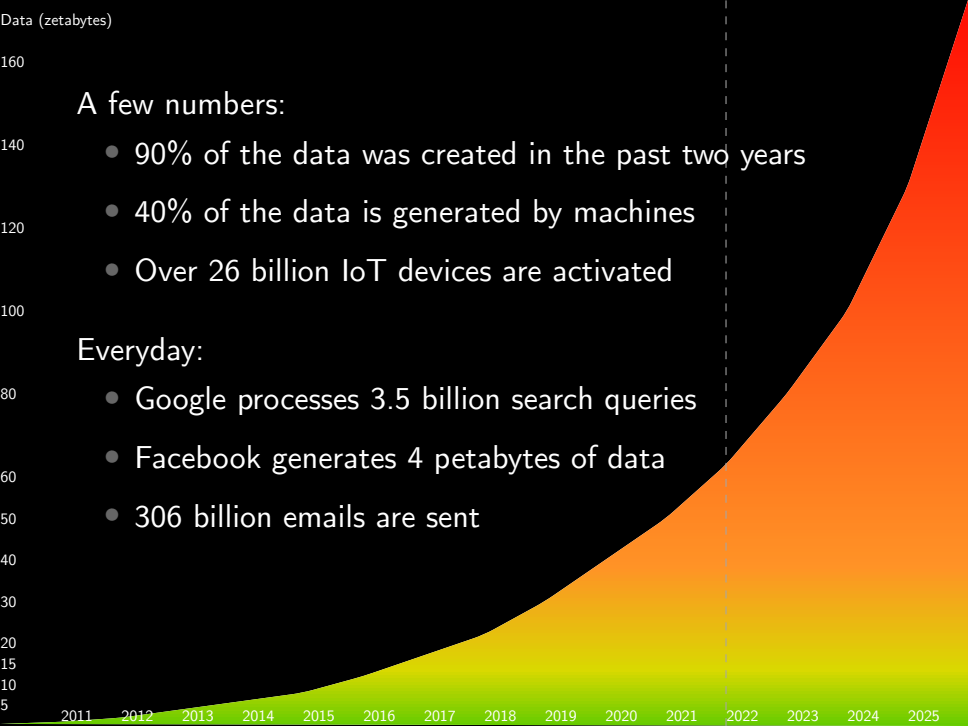
20

15

10

5

2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025





How to store and process data  
as it grows very big?

## Relational Database Management Systems:

- Data size: gigabytes
- Access: interactive and batch
- Update: read|write small proportions of the data
- Structure: schema defined at writing time
- Efficiency: low-latency retrieval for small amount of data

## Limitations of databases:

- Hard drive seek time increases slower than data transfer rate
- Data is often unstructured
- Slow to process as designed for read|write many times

## High-performance computing (HPC):

- Distributes computation across a cluster of machines
- Uses message passing interface
- Fits compute-bound jobs
- Data-flow controlled by programmer

## Limitations of HPC:

- Handling of node or process failure
- Require very high network bandwidth
- Expensive infrastructures, complex to extend
- Low level APIs

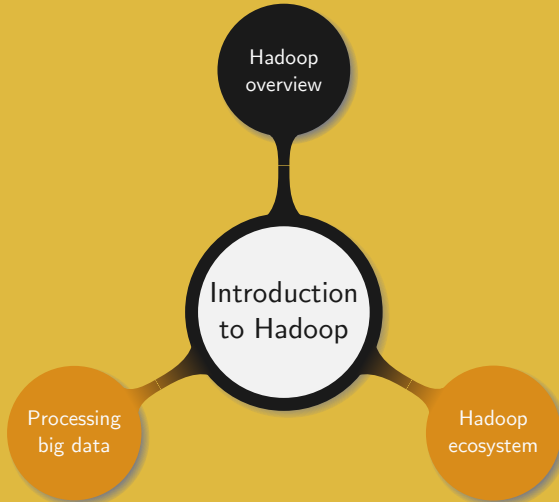
Random Access Machine (RAM) model:

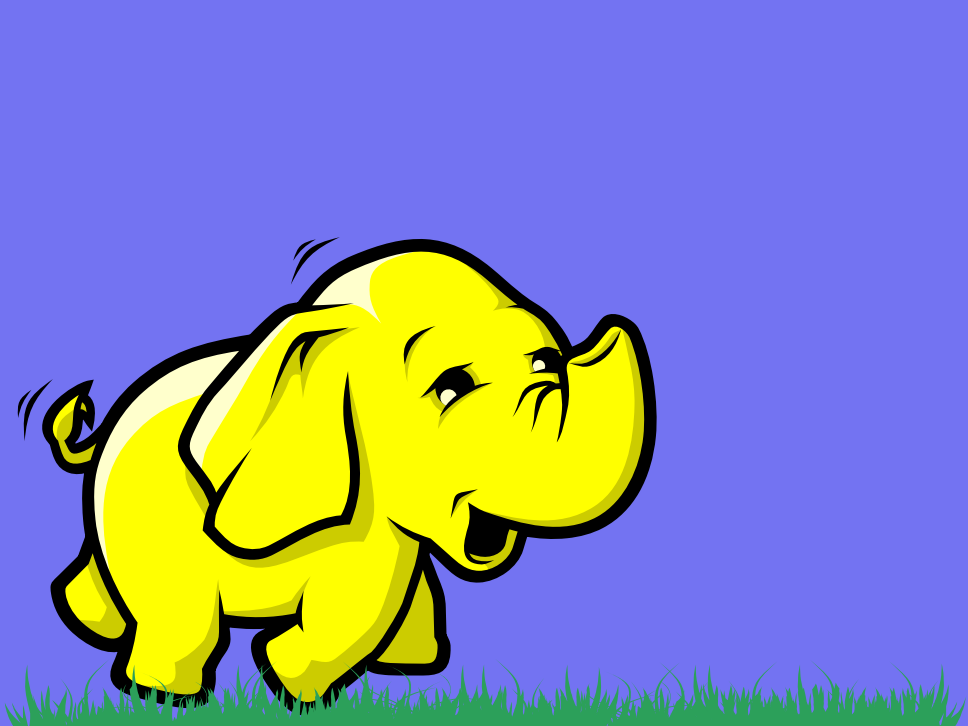
- A processor with a memory attached to it
- Each operation has a constant cost
- Runtime is proportional to the number of operations

Parallel Random Access Machine (PRAM) model:

- Several processors with one or more memory modules attached
- Need to specify how to deal with concurrent writes
- Each operation has a constant cost
- Runtime is defined when the slowest processor completes

*When dealing with “real” big data we need a distributed system*





## The birth of Hadoop:

- 2002: Nutch, an open source web search engine
- 2003: paper describing Google File System (GFS)
- 2004:
  - NDFS: open source implementation of GFS for Nutch
  - Paper describing data processing on large clusters (MapReduce)
- 2005: open source implementation of MapReduce for Nutch
- 2006:
  - NDFS and MapReduce moved out of Nutch
  - Hadoop 0.1.0 released
  - Hadoop is run in production at Yahoo!

## The adolescence of Hadoop:

- 2007 – 2008:
  - Number of companies using Hadoop jumps from 3 to over 20
  - Creation of Cloudera, first Hadoop distributor
- 2009:
  - MapR, new Hadoop distributor
  - HDFS and MapReduce become separate projects
- 2010 – 2011:
  - Many new “components” added to the Hadoop ecosystem
  - Receive two prizes at the Media Guardian Innovation Awards



## The maturity of Hadoop:

- 2012:
  - Hadoop 1.0 released
  - YARN ready to replace MapReduce (Hadoop 2.0)
- 2013 – 2014:
  - More than half of the Fortune 50 use Hadoop
  - Spark and Drill added to the Hadoop ecosystem
- 2017: Hadoop 3.0 released

Context where to adopt Hadoop:

- Massive amount of data to analysed
- Data stored over hundreds or thousands of computers
- Computation must be completed even if some nodes fail
- Cluster composed of commodity or high-end hardware

Hadoop's records:

- 2006: sort 1.8 TB of data in less than 48 h
- 2008: sort 1 TB of data in 209 s
- 2009: sort 1 TB of data in 62 s
- 2014: sort 100 TB of data in less than 23 min 30s

*The end goal is to efficiently analyse massive amount of data*

Hadoop is composed of core modules:

- Hadoop common: base libraries and utilities used by other modules
- Hadoop Distributed File System (HDFS): distributed file system
- Hadoop MapReduce: implementation of the MapReduce model
- Apache Yet Another Resource Negotiator (YARN): manages the cluster resources and schedules the user's tasks

Languages:

- Mainly Java
- Some C
- Shell scripts for command line utilities

## Characteristics of HDFS:

- Large files: at least hundreds of megabytes to terabytes
- Streaming data access: write once, read many times
- Commodity hardware: inexpensive common hardware

## Limitations of HDFS:

- High throughput at the expense of latency
- The “Master node” keeps the filesystem metadata in memory
- Write always in append mode, by a single writer

Programming paradigm composed of three main steps:

- Map:
  - A master node distributes the work and ensures exactly one copy of the redundant data is processed
  - Each worker node considers its local data and transforms it into key-value pairs
- Shuffle: each worker node redistributes its pairs based on the keys
- Reduce: each worker node combines a set of pairs into a smaller one

### MapReduce requirements:

- Mapping operations must be independent of each others
- Parallelism is limited by the number of sources and nearby CPUs
- Either all the output sharing the same key must be processed by a single reducer or the reduction must be associative

### MapReduce benefits:

- Highly scalable on commodity hardware
- Possible to recover for partial failure
- Great efficiency due to parallelism

A *container* is an environment with restricted resources where application-specific processes are run

YARN provides two types of daemons:

- Resource manager:
  - One per cluster
  - Manages the resources for the whole cluster
- Node manager:
  - One per cluster node
  - Launches and monitors containers

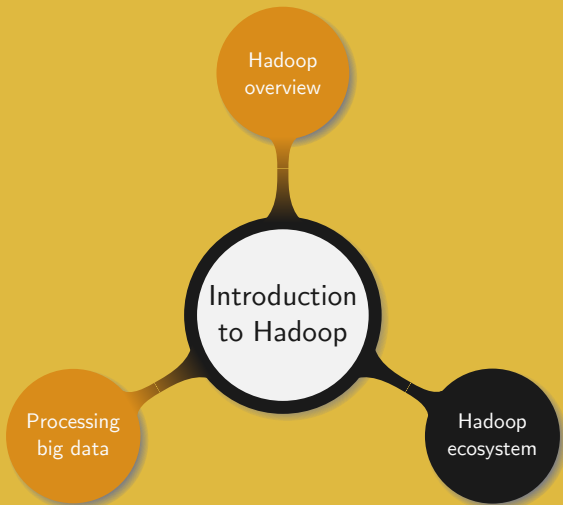
In Hadoop 1, MapReduce:

- Directly interacts with the filesystem
- Manages resources

In Hadoop 2, YARN:

- Manages the resources
- Interacts with the filesystem
- Hides low level details from the user
- Offers an intermediate layer supporting many other distributed programming paradigms





Goal: global scalable resource manager, not restricted to Hadoop

Mesos scheduling:

- Determine the available resources
- Offer “various options” to an application scheduler
- Allow any number of scheduling algorithm to be developed, plugged, and used simultaneously
- Each framework decides what scheduling algorithm to use
- Mesos allocates resources across the schedulers, resolves conflicts, and ensures a fair share of the resources

Goal: use Mesos to manage YARN resource requests

Simplified strategy:

- ① A job requests resources to YARN
- ② YARN uses the Myriad scheduler to allocate resources
- ③ Myriad scheduler matches requests to Mesos' resources offers
- ④ YARN allocates the resources

Benefits:

- Get the best from both worlds
- Give more flexibility to YARN

## Goals:

- Be a full replacement for MapReduce
- Efficiently support multi-pass applications
- Write and read from the disk as little as possible
- As much as possible take advantage of the memory

## Main ideas:

- Resilient Distributed Dataset (RDD): contains the data to be transformed or analysed
- Transformation: modifies an RDD into a new one
- Action: analyses an RDD

## Goals:

- Integrate into Hadoop as a MapReduce replacement
- Be an interactive ad-hoc analysis system for read-only data
- Be easily expandable using storage plugins
- Enjoy data agility

## Main ideas:

- Columnar execution: shredded, in-memory columnar data representation
- Runtime compilation and code generation: compile and re-compile queries at runtime
- Optimistic execution: stream data in memory to minimise disk usage

When to use Spark or Drill:

- Drill is an ANSI SQL:2003
- Spark has SQL query capabilities
- Drill allows fine grained security at the file level
- Drill is best used as a distributed SQL query engine
- Spark is best used to perform complex math, statistics, or machine learning

## Basics on Flink:

- Allows the execution of dataflow programs following a data-parallel and pipelined approach
- Provides a high-throughput and low-latency streaming engine
- Nicely handles node failures
- Can connect to various storage types

## Basics on Tez:

- Targets batch and interactive data processing applications
- Intends to improve MapReduce paradigm
- Exposes more simple framework and API to write YARN applications
- Expresses computation as a dataflow graph



## Basics on HBase:

- NoSQL database system for distributed filesystems
- Low latency access to small amount of data in a large data set
- Fast scan across tables
- Random access to rows

## Common use cases:

- Applications requiring sparse rows
- Not good for relational analytics and transactional needs

### Basics on Hive:

- Access SQL data in HDFS using an SQL-like query language HQL
- Convert queries to MapReduce, Tez, or Spark jobs
- Warning: does not fully comply to ANSI-standard SQL

### Basics on Spark SQL (formerly Shark):

- Was initially a port of Hive to Spark
- Follows Spark in-memory computing model
- Is “mostly” compatible with HQL

### Basics on Presto:

- Supports ANSI-SQL standard
- Uses a custom engine, not based on MapReduce
- Can access various data sources through storage plugins

Avro:

- Input: a schema describing the data and the data
- Output: generates the code to read/write data

Parquet:

- Columnar storage format
- Complex to handle

Java Script Object Notation (JSON):

- Not part of Hadoop
- Often preferred to XML by Hadoop community
- Represent data using key-value pairs

### Ambari:

- Production-ready, easy to use web-based GUI for Hadoop
- Eases the installation and monitoring of a cluster

### Zookeeper:

- Effective mechanism to store and share small amounts of states and configuration across the cluster
- Not a replacement for any key-value store
- Has built-in protections to prevent using it as large data-store
- Used as a coordination service

Major analytics helpers:

- Pig: high-level language to speak to MapReduce
- Hadoop streaming: write mappers/reducers in any language
- Mahout: set of scalable machine-learning algorithms for Hadoop
- MLlib: similar to Mahout, based on Spark (maintenance mode)
- Spark ML: similar to MLlib based on a higher level API
- Hadoop Image Processing Interface: package allowing to examine images and determine their differences and similarities

## Moving data to and from Hadoop:

- Sqoop: transfer data between HDFS and relational databases
- Flume: distributed system for collecting, aggregating, and moving large amount of data from various sources into HDFS
- Distributed Copy (DistCP):
  - Part of basic Hadoop tools
  - Used to move data between the clusters
  - Is the basis for more advanced Hadoop recovery tools

### Lambda data architecture:

- Setup three layers:
  - Batch layer: store all incoming data and batch process it
  - Speed layer: analyse incoming data in real time
  - Serving layer: serve curated data that can be analysed by other tools
- Drawback: maintain two code sets for batch and speed layers

### Kappa data architecture:

- Not a replacement but an alternative to lambda architecture
- Layers: batch layer is removed compared to lambda architecture
- Suitable for systems with strict end-to-end latency requirements
- Drawback: replay the whole stream in case of error

### Apache Storm:

- Distributed system for real-time processing of streaming data
- Able to process over a million records per second per cluster node
- Relies on Zookeeper for coordinating the nodes

### Apache Kafka:

- Distributed platform used to create real-time streaming data pipelines
- Heavily relies on zerocopy (OS kernel level) to move data around
- Commonly used together with Spark, Flink, or Storm

### Remote Dictionary Server (Redis):

- In-memory key-value data-store
- Extremely fast, simple, and versatile
- Benchmarked as the fastest DB in the world

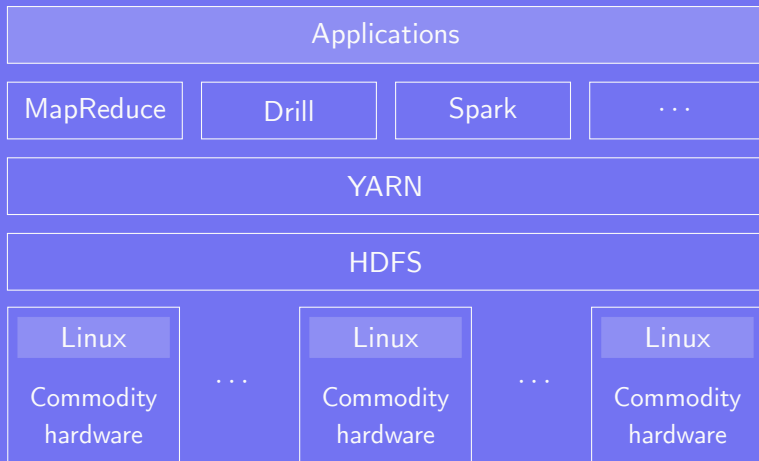


Ray:

- Spark's goal was to replace Hadoop and Ray's goal is to replace Spark
- Run fast machine learning or deep learning-based applications
- Hope to reach MPI power and granularity levels

TensorFlow:

- Python-friendly library for fast and easy machine learning computation
- Data is stored as a tensor
- Create dataflow graphs where the edges are tensors and vertices mathematical operations
- Can run on Hadoop (Yarn), Spark, and Ray

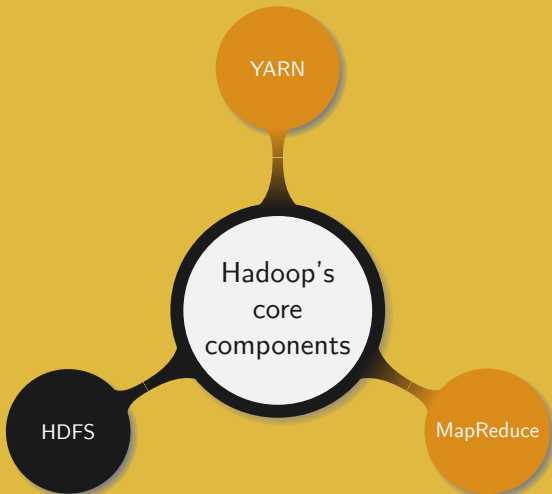


*Refer to Hadoop ecosystem table for more details*



## 2. Hadoop's core components

---



Regular filesystems on a computer:

- Partition
- Hard drive
- LVM

Distributed filesystems:

- Spans several computers
- Has to deal with potential network issues

*Idea behind HDFS summarised on slide 1.36*

## Blocks in HDFS:

- Default size of 128 MB
- Files smaller than a block size do not occupy the whole block
- A file can be larger than a whole disk
- Data and metadata handled separately
- Easy to implement fault tolerance and availability

Two types of node in a cluster:

- Namenode:
  - Maintains FS tree and metadata for all files and directories
  - Locally stores information in namespace image and edit log
  - Knows on which datanodes the blocks of a file are located
- Datanode:
  - Stores and retrieve blocks
  - Regularly reports the list of stored blocks to namenode
  - Can store certain blocks in cache



A namenode has no persistent copy of where blocks are:

- Each datanode announces the blocks it has
- All the information is kept in memory by the namenode
- When a write occurs an entry is added to the edit log

*What to do if the namenode fails?*

A namenode stores all the blocks of all the files in its memory:

- Assume 1 GB of memory for 1 million blocks
- 200 nodes cluster, 24 TB each:  $\sim 12$  GB of memory
- Cluster at Yahoo!: 25 PB  $\rightarrow \sim 64$  GB of memory
- Cluster at Facebook: 60 PB  $\rightarrow \sim 156$  GB of memory

*How about having more namenodes?*

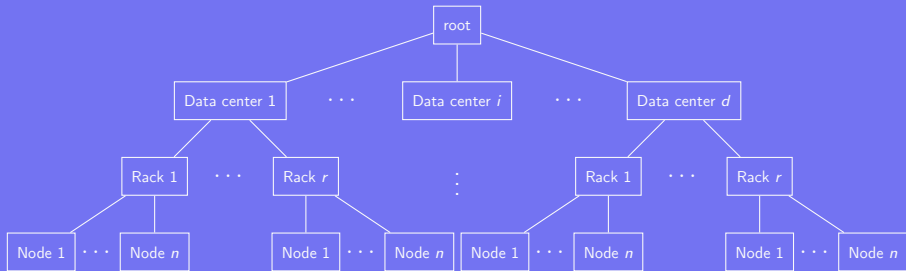
Allowing more namenodes:

- Split the filesystem over several independent namenodes
- Each namenode has a namespace
- Each namespace has its own pool of blocks
- A namespace with a block pool is called namespace volume
- A datanode is not attached to a specific namespace volume

Two namenodes in an active-passive mode:

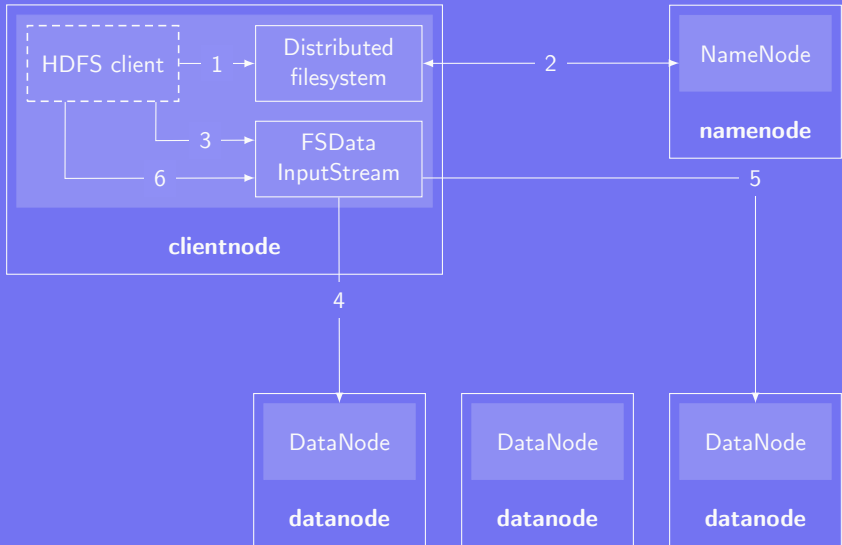
- Passive node takes over in case of failure of the active one
- The two namenodes share the same edit log
- Only the active namenode can write to the edit log
- Passive namenode reads entries when written in edit log
- Datanodes send block reports to both namenodes
- Passive namenode also works as secondary namenode
- Clients must be configured to handle namenode failures

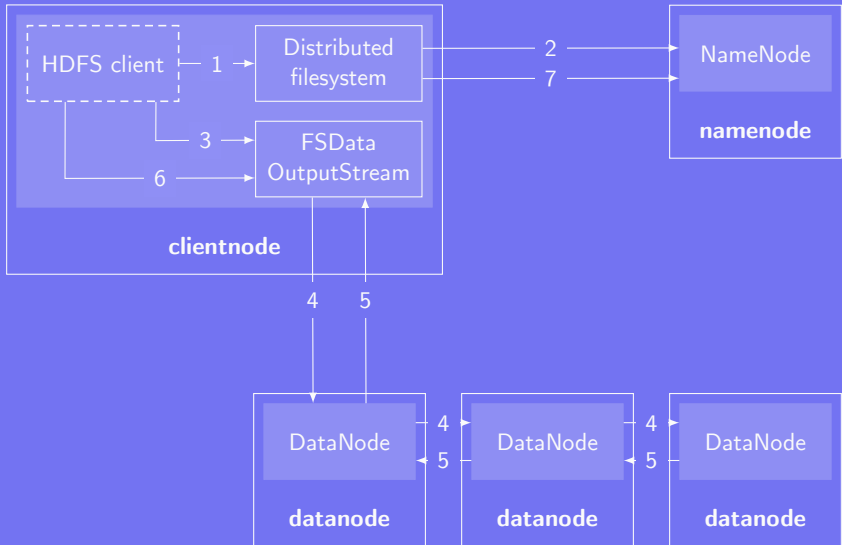
The goal is to evenly spread blocks across the whole cluster:



Replicas' location:

- First: same node as the client
- Second: random, different rack from the first
- Third: same rack as the second but different node
- Others: random nodes in the cluster





When write on a data node fails:

- 1 Close the pipeline
- 2 Add packets in front of the acknowledgment queue
- 3 Inform the name node of the failing data node
- 4 Remove the faulty data node from the pipeline
- 5 Construct a new pipeline using only the healthy data nodes
- 6 Complete the writing of the block across the pipeline
- 7 Arrange for the replication of under-replicated blocks

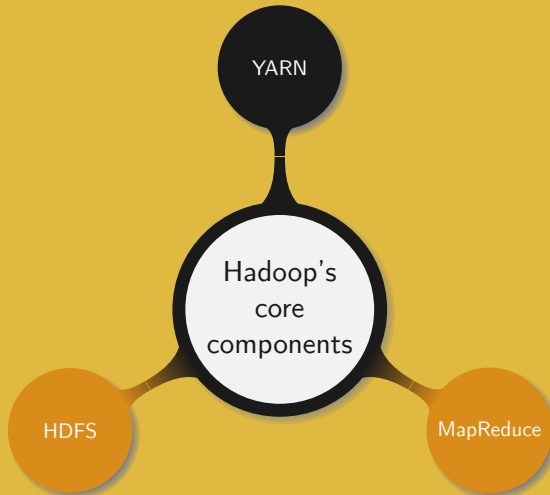


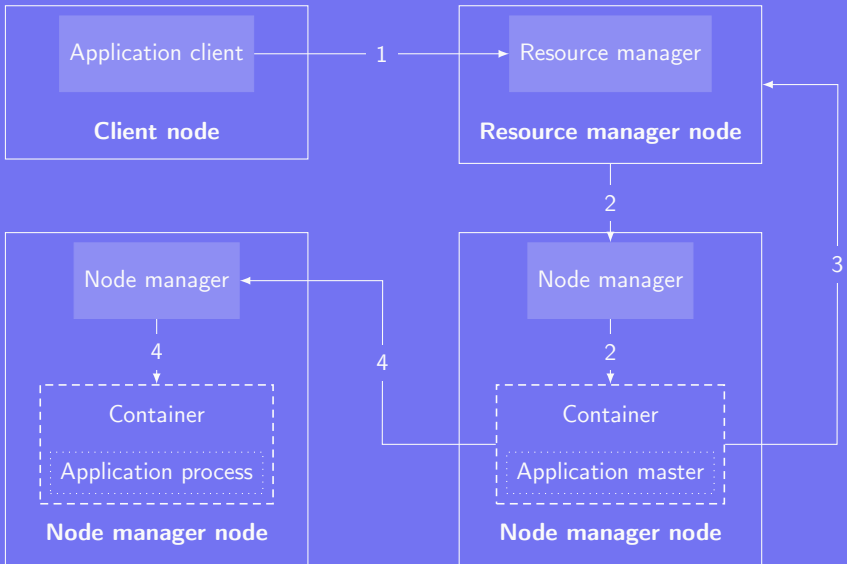
Alternative filesystems built on top of Hadoop abstract filesystem:

- HAR: pack HDFS files into an archive
- View: used to create mount points for federated FS
- FTP: FS backed by an FTP server
- S3: FS based on Amazon S3
- Azure: FS developed by Microsoft Azure

Interfaces to access HDFS data:

- libhdfs: C library with API similar to the Java one
- NFS: use Hadoop NFS gateway
- FUSE: based on libhdfs





A request has two components:

- Amount of resources for each container
- Preferred location of the containers

Common strategy for a request:

- Announce all the resources requests at the beginning
- Dynamically request resources based on the needs

YARN can be used in three ways by applications:

- One application per user job: simplest model
- One application per user session:
  - Containers can be reused between jobs
  - Possibility to cache data between jobs
- Long-running application shared among users:
  - Application master is always “on”
  - Application master acts as a coordinator for other applications

Three schedulers available in YARN:

- FIFO: request served one by one in a queue
- Capacity:
  - Define queues based on the “size” of the jobs to complete
  - All the jobs start early
  - Resources are wasted when unused
- Fair:
  - Resources are dynamically balanced over all the jobs
  - All the resources are fully used
  - Delay due the resource reallocation

## Capacity scheduler setup:

- Each queue is handled as a FIFO
- Queue elasticity
  - A single job cannot exceed the capacity of the queue
  - The capacity can be exceeded when several jobs wait and resources are available
- Containers are not preempted
- Can control:
  - The number of resources per user/application
  - The number of applications that can be run at a time
  - Access Control Lists (ACL) on the queues

*The challenge is to find a reasonable trade-off for the capacity*

## Fair scheduler queues:

- One or more queues allowed:
  - Single queue: resources are fairly shared among all applications
  - Several queues, each having:
    - Its own scheduling policy
    - A max/min resources and number of applications
- Queues can be precisely configured using an allocation file
- By default a queue is dynamically created for each user



Preemption in the fair scheduler:

- Efficiency is reduced as killed containers must be restarted
- Two timeout settings  $t_1$  and  $t_2$  to trigger preemption:
  - Minimum share: if a queue waits longer than  $t_1$
  - Fair share: if a queue remains below half of its fair share for longer than  $t_2$
- Timeouts can be set per queue or globally

Locality problem when scheduling:

- An application requests a specific node
- The node is busy
- Should the application wait for the node or loosen its request?

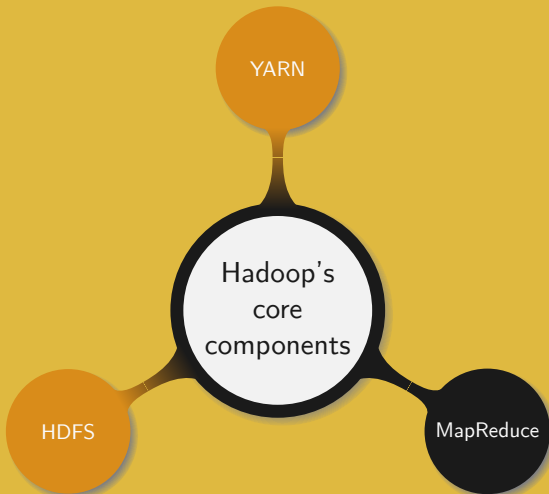
YARN schedulers' approach:

- Every second each node manager sends a heartbeat reporting the running containers and available resources
- Capacity scheduler: wait for a predefined number of heartbeats before loosening the requirement
- Fair scheduler: wait for a predefined portion of nodes in the cluster to offer opportunities before loosening the requirement

*How to fairly share resources between applications when they do not use the same type of resources?*

Basic idea for two applications:

- Consider the proportion of resources requested for a container by each application
- Call the largest proportion the dominant resource and use it as measure of cluster usage
- Proportionally offer less containers to the more demanding application



Parties involved in a MapReduce job:

- A client which initiates the job
- YARN resource manager
- YARN node manager
- MapReduce application master
- HDFS

Starting a MapReduce job:

- 1 Request a new application ID to the resource manager
- 2 Check the job parameters
- 3 Split the job into subtasks
- 4 Copy the splits and other necessary information to run the job onto the shared FS
- 5 Effectively submit the job on the resource manager

## Running a MapReduce job:

- ① YARN scheduler allocates a container
- ② Application master launched by the resource manager
- ③ Setup the tasks
- ④ Retrieve the splits from the shared FS
- ⑤ Create a Map task for each split and specify the number of tasks for the Reduce part
- ⑥ Resources for
  - a Small tasks: run on the same node
  - b Large tasks: contact the resource manager for more containers
    - i Request resources for the maps (high priority)
    - ii Request resources for the reducers when enough maps have completed
- ⑦ Locate the data on the distributed FS and start the task

Potential points of failure:

- Task failure
- Application master failure
- Node manager failure: YARN level (no heartbeat received)
- Resource manager failure: YARN level (high availability mode)

Protection and progress monitoring:

- Tasks are run in a separate JVM: avoid crashing namenode
- Each task has a status and some counters
- Each task reports its progress to the application master
- Client application polls the application master every second to retrieve the latest status



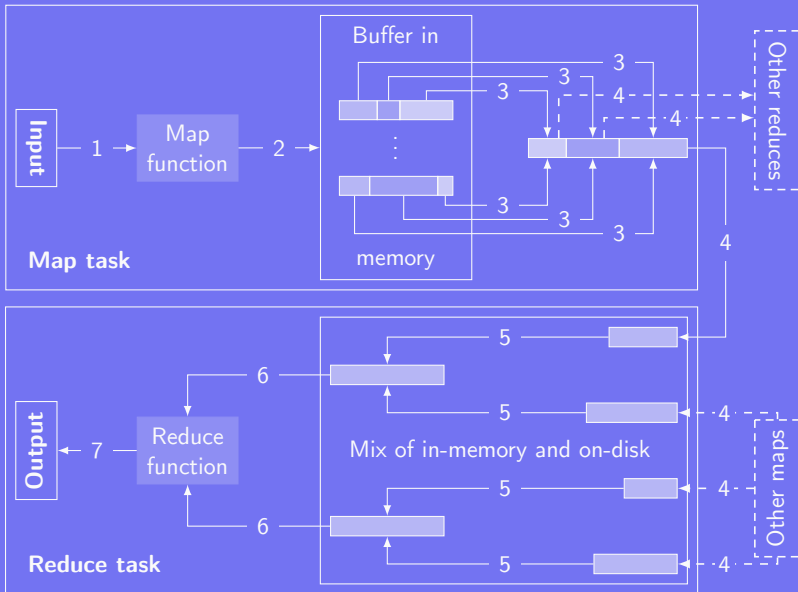
## Task failure occurrences and solutions:

- The map or reduce task fails:
  - When receiving a failure notice the application master marks the task as failed
  - The container is freed and resources released
- The JVM crashes: the node manager notices the application master of the failure
- A task hangs:
  - Tasks marked as failed if no report is received
  - The JVM is killed by the application master

*Failed tasks are rescheduled on a different nodemanager*

### Failure detection and recovery:

- The application master sends periodic heartbeats to the resource manager
- On failure a new instance is run on a new container
- The tasks progress is known so it is possible to resume without re-running the completed tasks
- Each task caches the application master's address
- Use a timeout after which the resource manager is contacted for the new application's master address



### Optimized configuration setup:

- Provide shuffle with as much memory as possible
- Keep enough memory for map and reduce functions
- Optimize the code with respect to memory consumption
- Minimize the number of spills for the map part
- As much as possible keep intermediate reduce data in memory

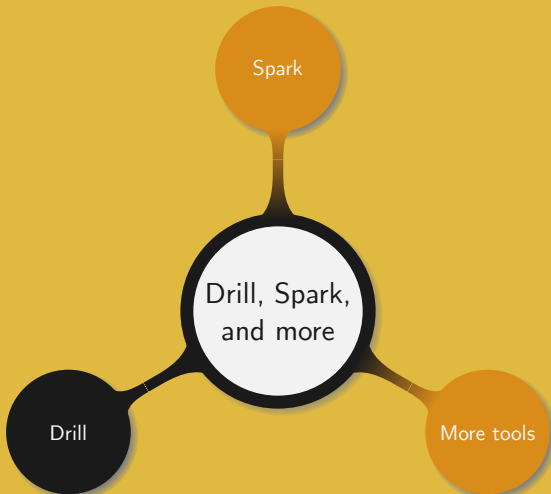
### Speculative execution:

- A task is detected as much slower than average
- Re-run it on a different node
- Kill all the other duplicates as soon as one completes



### 3. Drill, Spark, and more

---



Parties always involved in a Drill job:

- A client which initiates the job
- Zookeeper

Parties optionally involved in a Drill job:

- YARN
- HDFS
- Hive
- HBase

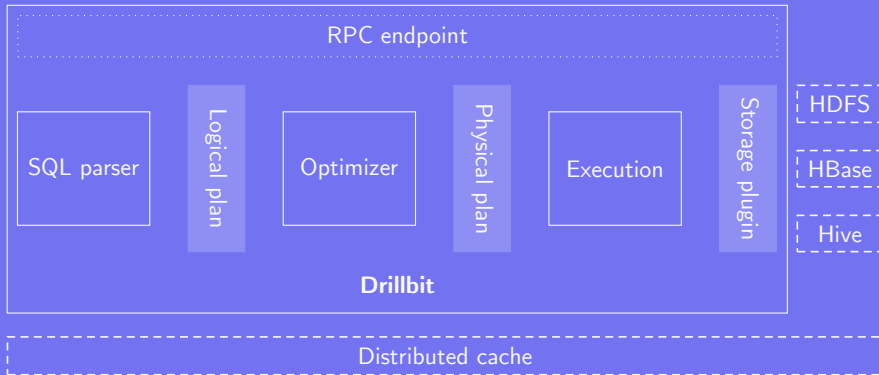


Running Drill as a YARN application:

- ① Start Drill on the client machine
- ② Upload resources to the FS and request resources for the application master
- ③ Ask a node manager to prepare and start a container for the application master
- ④ The application master contacts the resource manager to obtain more containers

Running Drill as a YARN application:

- ⑤ Request the start of Drill software on each assigned node
- ⑥ Start a “Drill process” called a *drillbit*
- ⑦ Each drillbit starts and registers with Zookeeper
- ⑧ The application master checks the health of each drillbit through Zookeeper
- ⑨ Use Zookeeper to retrieve information on the drillbits, run queries, etc.



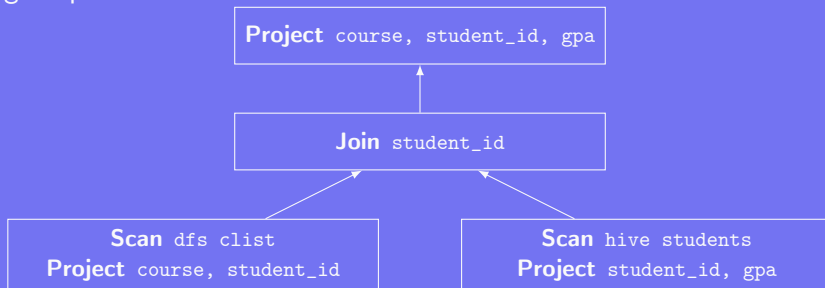
Foreman drillbit:

- Drillbit that receives the query
- It drives the entire query

Initial SQL query:

```
1 SELECT course, student_id, gpa
2 FROM clist.json l, hive.students s
3 WHERE l.student_id = s.student_id
```

Logical plan:

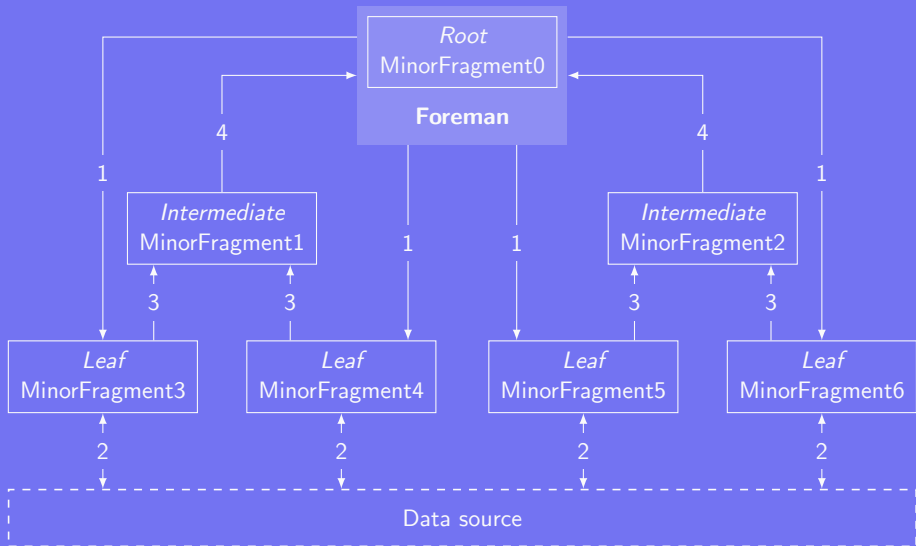


Major fragment:

- Concept representing a phase of the query execution
- Composed of minor fragments

Minor fragment:

- Logical unit of work running in a thread
- Contain one or more relational operators
- Usually as numerous as the number of available drillbits
- Scheduled based on data locality when possible and round-robin otherwise



### Architecture:

- No central server, no master-slave concept
- Each drillbit contains all the services and capabilities of Drill
- Nodes can be added or removed at no cost

### Columnar execution:

- Avoid access for columns not involved in the query
- Directly performs SQL processing on columns

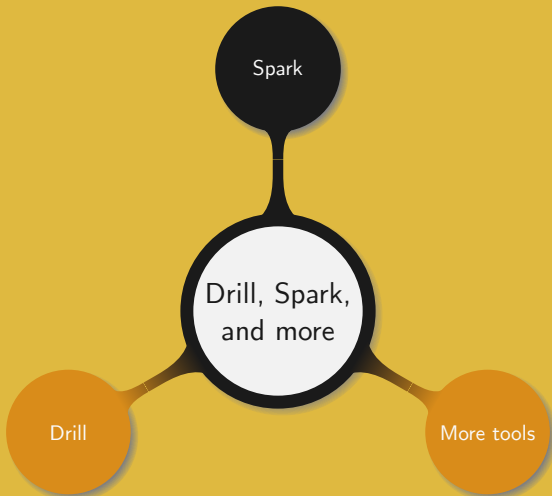
Optimistic query execution:

- Assume no failure will occur during query execution
- Rerun the query in case of failure
- Only write on disk when memory overflows

Vectorization: allow the CPU to operate on vectors

Runtime compilation: generate efficient code for each query





## Spark organisation:

- Application: user program built on Spark and composed of:
  - A driver program: process running the main function
  - Executors: processes launched for an application on worker nodes
- The driver program connects to a cluster manager
  - Standalone: cluster manager provided with Spark
  - YARN
  - Mesos
  - Kubernetes

Two modes available:

- Client mode:
  - Driver runs in the client
  - Required in the case of interactive programs
  - Useful when building a Spark program
- Cluster mode:
  - The entire application runs in the cluster
  - Appropriate from production jobs
  - YARN application master failure strategy (slide 2.90) is applied

Spark job workflow:

- ① Start the driver program on a client node
- ② The driver requests a container to the resource manager
- ③ A container starts and runs an Executor Launcher application master
- ④ The Executor Launcher requests more resources to start Executor backends processes in new containers
- ⑤ Each Executor Backend registers with the driver

Workflow similar to client mode but:

- The driver program runs in a YARN application master process
- The client submit a job but does not run any user code
- The application master starts the driver program
- The driver program “replaces” the Executor Launcher

Remark. Data locality:

- Executors are launched before data locality information is available
- The driver can optionally specify preferred locations

## Resilient Distributed Dataset (RDD):

- Core abstraction in Spark
- Collection of objects distributed across a cluster:
  - Read-only: do not alter a dataset, transform it into a new one
  - Resilient: no disk write, reconstruct the RDD in case of partition loss
- Loaded as input:
  - Created from an external dataset
  - From an existing RDD
  - Parallelising an existing collection

Two types of operations on an RDD:

- Transformation:
  - Create a new dataset from an existing one
  - Only compute the result when an action is run
  - Do not return any result to the driver program
- Action:
  - Run a computation on a dataset
  - Return the value to the driver program

Benefits of this approach:

- Transformed RDD is in memory when performing an action
- No large dataset to send back to the driver program

Datasets are cached in memory across operations:

- An RDD is stored on the node where it was computed
- An old RDD is dropped following the LRU algorithm
- A lost RDD is automatically recomputed if needed

Caching levels:

- Memory only: no compression, lost partitions are recomputed
- Memory and disk: partitions that do not fit in the memory are spilled on disk
- Memory only serialized: compression enabled
- Replication: all the above but also replicate on another node



Serialization of data and functions:

- Used to share information among the executors
- Transparent to the user

Task closure:

- Cannot share variables among executors
- Determine what variables and methods an executor needs
- Serialize this closure and send it to the executor
- Each executor receives a copy of the original variable
- Variables are not updated on the driver

### Broadcast variables:

- Read-only variables broadcasted to each executor
- Data sent in an efficient way to minimize traffic
- Useful for data needed over several stages of the computation

### Accumulators:

- Variables that can be added to, using associative and commutative operations
- The driver can retrieve their value
- They are only updated on action tasks
- Update only occurs once, even if an action is rerun

### Job submission and execution:

- A job is submitted when an action is performed on an RDD
- The transformations on the RDD are organised into a logical execution plan
- Spark DAG scheduler transforms the logical plan into a execution physical plan
- The physical plan defines stages, split into tasks
- Spark task scheduler constructs a mapping of tasks to executors
- The executor runs the task
- Executors send status updates to the driver when a task is completed or has failed

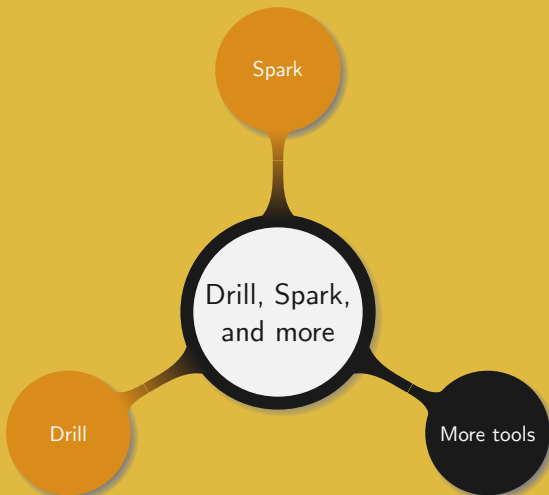
Newer and higher level abstractions relying on RDD:

- Datasets:
  - Foundational type of the structured APIs
  - Provide type-safety and allow much flexibility
  - Only available in Java and Scala and comes at a performance cost
- DataFrames:
  - Similar to a spreadsheet split into partitions
  - Internally defined as DataSets of type `Row`
  - Most common structured API, supported in all languages
- SQL Tables:
  - Data structure similar to DataFrames but defined within a database
  - Unmanaged table: defined from a file on the disk
  - Managed table: imported in and managed by Spark

### Advantages of DAG:

- Any lost RDD can easily be recovered
- Offers more possibilities than a simple Map and Reduce approach
- Transformation on RDDs are not directly applied:
  - Allows better optimizations
  - Decreases disk writes and data transfer

Remark. Spark generalises the MapReduce approach, is much faster, and features many more high-level operators



### Simple observations:

- Over 20 billions devices are connected to the internet
- The complexity of software keeps increasing
- New security challenges need to be addressed
- Package manager dependencies are complex to handle

### Alternative package management systems:

- Flatpak, Snap, AppImage:
  - Distribution agnostic packages
  - Applications are sandboxed, i.e. isolated from each others and the host
- Nix: *all* packages are isolated from each others

## Linux Containers (LXC):

- Operating-system-level virtualization method
- Relies on the kernel's *cgroup* and *namespace* isolation functionalities
- Concurrently run multiple isolated Linux OS on a machine
- Each container must be individually maintained
- Containers can be either privileged or unprivileged
- Containers access the bare machine and rely on the host kernel

*LXC requires the setup of a whole OS for each container*



Docker uses a different approach than LXC:

- Initially based on LXC but *now* completely independent
- A daemon manages the docker containers
- A container is an encapsulated environment that runs applications
- An image containing an application and its dependencies is built based on a “configuration file”
- To update simply replace the old image with a new one

*Docker needs to be “manually” deployed, managed, and scaled*

Kubernetes is a container orchestration tool:

- Mostly used with, but not limited to Docker
- Initially developed as an internal Google project
- Kubernetes is a cluster application that handles a cluster:
  - *Master*: controls all other machines in the cluster
  - *Nodes*: the machines onto which applications are running
  - *Pods*: single instance of an application or running process

## Main tasks of Kubernetes:

- Monitor the health of the running applications
- Balances the load
- Manages hardware resources allocation
- Eases the deployment of preconfigured applications
- Allows access to storage in the same way as any other resources

## During the lifespan of an application:

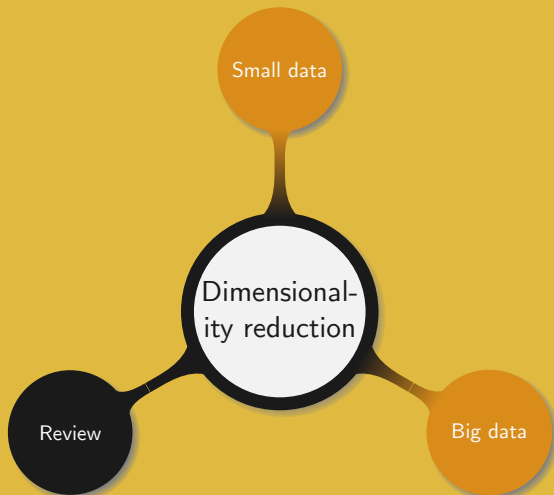
- Containers can live, die, be resurrected
- Kubernetes handles everything without any human interaction

*Kubernetes can be coupled to Hadoop or work independently*



## 4. Dimensionality reduction

---



Before the big data age:

- Extract a sample from a large population, e.g. phone survey
- Derive measurements on the population based on the sample
- Classify the population with respect to the measurements
- Take decisions based on some given goals

Limitations:

- Not all people in a same category behave similarly
- Goals must be general and cannot be personalised

Given a dataset  $X = [X_1, \dots, X_n]$ , we define the

- *Median*: which corresponds to the “middle” value
- *Mean*:  $\bar{X} = \sum_{i=1}^n \frac{X_i}{n}$
- *Standard deviation* as the average distance between  $X_i$  and  $\bar{X}$ :
  - It represents the spread of the dataset
  - It is given by  $\sigma = \sqrt{\frac{(\sum_{i=1}^n X_i - \bar{X})^2}{n-1}}$
- *Variance* which also measures the spread and is given by  $\sigma^2$

*Those measures are one-dimensional*



Given a dataset  $[X_1, X_2] = [[X_{1,1}, \dots, X_{1,n}], [X_{2,1}, \dots, X_{2,n}]]$ , *covariance*:

- Is given by

$$\sigma_{X,Y} = \frac{\sum_{i=1}^n (X_{1,i} - \bar{X}_1)(X_{2,i} - \bar{X}_2)}{n - 1}$$

- Verifies  $\sigma_{X_1, X_2} = \sigma_{X_2, X_1}$  and  $\sigma_{X, X} = \sigma^2$
- Evaluates how much two dimensions vary from the mean with respect to each other
- Provides information on whether both dimensions “increase together”

Given  $n$  variables  $X_1, \dots, X_n$ , their covariance matrix is given by

$$\begin{pmatrix} \sigma_{X_1, X_1} & \cdots & \sigma_{X_1, X_n} \\ \vdots & \ddots & \vdots \\ \sigma_{X_n, X_1} & \cdots & \sigma_{X_n, X_n} \end{pmatrix}$$

Let  $\mathcal{B}$  be a basis of a vector space  $V$ . For  $M \in \mathcal{M}_n(\mathbb{K})$ ,  $\lambda \in \mathbb{K}$  is an *eigenvalue* of  $M$  if and only there exists an *eigenvector*  $X \in \mathcal{M}_{n,1}(\mathbb{K})$  such that  $MX = \lambda X$ .

Important properties related to eigenvectors and eigenvalues:

- If  $M$  has rank  $n$  then it has  $n$  non-zero eigenvalues
- If  $M$  has  $n$  eigenvalues, distinct two by two, then it is diagonalizable
- $M$  is diagonalizable if and only if its eigenvectors form an orthogonal basis  $\mathcal{B}'$
- $M = PDP^{-1}$ , where  $D$  is a diagonal matrix featuring the eigenvalues of  $M$  on its diagonal, and  $P$  is the transition matrix from  $\mathcal{B}'$  into  $\mathcal{B}$
- For any  $k \in \mathbb{N}$ ,  $M^k = PD^kP^{-1}$

The *characteristic polynomial* of  $M$  is defined as

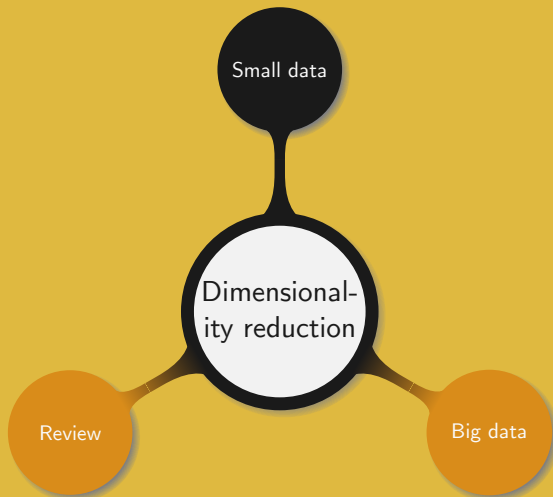
$$\begin{aligned}\chi_M : \mathbb{K} &\longrightarrow \mathbb{K} \\ \lambda &\longmapsto \det(M - \lambda I_n).\end{aligned}$$

Properties of  $\chi_M$ :

- If  $\lambda$  is an eigenvalue of  $M$ , then it is a root  $\chi_M$
- $M$  is diagonalizable if and only if  $\chi_M$  splits on  $\mathbb{K}$  and the dimension of the eigenspace associated to each eigenvalue  $\lambda$  is equal to the multiplicity of  $\lambda$

General remarks:

- Not all matrices are diagonalizable
- Not all square matrices are diagonalizable
- In general the determinant is “complicated” to compute
- A matrix that is not invertible is called *singular*



*I admire the elegance of your method of computation; it must be nice to ride through these fields upon the horse of true mathematics while the like of us have to make our way laboriously on foot.*

Albert Einstein

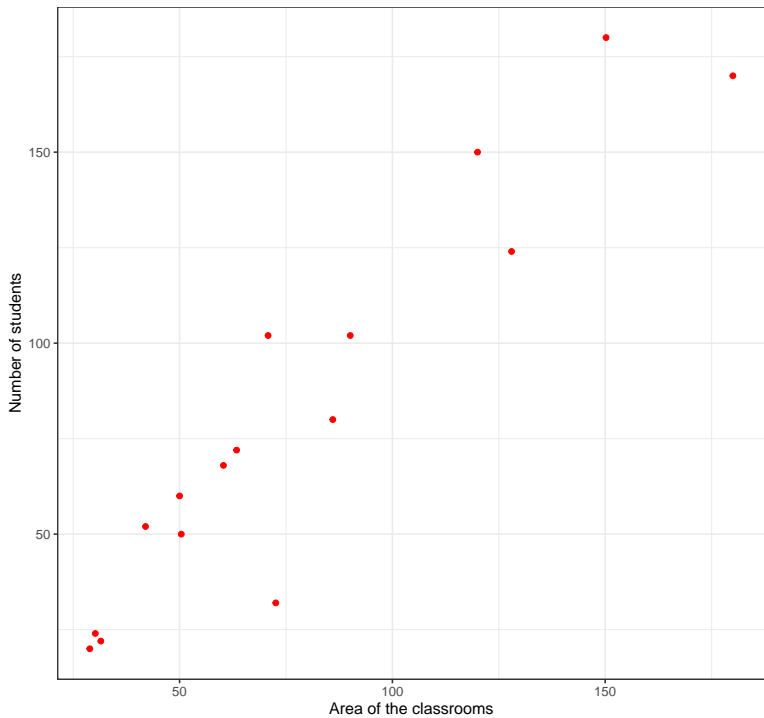
Basic idea behind Principal Component Analysis (PCA):

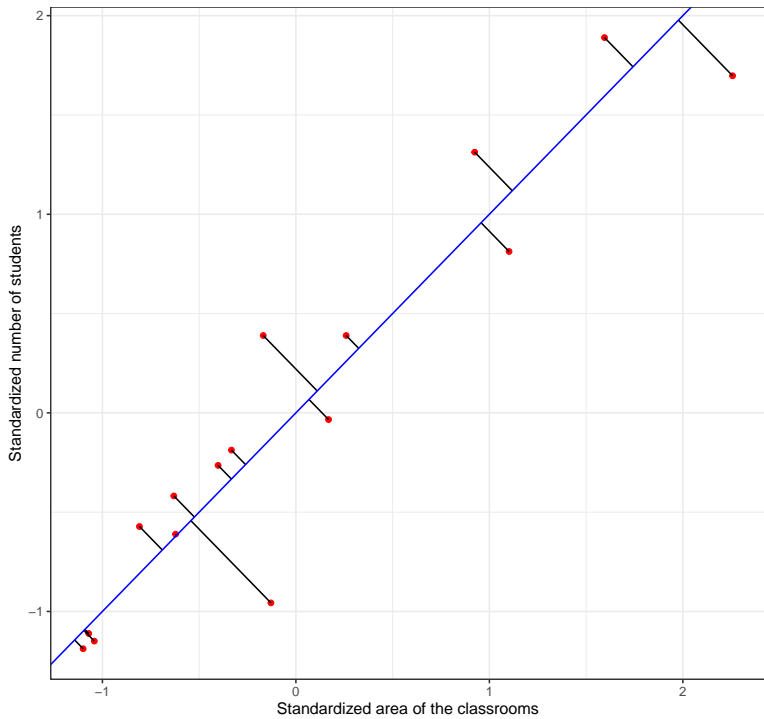
- Provides the best “perspective” that emphasises similarities and differences in the data
- This new perspective combines the original “characteristics” in order to best summarize the data

Example. Data that can be collected about teaching classrooms:

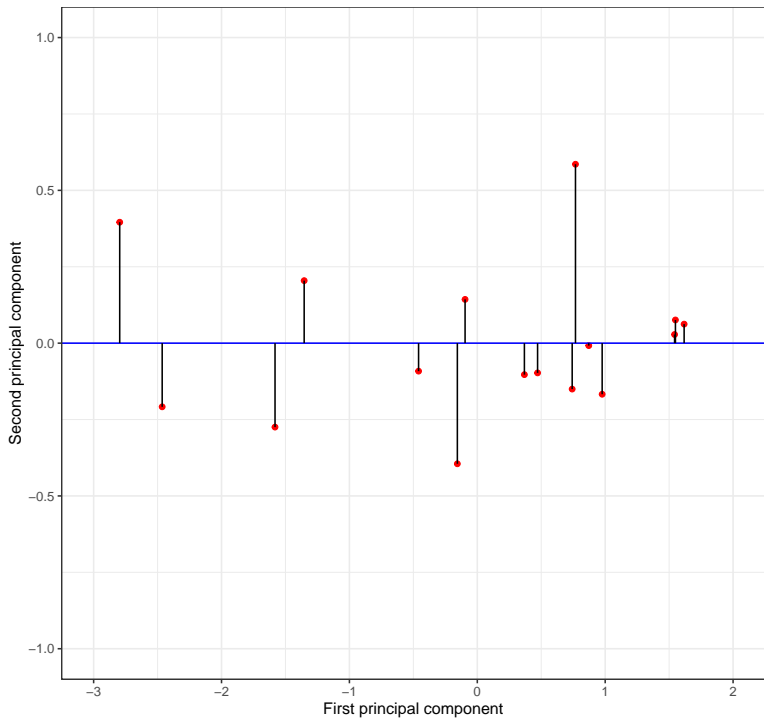
- The area
- The number of students who can seat in
- The number of blackboards
- The number of desktop computers

*Which are useful, useless, or redundant?*









Process:

① Standardize the range of the variables:

- Prevents large differences in the range of the variables
- Ensures all variables “contribute equally”
- Compute

$$Z_{ij} = \frac{X_{ij} - \bar{X}_i}{\sigma_{X_i}}$$

② Determine the “relationship” between the variables:

- Find correlations between the variables
- Construct the covariance matrix over all the variables

③ Identify the principal components:

- Compute the eigenvalues and eigenvectors of the covariance matrix
- Reorder the eigenvalues in non-increasing order

*The stability of a method defines how it “reacts” to small perturbations*

Since  $M = PDP^{-1}$  we have  $D = P^{-1}MP$ , and for a small perturbation  $\delta M$  we get

$$D + \delta D = P^{-1}(M + \delta M)P.$$

This yields  $\delta D = P^{-1}\delta MP$ , which in term of norms translates as

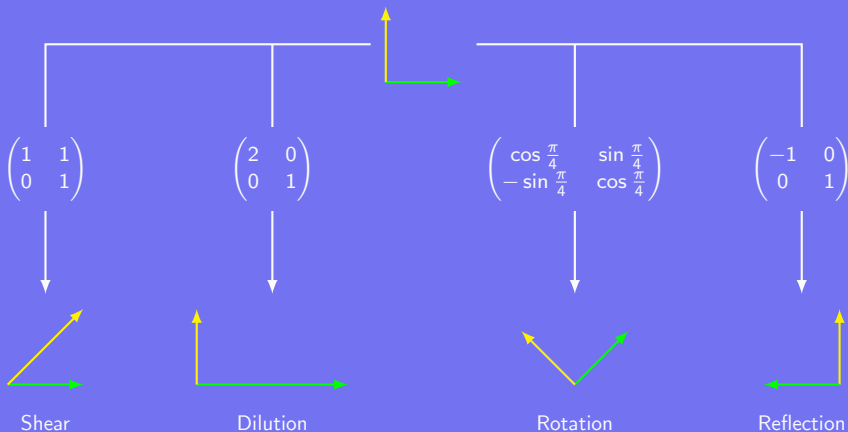
$$\|\delta D\| \leq \|P^{-1}\| \|P\| \|\delta M\|, \quad (4.1)$$

where the  $p$ -norm of an  $m \times n$  matrix  $A = (a_{i,j})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$  is defined as

$$\|A\|_p = \left( \sum_{i=1}^m \sum_{j=1}^n |a_{i,j}|^p \right)^{\frac{1}{p}}.$$

Equation (4.1) means that a perturbation  $\|\delta M\|$  might be magnified by a factor as large as  $\|P^{-1}\| \|P\|$ .

*For the sake of simplicity we consider the dimension 2 case*



Let  $\{v_1, v_2\}$  be an orthonormal basis, and  $M$  be the matrix of a linear transformation. Then for two unit vectors  $u_1$  and  $u_2$  we obtain

$$Mv_1 = u_1\sigma_1 \quad \text{and} \quad Mv_2 = u_2\sigma_2,$$

with  $\sigma_1$  and  $\sigma_2$  in  $\mathbb{R}$ . For a vector  $x = (x \cdot v_1)v_1 + (x \cdot v_2)v_2$ , we have

$$\begin{aligned} Mx &= (x \cdot v_1)Mv_1 + (x \cdot v_2)Mv_2 \\ &= (x \cdot v_1)u_1\sigma_1 + (x \cdot v_2)u_2\sigma_2. \end{aligned}$$

Recalling that  $x \cdot v_1 = x^\top v_1 = v_1^\top x$ , we get  $Mx = u_1\sigma_1 v_1^\top x + u_2\sigma_2 v_2^\top x$ , which yields  $M = u_1\sigma_1 v_1^\top + u_2\sigma_2 v_2^\top$ . In term of matrices this translates into

$$M = \begin{pmatrix} u_1 & u_2 \end{pmatrix} \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{pmatrix} \begin{pmatrix} v_1^\top \\ v_2^\top \end{pmatrix}.$$

More generally, if  $M$  is an  $m \times n$  real matrix we can write  $M = U\Sigma V^\top$ , where both  $U = (u_1 \cdots u_m)$  and  $V = (v_1 \cdots v_n)$  are rotation matrices, and  $\Sigma$  is diagonal.

Size of the matrices:

- $M$ :  $m \times n$
- $U$ :  $m \times m$
- $\Sigma$ :  $m \times n$
- $V$ :  $n \times n$

The elements on the diagonal of  $\Sigma$  are called *singular values*, the columns of  $U$  *left singular vectors*, and the rows of  $V^\top$  *right singular vectors*.

If  $U\Sigma V^\top$  is an SVD for an  $m \times n$  matrix  $X$  of rank  $r$ :

- $\Sigma$  has exactly  $r$  strictly positive elements which are the square roots of the  $r$  eigenvalues of  $X^\top X$ , with corresponding multiplicities
- The columns of  $V$  are eigenvectors of  $X^\top X$
- The columns of  $U$  are eigenvectors of  $XX^\top$

General remarks on the SVD:

- SVD is not unique and exists for any matrix, whether invertible or not
- Singular values can be re-ordered as long as their corresponding singular vectors are re-arranged accordingly
- If  $U\Sigma V^\top$  is an SVD for  $X^\top$ , then  $V\Sigma^\top U^\top$  is an SVD for  $X$

As  $U$  and  $V$  are rotation matrices, they are orthogonal, i.e.  $U^\top U = I_m$  and  $V^\top V = I_n$ . In particular, referring to slide 4.137, we see that in term of stability a small perturbation gets increased by a factor:

- $\|P^{-1}\| \|P\|$ , for eigen decomposition
- $\|U^{-1}\| \|(V^\top)^{-1}\| = \|U^\top\| \|V\| = 1$ , for SVD

Any  $X \in \mathcal{M}_{m,n}(\mathbb{K})$ , with linearly independent columns, can be written  $X = QR$  where:

- $Q \in \mathcal{M}_{m,n}(\mathbb{K})$  is orthogonal
- $R \in \mathcal{M}_{n,n}(\mathbb{K})$  is upper triangular

Remark. If  $X$  is invertible, then so is  $R$ .

Various approaches can be applied to compute the QR decomposition of  $X$ . The most famous one, Gram-Schmidt is unfortunately unstable. The two other most common options are based on Givens rotations and on Householder reflections. The former is slower but simpler to parallelize.

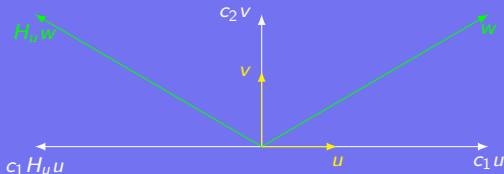
Given a vector  $u$ , a *Householder reflection* is a linear transformation  $H_u = I - \frac{2uu^\top}{u^\top u}$ . For instance in an orthonormal basis  $\{u, v\}$  we can write  $w = c_1 u + c_2 v$ , with  $c_1 = \frac{\langle w, u \rangle}{\|u\|_2^2}$  and  $c_2 = \frac{\langle w, v \rangle}{\|v\|_2^2}$ .



Applying  $H_u$  to  $w$  we get

$$\begin{aligned} H_u w &= \left( I - \frac{2uu^\top}{u^\top u} \right) (c_1 u + c_2 v) = c_1 u + c_2 v - 2 \frac{uu^\top}{u^\top u} (c_1 u + c_2 v) \\ &= c_1 u + c_2 v - 2c_1 = -c_1 u + c_2 v - 2c_2 \frac{u}{u^\top u} \langle u, v \rangle \\ &= -c_1 u + c_2 v. \end{aligned}$$

This example can be visually represented as a reflection about the plane orthogonal to  $u$  and  $v$ .



Householder reflections can be used to obtain the QR decomposition of a matrix  $X$ . More specifically, *Householder reflection theorem* states that for two vectors  $x$  and  $y$  with similar norm, there exists an orthogonal matrix  $Q$  such that  $y = Qx$ . This provides a method for iteratively constructing  $R$  and  $Q$ .

Idea on how to apply the above discussion:

- Determine an orthogonal matrix  $Q_1$  such that  $Q_1 c_1 = (\gamma_1, 0, \dots, 0)^T$ , where  $\gamma_1$  has similar norm as  $c_1$ . At this stage we have

$$R_1 = \left( \begin{array}{c|cccccc} \gamma_1 & * & \dots & \dots & * & \\ \hline 0 & & & & & \\ \vdots & & & & & \\ \vdots & & & & & \\ \vdots & & & & & \\ 0 & & & & & \end{array} \right) \begin{array}{c} \\ \\ \\ \\ \\ X_1 \end{array}.$$

- Determine an orthogonal matrix  $\tilde{Q}_2$  such that  $\tilde{Q}_2 \tilde{c}_1 = (\gamma_2, 0, \dots, 0)^\top$ . Then define  $Q_2 = \text{diag}(\text{Id}_1, \tilde{Q}_2)$  and compute

$$R_2 = Q_1 R_1 = \left( \begin{array}{cc|cccccc} \gamma_1 & * & \dots & \dots & \dots & \dots & * \\ 0 & \gamma_2 & & & & & * & \dots & \dots & * \\ \hline 0 & 0 & & & & & & & & \\ \vdots & \vdots & & & & & & & & \\ \vdots & \vdots & & & & & & & & \\ \vdots & \vdots & & & & & & & & \\ 0 & 0 & & & & & & & & \end{array} \right) \begin{array}{c} \\ \\ X_2 \\ \\ \\ \\ \end{array}.$$

- Repeat the above process  $t = \min(m-1, n)$  times and obtain  $R = Q_t Q_{t-1} \cdots Q_2 Q_1 X$ . By orthogonality  $Q^{-1} = Q^\top$ , yielding

$$X = Q_1^\top \cdots Q_t^\top R = QR.$$

Application to the SVD of  $X$ :

- Write  $X = QR$
- Obtain the SVD for

- Perform SVD on  $R = U_1 \Sigma V^\top$

$$X = Q U_1 \Sigma V^\top = U \Sigma V^\top$$

Let  $X$  be an  $m \times n$  matrix representing a dataset.



A diagram showing a large blue 'X' representing a matrix. A horizontal curly brace below the 'X' is labeled 'n attributes'. A vertical curly brace to the right of the 'X' is labeled 'm cases'.

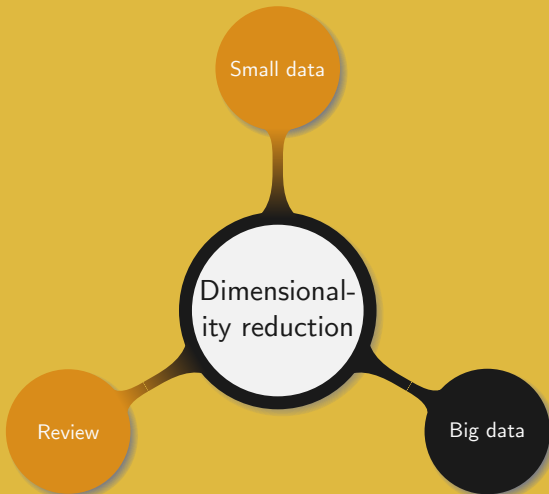
PCA requires to determine the eigenvalues of the covariance matrix  $C = X^\top X$  (slide 4.136). Now observe that using the SVD decomposition  $X = U\Sigma V^\top$ ,

$$\begin{aligned} X^\top X &= (U\Sigma V^\top)^\top (U\Sigma V^\top) \\ &= V\Sigma^\top U^\top U\Sigma V^\top \\ &= V\Sigma^2 V^\top. \end{aligned}$$

It suffices to compute the singular values of  $X^\top X$  and then  $U = XV\Sigma^{-1}$ .

Given a dataset one wants to:

- Retain a maximum of information in a minimum amount of space
- Run PCA using:
  - The covariance matrix:
    - Might be a bit faster but is unstable
    - Eigenvalues correspond to the variances of the principal components
  - SVD:
    - Might be a bit slower but stable
    - The square root of the singular values corresponds to the variances of the principal components
- Only keep the  $k$  largest principal components by truncating the matrices



An  $m \times n$  matrix  $X$  can be:

- Dense
- Sparse
- Square
- Tall and skinny
- Short and fat

Ways to represent a matrix over many nodes:

- By row or column
- By elements
- By blocks

Remarks.

- The matrix does not fit anymore in memory
- The shape and representation of the matrix impacts the speed

Setup:

- $A$  does not fit in memory
- $A$  has many more rows than columns ( $m \gg n$ )
- $n^2$  fits in memory on a single machine

When running PCA:

- Complexity of a full SVD:  $\mathcal{O}(mn^2)$
- We only need the  $k$  most significant singular values and vectors
- The work needed is  $\mathcal{O}(mk^2)$

*Can we take advantage of the shape of  $X$  to be faster?*



Recalling the relation between PCA and SVD,

$$C = X^T X = V \Sigma^2 V^T, \quad (4.2)$$

we see that  $C$  can fit in memory since it has dimension  $n \times n$  and by assumption a single machine has  $n^2$  memory. Thus we can compute  $X^T X$ , without any dependence on  $m$ . Then using equation 4.2 we can find the eigenvalues of  $C$  and retrieve  $V$  and  $\Sigma$ . The main challenge is to ensure the singular values of  $C$  are not significantly altered in the process.

Setup preparation:

- Store matrices row-by-row on disk
- Ensure all elements are in  $[-1, 1]$ , i.e. divided by the largest element

Simple MapReduce approach to compute  $X^T X$ :

- Mapper:
  - For all pairs  $(x_{ij}, x_{ik})$  on row  $i$ , return the *(key, value)* pair  $((c_j, c_k), x_{ij}x_{ik})$ ,  
where  $c_j$  and  $c_k$  correspond to columns  $j$  and  $k$ , respectively.
- Reducer: consider all the *(key, value)* pairs  $((c_i, c_j), \langle v_1, \dots, v_R \rangle)$ , where each of the  $v_k$  corresponds to a product of elements in  $X$  and  $R$  is the number of non-zero products, and compute  $\sum_{i=1}^R v_i$ .

Considerations on the complexity:

- Communication: *shuffle size*  $\mathcal{O}(mL^2)$ , with  $L$  the maximum number of non-zero elements on a row
- Overload on a single machine: *reduce-key complexity*  $\mathcal{O}(m)$

Measuring the distance between several documents:

- Count the number of occurrence for all the words
- Compare the scores
- If documents feature many common words, then they are similar

Limitation:

- Take three documents  $d_1$ ,  $d_2$ , and  $d_3$ , where  $d_1$ ,  $d_2$  are long and  $d_3$  is an excerpt of  $d_1$
- What is likely to happen?

For two vectors  $d_1$  and  $d_2$  we define their *cosine similarity* as

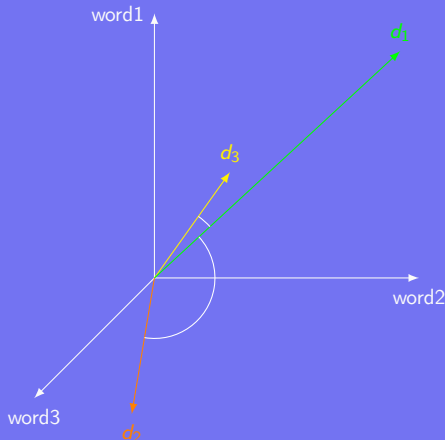
$$\cos(d_1, d_2) = \frac{d_1 \cdot d_2}{\|d_1\| \|d_2\|}.$$

Basic idea:

- The closer two documents, the smaller the angle
- The smaller the angle, the larger the cosine

More refined strategy:

- Classify words by meaning
- Consider the cosine similarity based on word semantic



We fix  $\gamma$ , a parameter that can be adjusted when running MapReduce.

MapReduce using cosine similarity to compute  $X^T X$ :

- Mapper:

- For all pairs  $(x_{ij}, x_{ik})$  on row  $i$ , return the  $(key, value)$  pair

$$((c_j, c_k), x_{ij}x_{ik}), \text{ with probability } \min\left(1, \frac{\gamma}{\|c_j\| \|c_k\|}\right),$$

where  $c_j$  and  $c_k$  correspond to columns  $j$  and  $k$ , respectively.

- Reducer: consider all the  $(key, value)$  pairs  $((c_i, c_j), \langle v_1, \dots, v_R \rangle)$ , where each of the  $v_k$  corresponds to a product of elements in  $X$  and  $R$  is the number of non-zero products and proceed as follows.
  - If  $\frac{\gamma}{\|c_j\| \|c_k\|} > 1$ , then return  $\frac{1}{\|c_j\| \|c_k\|} \sum_{i=1}^R v_i$
  - Otherwise, return  $\frac{1}{\gamma} \sum_{i=1}^R v_i$

What the reducer returns is in fact not  $X^\top X$ , but a matrix  $X'$  which contains the cosine similarities between the columns of  $X$ . Defining  $\bar{v}_i$  to be  $v_i$ , if a  $(key, value)$  pair is returned and 0 otherwise, this can be seen by looking at the expectation of an output

$$E \left( \frac{1}{\gamma} \sum_{i=1}^R \bar{v}_i \right) = \frac{1}{\gamma} P(\bar{v}_i = v_i) \sum_{i=1}^R v_i = \frac{1}{\|v_j\| \|v_k\|} \sum_{i=1}^R v_i.$$

To recover  $X^\top X$  from the matrix  $X'$ , it suffices to define a diagonal matrix  $D$  whose elements  $d_{ii}$  are exactly  $\|c_i\|$ , and then compute  $X^\top X = DX'D$ .

Using Latala's theorem it is possible to prove that when using cosine similarities to sample columns, singular values are preserved with "high enough" probability.

## Remarks.

- Since  $\|c_i\|$  is used in the mapper it means the norm of all the columns must be pre-computed. This requires an all-to-all communication.
- The parameter  $\gamma$  can be adjusted depending on what is expected:
  - Preserve similar entries in  $X^\top X$ :  $\gamma = \Omega\left(\frac{\log n}{s}\right)$
  - Preserve singular values of  $X^\top X$ :  $\gamma = \Omega\left(\frac{n}{\varepsilon^2}\right)$

Complexity, with  $h$  the smallest non-zero value after normalization:

- Shuffle size:  $\mathcal{O}(nL\gamma/h^2)$
- Reduce key complexity:  $\mathcal{O}(\gamma/h^2)$

As explained in slide 4.145 first decomposing  $X$  into  $X = QR$  can speed up the SVD of  $X$ . The basic idea related to Householder reflection theorem can in fact be extended into a “tilted QR decomposition” algorithm, which is applicable when the matrix  $X$  does not fit in memory.

In this algorithm, the tiles correspond to matrix blocks which can be stored and processed on different nodes.

Represent  $X$  in tall and skinny blocks, then start process:

- Obtain the QR decomposition of block  $X_{k,k}$
- Adjust all the blocks  $X_{k,j}$  on the right of  $X_{k,k}$
- Adjust all the blocks  $X_{i,k}$  below  $X_{k,k}$  and the blocks  $X_{i,j}$  on their right
- Repeat for all blocks on the diagonal



### Brief summary:

- Data has many variables
- Apply PCA to find a better perspective
- Ignore low contributions, only keep the most “prominent” dimensions
- Apply SVD in order to compute PCA
- Run further tasks based on the PCA “approximation”

### Comments on PCA:

- It works well on small data as well as on tall and skinny big data
- It searches for a “best” solution

*How much better is the best solution compared to a random solution?*

*Construct a map preserving the pairwise distance between points*

Setup and goals:

- Map  $m$  points  $(u_i)_{1 \leq i \leq n}$  in  $\mathbb{R}^n$  into  $m$  points in  $\mathbb{R}^d$ , with  $d \ll n$
- Ensure that for all  $i, j$ ,

$$\|v_i\|_2 \approx \|u_i\|_2 \text{ and } \|v_i - v_j\|_2 \approx \|u_i - u_j\|_2 \quad (4.3)$$

*Johnson-Lindenstrauss lemma* states that after a random projection, with high probability the distance between any two points is “distorted” by a factor at most  $1 \pm \epsilon$ .

If  $X' = \frac{1}{\sqrt{d}}XR$ , where  $R \in \mathbb{R}^{n \times d}$  is random with independent identically distributed entries and zero mean, then  $X'$  is much smaller and preserves all pairwise distances in expectation.

Feeling behind the random projection idea:

- Project a random vector onto a fixed subspace:
  - Consider  $m$  points in  $\mathbb{R}^n$  and fix  $k$  coordinates uniformly at random
  - Two vectors differing by only few coordinates can see their distance totally changed after projection
- Project a fixed vector onto a random subspace:
  - Consider  $m$  points in  $\mathbb{R}^n$  and project them in a  $k$ -dimensional subspace
  - Two vectors differing by only few coordinates will “spread out” all coordinates and prevent missing out on “important” coordinates

Generation of the matrix  $R = (r_{i,j})$ :

- Different  $r_{i,j}$  can impact the variance and the error tail bounds
- The  $r_{i,j}$  are independent identically distributed with zero mean
- The  $r_{i,j}$  are often select following a symmetric distribution about zero with unit variance

## Remarks.

- A random projection is likely not a projection in the algebraic sense:
  - Most often:  $R^2 \neq R$
  - Eigenvalues of  $R$  are not necessarily in  $\{0, 1\}$
  - It is  $\epsilon$ -close to a projection
- Requirement (4.3) does not extend to other norms
- The norm of a projected vector is  $\sqrt{\frac{d}{n}}$  in expectation, with an error exponentially small in  $d$
- Time complexity:  $\mathcal{O}(mnd)$
- The projection can be made very sparse with little loss of accuracy and a major speedup

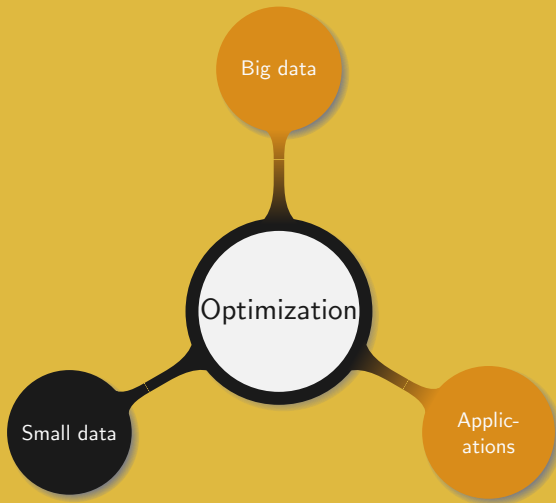
Preserving important structural properties of the data:

- Tall and skinny can be handle using PCA and SVD
- For other cases random projections are a good alternative
- PCA will ensure a good result, random projections might not
- Randomized PCA:
  - Randomly project  $X$  to get  $X'$
  - Perform a QR decomposition on  $X' = QR$
  - Compute  $B = Q^T X$
  - Do SVD on  $B = U_1 \Sigma V^T$
  - Get the SVD on  $X \approx QRQ^T X = Q (U_1 \Sigma V^T) = U \Sigma V^T$



## 5. Optimization

---



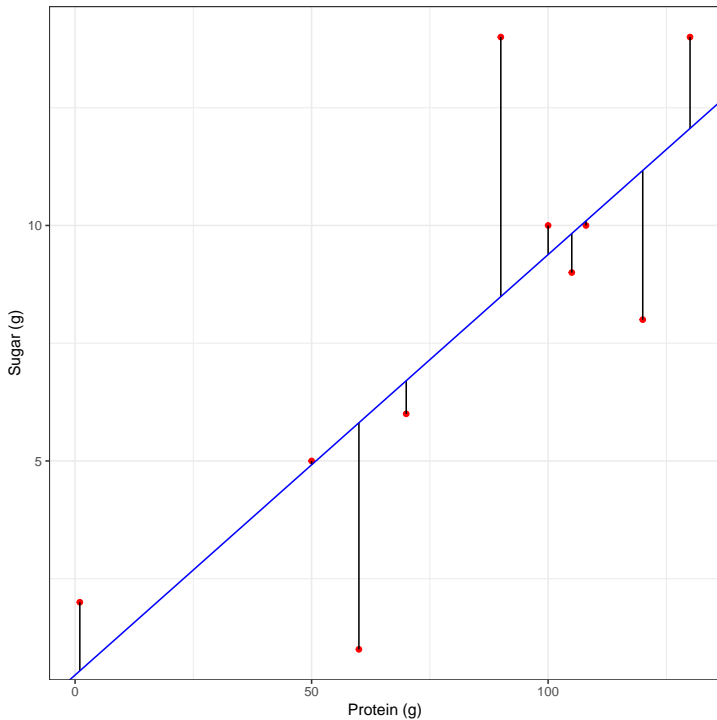


## Basics on regression:

- Find the relationship between:
  - A *dependent variable*, i.e. the outcome or response variable
  - *Independent variables*, i.e. predictors or explanatory variables
- Many types of regression exist, most common ones are linear, logistics, and polynomial
- The goal is to minimize the error of the prediction

## Error evaluation:

- Error is often measured using the *the sum of squares*, i.e. summing up the square of the error at each point
- The minimum of the sum of squares is called *least squares*
- The smaller the error the better the model



### Reminders on optimization:

- The goal is to maximize or minimize a function  $f$  depending in its input  $x$
- The function  $f$  is called *objective function* or *criterion*
- During the minimization process  $f$  is often referred to as *cost*, *loss*, or *error function*

### Remark on minimization with respect to least square:

- Whether positive or negative a large error is bad
- Squaring allows to penalize larger residuals more than smaller ones
- Error is often composed of *systematic* and *random* noises
- Minimizing the sum of squared errors is the same as minimizing the variance

Common examples of optimization problems:

- Chip design: ensure no tracks cross on a computer chip
- Timetable: given a list of students in each course, minimize the number of collisions
- Traveling salesman: given a list of cities, minimize the distance necessary to visit all of them

No free lunch theorem:

- There is no best solution to all search problems
- Algorithms performing better on certain problems, do worse on others
- Work is often needed to find the most suitable algorithm

*Description is usually done in term of minimization*

Basic setup for a function  $f(X)$ , where  $X = (x_1, \dots, x_n)$ :

- Iteratively create a sequence of  $X_i$
- At each step, determine the gradient in each direction
- Select a direction and keep going down
- Repeat until the gradient is 0 in all directions

Remark. The function  $f$  should be convex, i.e. for any  $x, y \in \mathbb{R}^d$ , and  $\alpha \in [0, 1]$ ,

$$f(\alpha x + (1 - \alpha)y) \leq \alpha f(x) + (1 - \alpha)f(y),$$

to ensure it has a global minimum, and the algorithm does not return a local one.

Example. For  $f(x) = 0.5x_1^2 + 0.2x_2^2 + 0.6x_3^2$ , we obtain

$$\nabla f(x) = (x_1, 0.4x_2, 1.2x_3).$$

Using a step of length 1, and starting with  $X_0 = (-2, 2, -2)$ , the steepest downhill direction is  $(-2, 0.8, -2.4)$ , yielding  $X_1 = (0, 1.2, 0.4)$ .

After a few iterations we find  $X_6 = (0, 0.0569, 0.0000256)$ .

Remarks.

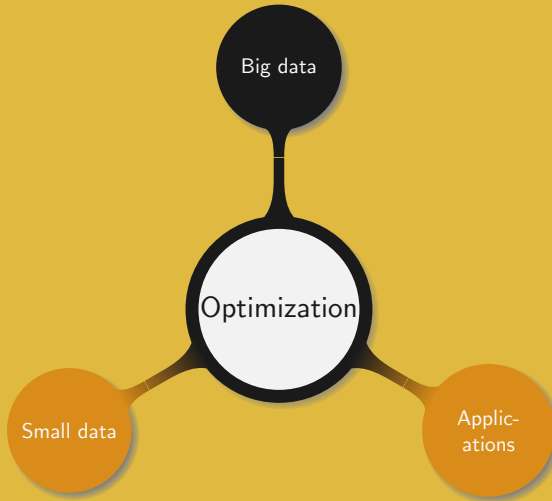
- Using a constant step could lead to either going very slowly or overstepping the minimum
- Computation can be speeded up by using Taylor expansion. This involves inverting the Hessian matrix of  $f$  which has a cost of  $\mathcal{O}(n^3)$ . However this method allows to keep a constant step of 1, simplifying other parts of the computation

## Gradient descent:

- Use the whole dataset
- Deterministic method
- Slow but fast to converge
- Yields an optimal solution
- Slow to escape local minima

## Stochastic gradient descent:

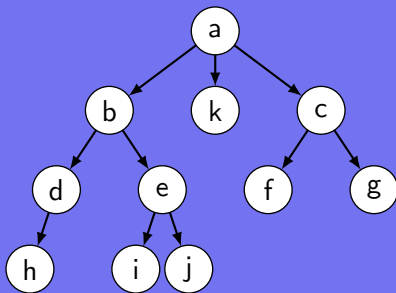
- Randomly select a sample
- Stochastic method
- Fast but slow to converge
- Yields a good enough solution
- Faster to escape local minima





When working with PRAM (slide 1.28):

- Dependencies between instructions are represented using a DAG
- Each instruction is represented as a node
- An edge  $(u, v)$  represents the dependency of  $u$  upon  $v$
- The root corresponds to the result of the computation



Computation finishes when the last processor completes its job:

- The amount of time when using one CPU is referred to as  $T_1$
- The amount of time when using  $p$  CPUs is referred to as  $T_p$
- The amount of time when using infinitely many CPUs is referred to as  $T_\infty$

Remarks.

- The *depth of an algorithm* is defined with respect to the last CPU to complete its tasks
- The *work of an algorithm* corresponds to the amount of time necessary to complete all tasks multiplied by the number of CPUs

*Does  $T_\infty$  tend to zero?*

Brent's theorem:

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

Meaning of the result:

- If work is evenly shared among all  $p$  CPUs then we get  $\frac{T_1}{p}$
- $T_\infty$  helps define how far we are from the ideal case

Remarks.

- Together  $T_1$  and  $T_\infty$  provide information on how well an algorithm performs on  $p$  CPUs
- Increasing the number of CPUs will never impact performance

*The work-depth model helps designing better parallel algorithms*

From a general point of view gradient descent is an optimization problem

$$\min_w F(w) = \sum_{i=1}^m F_i(w, x_i, y_i),$$

where  $x_i \in \mathbb{R}^n$ ,  $y_i \in \mathbb{R}$  is a “label”, and  $w$  is the parameter we expect to optimize over.

In essence gradient descent starts with a random initial  $w$  and has the goal of iteratively improving on it, by computing  $w_{k+1} = w_k - \alpha \nabla F(w_k)$ , for some small  $\alpha$ . In our case the objective function  $F$  corresponds to a loss function.

Remark. From a theoretical point of view, this works especially well when  $F$  is strongly convex, differentiable, and  $\nabla F$  is  $L$ -Lipschitz continuous. In such a case taking  $\alpha < \frac{1}{L}$  leads to an exponential convergence rate to a global minimum!

For the sum of squares loss function, we want to minimize

$$F(w) = \sum_{i=1}^m F_i(w, x_i, y_i) = \sum_{i=1}^m \|x_i^\top w - y_i\|_2^2,$$

with  $x_i$ ,  $y_i$ , and  $w$  as above.

Notes on our setup:

- $F$  is strongly convex and Lipschitz continuous
- $m$  corresponds to the data parallelism
- $n$  corresponds to the model parallelism

*How well does gradient descent scale up?*

*How to sum up  $n$  elements in parallel?*

Algorithm. (*Basic summation*)

---

**Input** :  $a$  an array with  $n$  elements

**Output**:  $s$  the sum over all the elements of  $a$

```
1  $s \leftarrow 0$  ;  
2 for  $i \leftarrow 1, \dots, n$  do  
3    $s \leftarrow s + a[i]$  ;  
4 end for  
5 return  $s$ 
```

---

Basic summation:

- Work:  $\mathcal{O}(n)$
- Depth:  $\mathcal{O}(n)$

How to achieve:

- Work:  $\mathcal{O}(n)$
- Depth:  $\mathcal{O}(\log n)$

## Complexity of computing

$$F(w) = \sum_{i=1}^m \|x_i^\top w - y_i\|_2^2$$

- Work:  $\mathcal{O}(mn)$
- Depth:  $\mathcal{O}(\log mn)$

## Complexity of computing

$$\nabla F(w) = 2 \sum_{i=1}^m x_i^\top (x_i^\top w - y_i)$$

- Work:  $\mathcal{O}(mn)$
- Depth:  $\mathcal{O}(\log mn)$

Performing a complete gradient descend:

- Error  $\frac{1}{\varepsilon}$  can be achieved after  $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$  iterations
- Total depth:  $\mathcal{O}\left(\log \frac{1}{\varepsilon} \log mn\right)$

*How suitable is gradient descent for parallelization and big data?*

*Instead of computing a full gradient, apply it to a randomly selected point*

Stochastic gradient descent (using notations from slide 5.178):

- Call  $s_k$  the index uniformly sampled at iteration  $k$
- Compute the sequence

$$w_{k+1} = w_k - \alpha F_{s_k}(w_k)$$

Notes on stochastic gradient descent:

- Error  $\varepsilon$  will be achieved after  $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$  iterations
- Work decreases linearly with the number of points considered
- The number of iterations does not increase linearly with the number of sample points



## Gradient descent:

- Per iteration:
  - Work:  $\mathcal{O}(mn)$
  - Depth:  $\mathcal{O}(\log mn)$
- Total:
  - Work:  $\mathcal{O}\left(mn \log \frac{1}{\varepsilon}\right)$
  - Depth:  $\mathcal{O}\left(\log \frac{1}{\varepsilon} \log mn\right)$

## Stochastic gradient descent:

- Per iteration:
  - Work:  $\mathcal{O}(n)$
  - Depth:  $\mathcal{O}(\log n)$
- Total:
  - Work:  $\mathcal{O}\left(\frac{n}{\varepsilon}\right)$
  - Depth:  $\mathcal{O}\left(\frac{\log n}{\varepsilon}\right)$

*Which is best, gradient descent or stochastic gradient descent?*

Setup in PRAM:

- Save  $w$  in a shared piece of the memory
- All CPUs have access to  $w$  and the whole dataset

Things which could go wrong:

- A model is read, transformed, and written in the memory but in the meantime the model has been updated by another CPU
- An updated model is overwritten by an older one

*How bad is this situation?*

Adding locks:

- Solves the race condition problem
- Only one CPU can access  $w$  at a time
- All the benefits from the parallelism get lost

*Are locks really needed?*

In stochastic gradient descent:

- $w = (w^{(1)}, \dots, w^{(n)})$  is a vector of dimension  $n$
- A different  $w^{(i)}$  will be *randomly* chosen for each CPU

*In a big data setup the probability of collision of low*

Hogwild! strategy for  $p$  processors:

- Proceed in parallel over all available CPUs
- Until the expected error condition is met:
  - Select a random index  $i$  from  $\{1, \dots, m\}$
  - Compute  $F_i(w)$  and  $\nabla F_i(w)$  for  $w$  concurrently in the shared memory
  - Update  $w^{(i)}$  with  $w^{(i)} - \alpha [\nabla F_i(w)]^{(i)}$

Remarks.

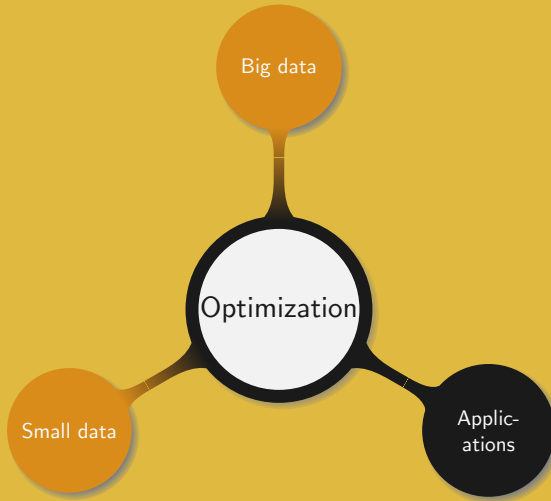
- No locks are used
- The speedup is nearly linear in term of CPUs
- In general the locking overhead should be avoided when processing big data

Looking back at gradient descent in PRAM:

- Stochastic had lower depth but was not easy to parallelize
- Batch could be easily parallelized
- Hogwild! renders stochastic almost *embarrassingly parallel*

Stochastic and batch on a distributed system:

- Communication at each iteration:
  - Batch:  $\mathcal{O}\left(\log \frac{1}{\epsilon}\right)$
  - Stochastic:  $\mathcal{O}\left(\frac{1}{\epsilon}\right)$
- Stochastic is often preferred on a single computer with many GPUs
- Batch is more common on distributed systems



Questions to consider first:

- What tool to use?
- Is batch or stochastic gradient descent most appropriate?
- How large is the data, i.e. can  $n$  and  $m$  fit in the memory?

In our setup we expect to:

- Use Spark
- Check how both gradient descents behave
- Work with  $n$  small enough to fit in memory but no restriction on  $m$

*How to store the data on the cluster?*

High-level idea for a Spark implementation:

- 1 Organise the data by row and store it in an RDD
- 2 Use a `map` transformation to generate a closure for each point
- 3 Use the `cache` action to ensure Spark keeps the RDD in memory
- 4 For each point  $p$ , apply a `map` to transform  $p$  into  $\nabla F_p(w)$
- 5 Apply a `reduce` to sum up all the  $\nabla F_p(w)$
- 6 Update  $w$  and repeat from step 4 until the expected error is reached



Optimizing our approach:

- How many times is  $w$  sent?
- Is  $w$  modified by the mappers?
- With respect to bandwidth and memory usage, how large is  $w$ ?
- How should  $w$  be shared among all the machines?

Basic analysis:

- Where is the bottleneck in our approach?
- How good or bad is the communication cost?

Reminders on stochastic gradient descent:

- Depth is much larger than for batch gradient descent
- Apply Hogwild! to speed up the process

Hogwild! on Spark:

- Broadcast needs to be completed to start the mappers
- All mappers and reducers must be done before broadcasting again
- Synchronisation barriers are essential to fault-tolerance

Minimizing an objective function in Spark:

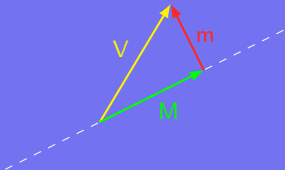
- Try random updates and accept any one lowering the objective
- Use mini-batches:
  - At each iteration select “many” samples, instead of one
  - Apply batch gradient descent to them

Common strategy for gradient descent when data is too large:

- Extract the principal components of the big dataset
- Apply gradient descent on the resulting approximation

Relating gradient descent to PCA:

- Find the axes which maximize the variance
- Find the direction for which the expectation of  $XX^T$  is maximized
- Find the direction which minimizes the residuals



For mean centered data:

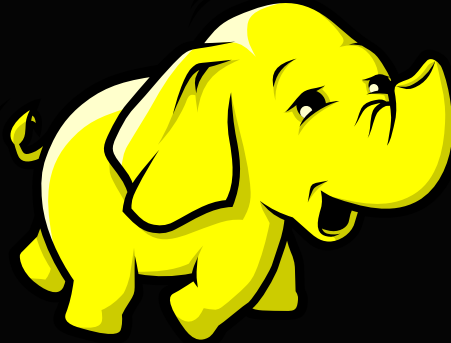
- Pythagorean theorem:  $V^2 = M^2 + m^2$
- PCA maximizes  $M$
- Gradient descent minimizes  $m$

*Given a mathematical model where an objective is represented as a linear function and some constraints are expressed as equalities or inequalities, maximize the objective while respecting the constraints.*

A gradient descent reading of linear programming:

- Any maximization problem can be rephrased into a minimization one
- The simplex method solves linear programming problems:
  - A basic solution is used to start with
  - A “local search” find a *pivot* to improve on the basic solution
  - Different choices of pivot lead to different convergence rates
  - Any pivot leads to an optimal solution
- Finding the best pivot in the simplex is equivalent to finding the best direction to improve on the objective





Thank you, enjoy the Summer break!

