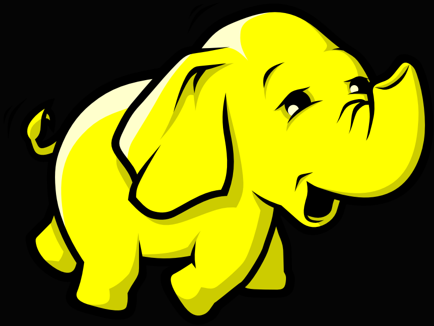


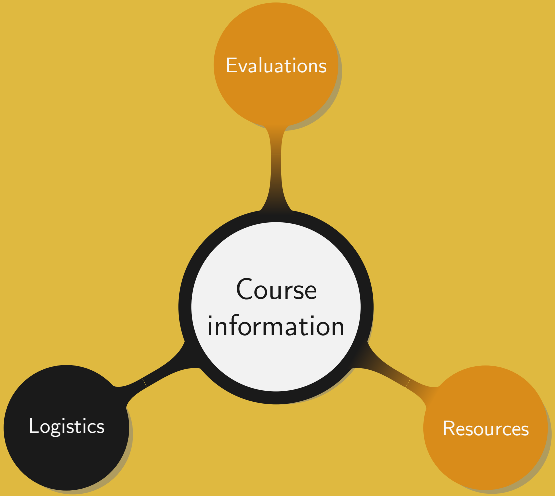
Methods and tools for big data

Manuel – Summer 2023

Table of contents

0. Course information

Chapter organisation



Evaluations

Course information

Logistics

Resources

Basic information

Teaching team:

- Instructor: Manuel (charlem@sjtu.edu.cn)
- Teaching assistant: Yuxuan (zyxdenny@sjtu.edu.cn)

Important rules:

- When contacting a TA for an important matter, `CC` the instructor
- Prepend [ECE472] to the subject, e.g. `Subject: [ECE472] Grades`
- Use SJTU jBox service to share large files (>2 MB)

Never send large files by email

Course schedule

Course arrangements:

- Lectures:
    - Monday 12:10 – 13:40
    - Wednesday 12:10 – 13:40
- Labs: Friday 18:20 – 20:40

Office hours:

- Anytime on Piazza
- On appointment

Course objectives

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

Course workflow

Learning strategy:

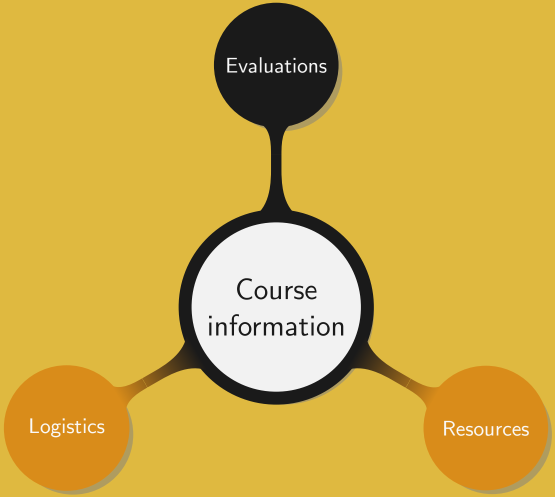
- Course side:
  1. Understand the new issues appearing as datasets grow
  2. Be able to setup a Hadoop cluster and use it
  3. Understand why traditional algorithms fail on big data
  4. Be able to implement advanced algorithms for big data
- Personal side:
  1. Derive algorithms for big data
  2. Use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

Course outcomes

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

Chapter organisation



Assignments

Homework:

- Total: 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

Grading policy

Grade weighting:

- Midterm exam: 25%
- Final exam: 25%
- 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 $\llbracket B, B+ \rrbracket$

Honor Code

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

Honor Code

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

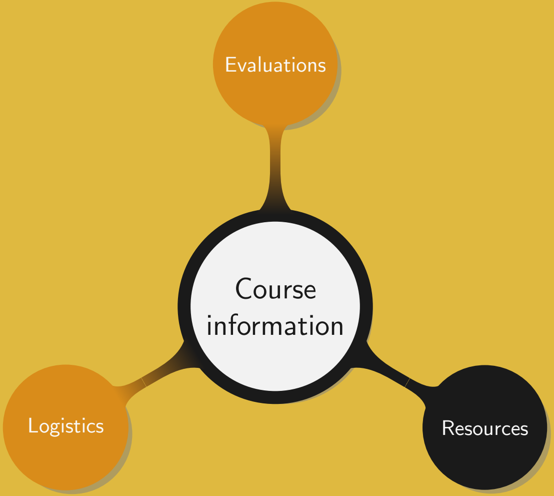
Special circumstances

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

Chapter organisation



Canvas

Information and documents available on the Canvas platform:

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

References

Useful places where to find information:

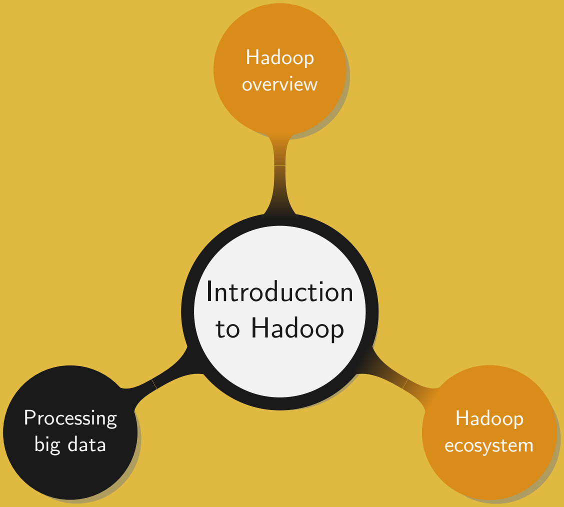
- *Hadoop the definitive guide*
- *Spark the definitive guide*
- *Machine learning, an algorithmic perspective*
- *Introduction to Data Mining*, by Tan et al..
- *Mining of Massive Datasets*, by Leskovec et al.. by White
- Search information online, i.e. $\{websites \setminus \{non\text{-}English\ websites\}\}$

Key points – Key points

- Work regularly, do not wait the last minute/day
- Respect the Honor Code
- Go beyond what is taught
- Do not learn, understand
- Keep in touch with us
- Advice and suggestions are always much appreciated

1. Introduction to Hadoop

Chapter organisation



Data analysis

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

Von Neumann bottleneck

Problem for a regular computer:

- Fast CPU
- Large memory
- Limited throughput

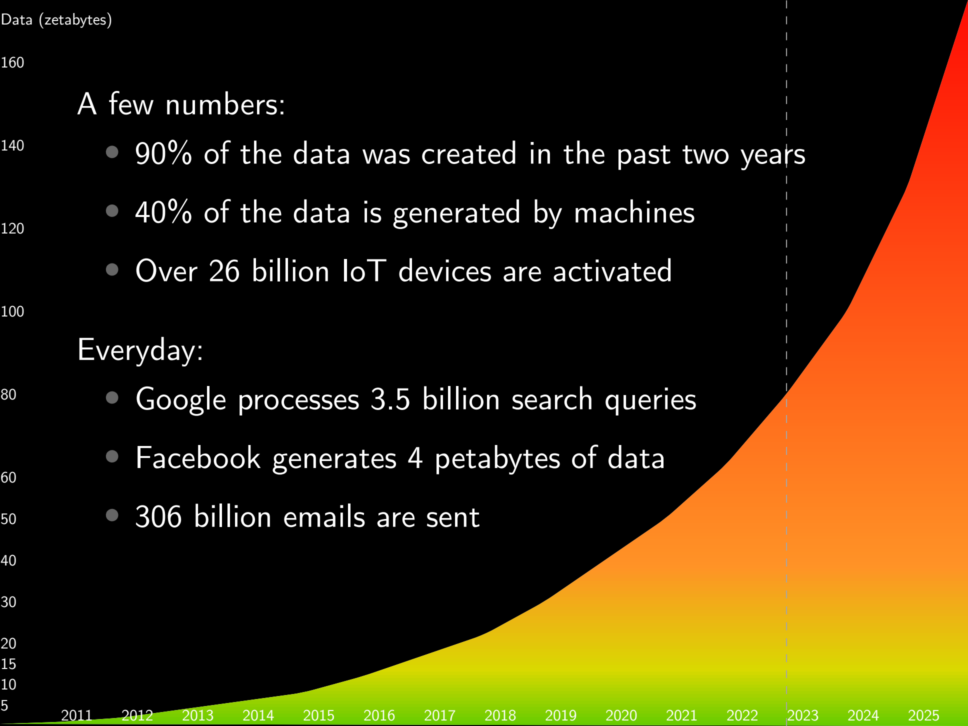
Mitigating the problem:

- 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!



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

Everyday:

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

How to store and process data

as it grows very big?

Storing data

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

Processing data

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

Sequential vs. parallel algorithms

Random Access Machine (RAM) model:

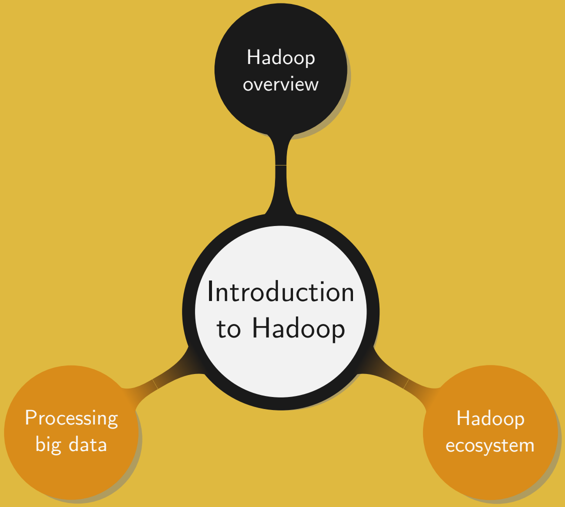
- A processor with a memory attached to it
- Each operation has a constant cost
- Runtime is proportional to the number of operations

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

Chapter organisation

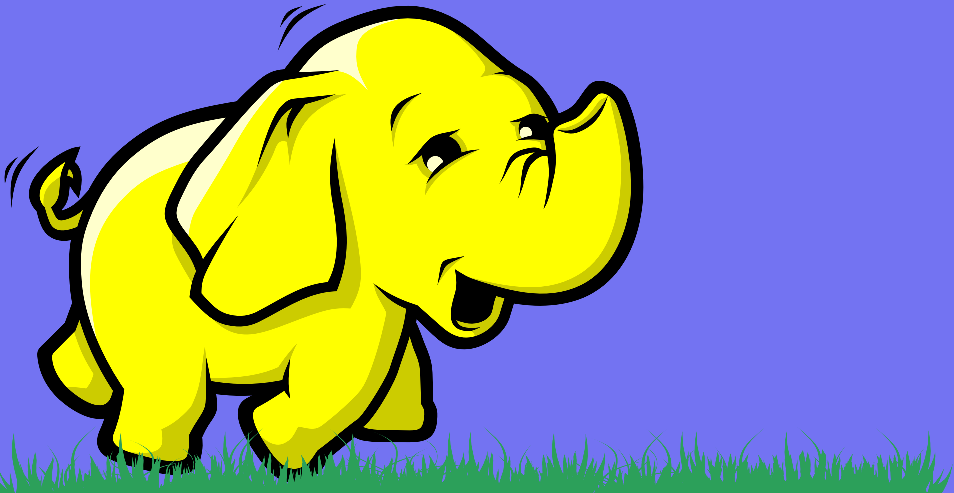


Introduction to Hadoop

Hadoop overview

Processing big data

Hadoop ecosystem



A short history

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!

A short history

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

A short history

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

Hadoop's goals

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 framework

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

HDFS overview

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

MapReduce overview

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 overview

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

YARN overview

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

MapReduce vs. YARN

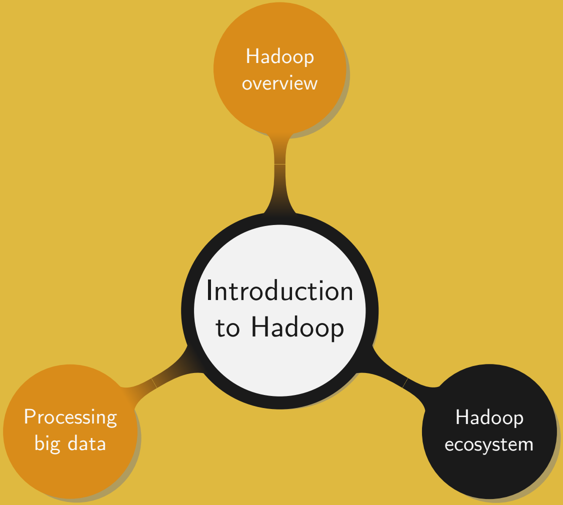
In Hadoop 1, MapReduce:

- 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

Chapter organisation



Mesos

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

Myriad

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

Spark

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

Drill

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

Spark vs. Drill

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

Flink

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

Tez

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

HBase

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

Hive, Spark SQL, and Presto

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

Serialization and storage

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

Management and monitoring

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

Analytics helpers

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

Data transfer

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

From batch to realtime processing

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

Realtime processing

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

AI and ML

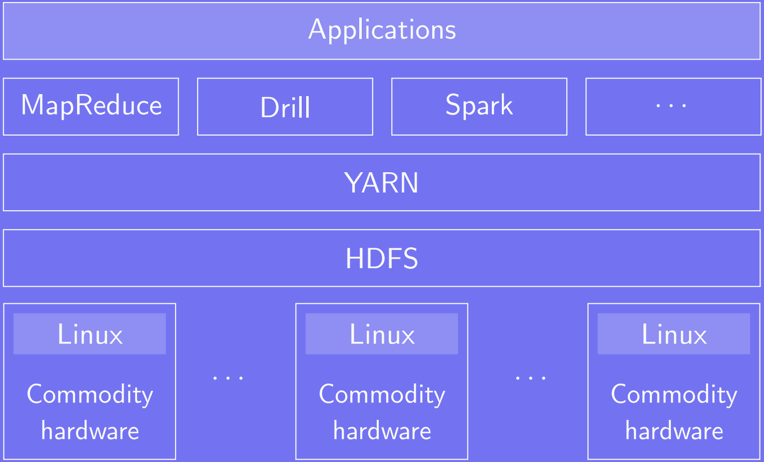
Ray:

- 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 is a tensor
- Create dataflow graphs where the edges are tensors and vertices mathematical operations
- Can run on Hadoop (TonY), Spark, and Ray

Hadoop ecosystem in VE472



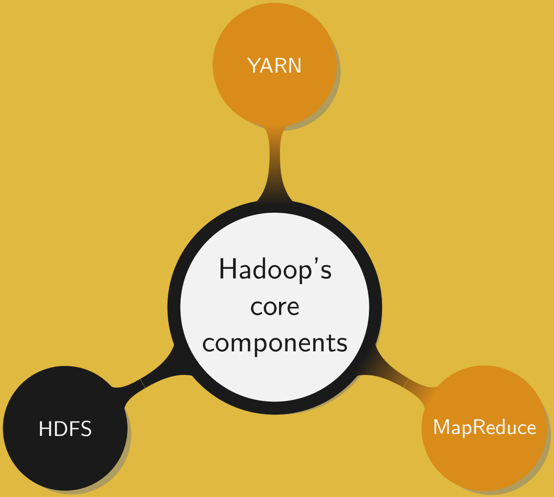
Refer to Hadoop ecosystem table for more details

Key points

- What is big data?
- List limitations of common solutions when analysing large data sets?
- What is Hadoop?
- What are Hadoop's goals?
- Briefly describe the Hadoop ecosystem

2. Hadoop's core components

Chapter organisation



Distributed filesystems

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

File blocks

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

Nodes

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

Finding a file

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?

Memory limitations

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?

HDFS federation

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

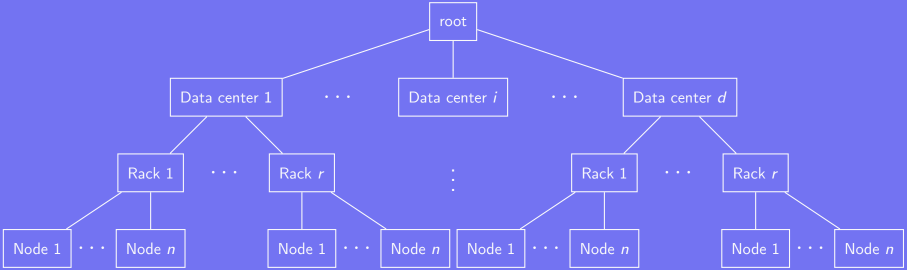
High availability

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

Cluster balancing

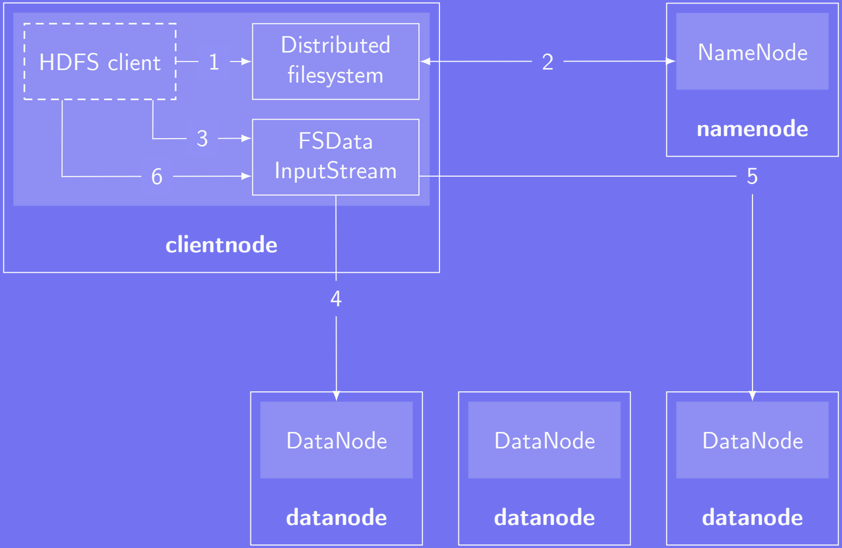
The goal is to evenly spread blocks across the whole cluster:



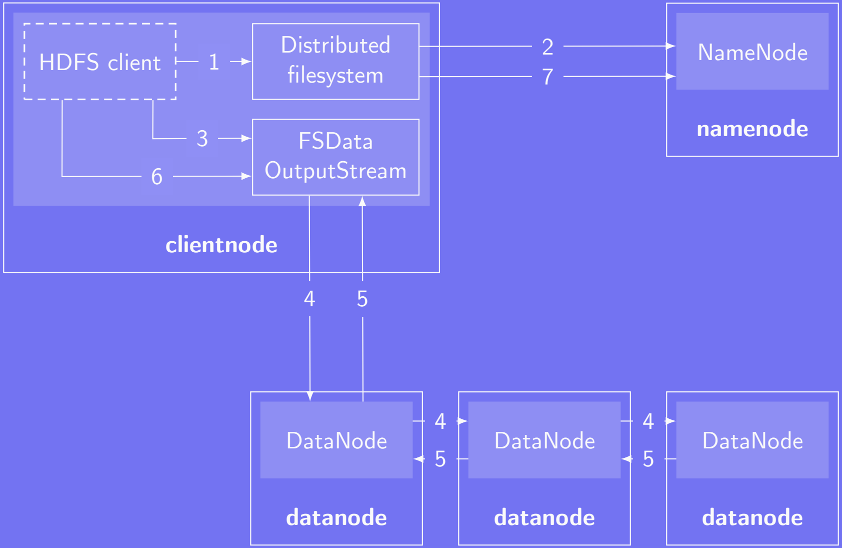
Replicas' location:

- 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

File read


HDFS client
1
Distributed filesystem
2
NameNode
namenode
3
FSData InputStream
6
5
clientnode
4
DataNode
datanode
DataNode
datanode
DataNode
datanode

File write


HDFS client
1
Distributed filesystem
2
7
NameNode
namenode
3
6
FSData OutputStream
clientnode
4
5
DataNode
datanode
4
5
DataNode
datanode
4
5
DataNode
datanode

Failing file write

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

Other filesystems

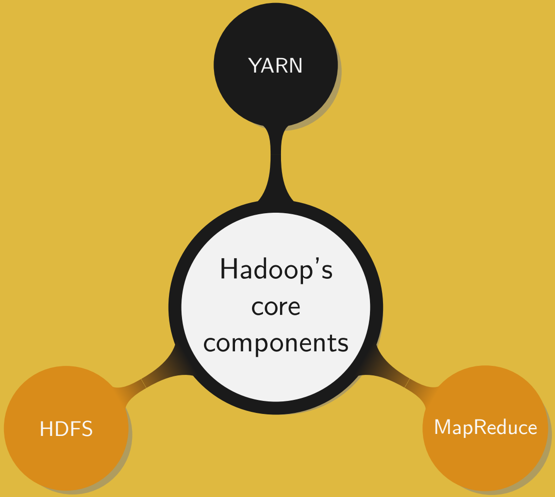
Alternative filesystems built on top of Hadoop abstract filesystem:

- 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

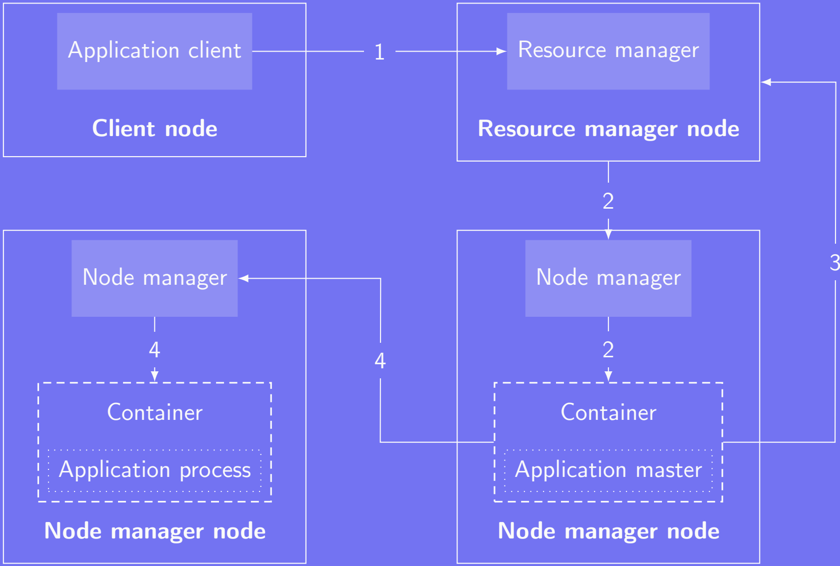
Interfaces to access HDFS data:

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

Chapter organisation



Running an application



Requesting resources

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

Categories of application

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

YARN schedulers

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

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

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

Fair scheduler

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

Delay scheduling

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

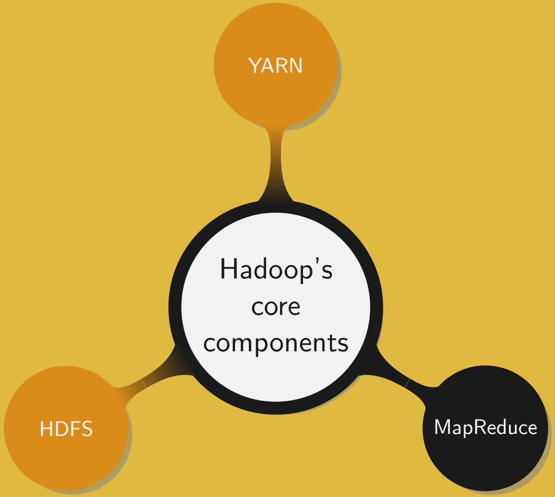
Dominant resource fairness

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

Chapter organisation



MapReduce job

Parties involved in a MapReduce job:

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

Job initialisation

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

Job startup

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

Job failure

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

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

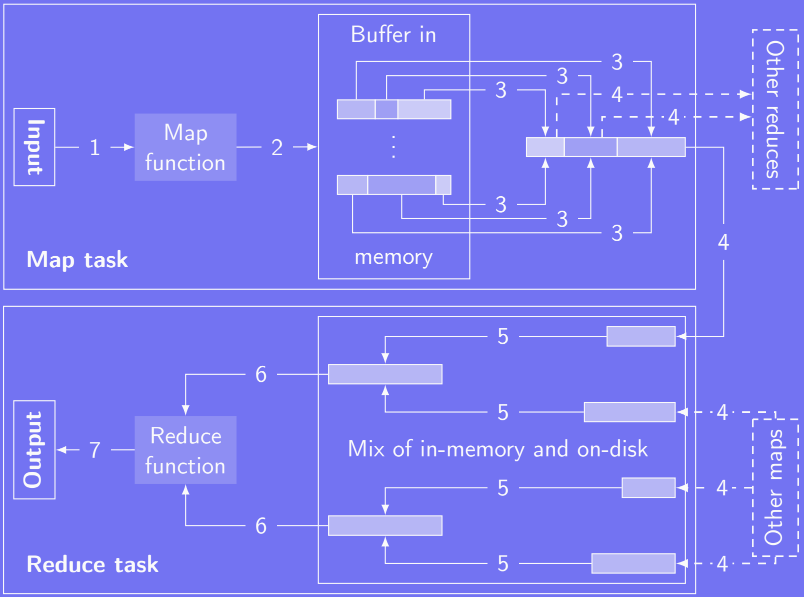
Failed tasks are rescheduled on a different nodemanager

Application master failure

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

Map and Reduce



Speeding up a MapReduce job

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:

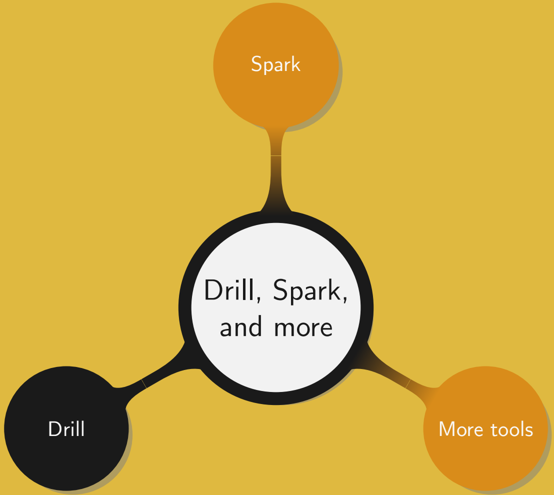
- A task is detected as much slower than average
- Re-run it on a different node
- Kill all the other duplicates as soon as one completes

Key points

- Explain the basic structure of Hadoop
- What are the main principles behind HDFS?
- How is YARN used by higher level applications?
- Describe the structure of a MapReduce job

3. Drill, Spark, and more

Chapter organisation



Drill job

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

Drill on YARN workflow

Running Drill as a YARN application:

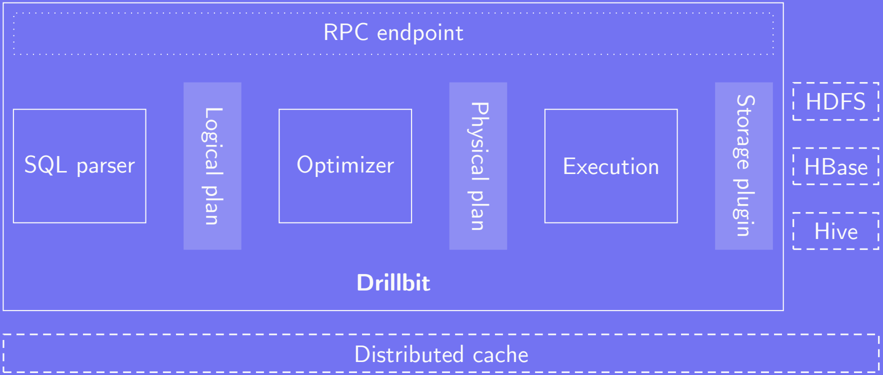
1. Start Drill on the client machine
2. Upload resources to the FS and request resources for the application master
3. Ask a node manager to prepare and start a container for the application master
4. The application master contacts the resource manager to obtain more containers

Drill on YARN workflow

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.

Drillbit



Foreman drillbit:

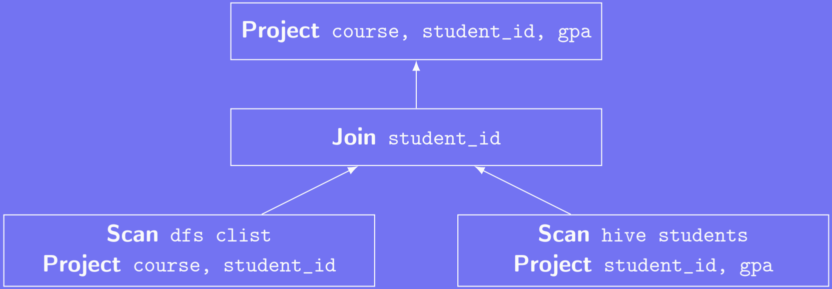
- Drillbit that receives the query
- It drives the entire query

Logical plan

Initial SQL query:

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

Logical plan:



Project course, student_id, gpa

Join student_id

Scan dfs clist
Project course, student_id

Scan hive students
Project student_id, gpa

Drill query execution

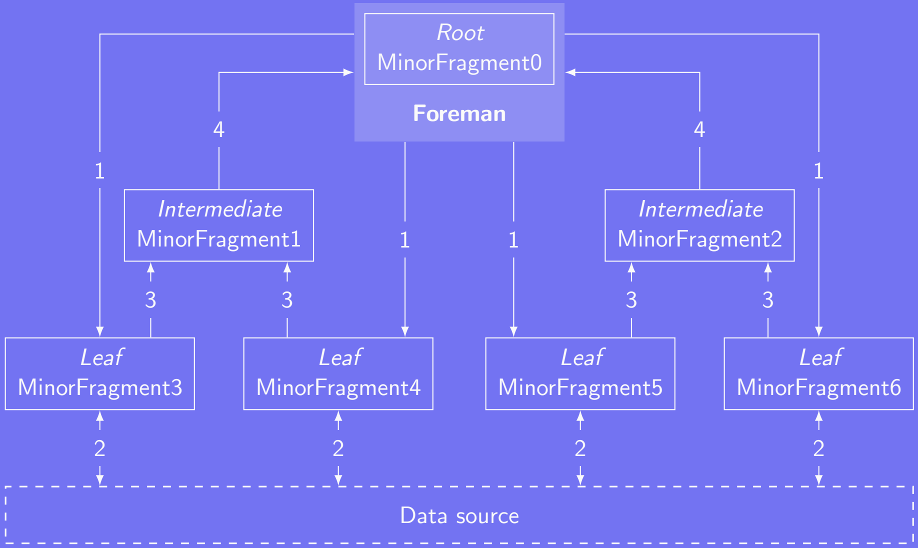
Major fragment:

- 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

Minor fragments execution tree



Drill's strength

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

Drill's strength

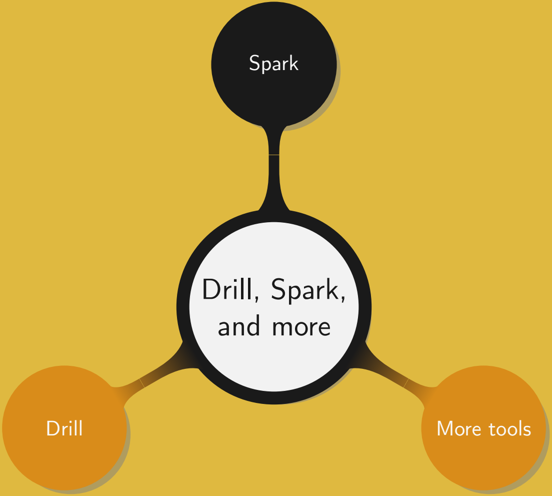
Optimistic query execution:

- 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

Chapter organisation



Spark job

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

Spark on YARN workflow

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

YARN client mode

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

YARN cluster mode

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

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

RDD operations

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

RDD persistence

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

Functions and variables

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

Shared variables

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

Running of a Spark job

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

Higher level abstractions

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

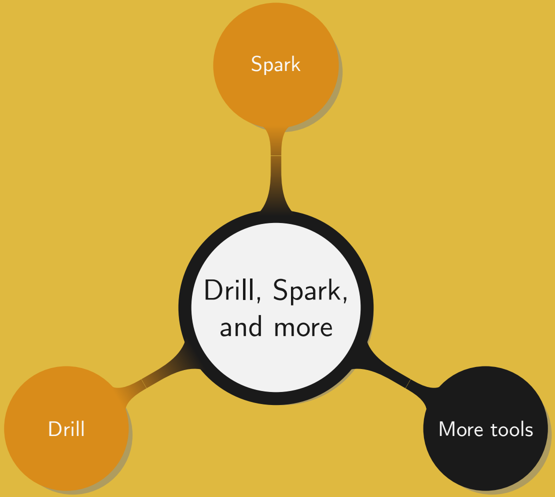
Spark's strength

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

Chapter organisation



Scaling up software

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

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

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

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

Kubernetes

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

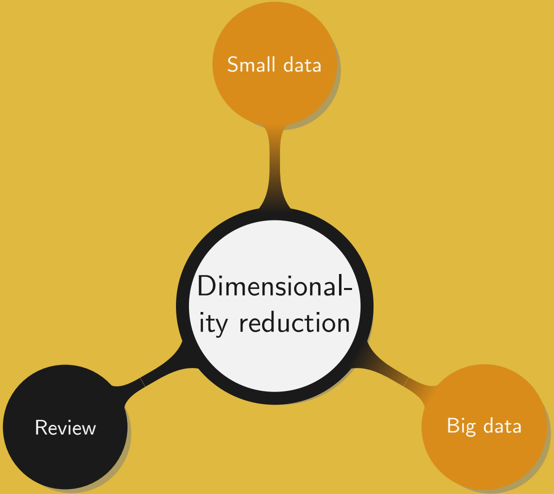
Kubernetes can be coupled to Hadoop or work independently

Key points

- When to use Drill?
- Explain the organisation of a drillbit
- When to use Spark?
- How does Spark interact with YARN?
- What are containerized solutions?

4. Dimensionality reduction

Chapter organisation



Statistical strategy

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

Statistical reminders

Given a dataset $X = [X_1, \cdots, X_n]$, 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 $\sigma = \sqrt{\frac{\left(\sum_{i=1}^{n} X_i - \bar{X}\right)^2}{n-1}}$
- *Variance* which also measures the spread and is given by σ^2

Those measures are one-dimensional

Statistical reminders

Given a dataset $[X_1, X_2] = \big[[X_{1,1}, \cdots, X_{1,n}], [X_{2,1}, \cdots, X_{2,n}]\big]$, *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 $X_1, \cdots, X_n$, their covariance matrix is given by

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

Algebraic reminders

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 $k \in \mathbb{N}$, $M^k = PD^kP^{-1}$

Algebraic reminders

The *characteristic polynomial* of M is defined as

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

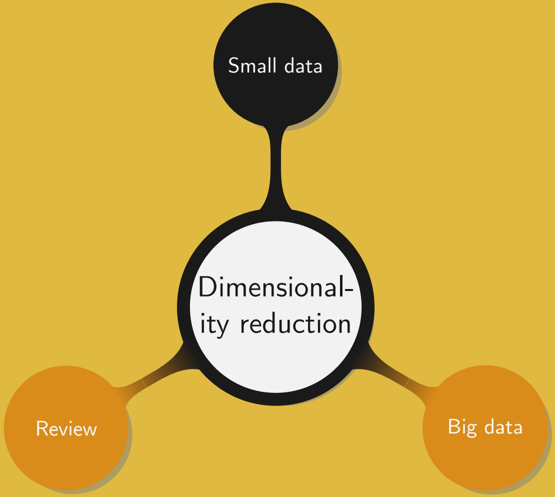
Properties of χ_M:

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

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*

Chapter organisation



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

Principal component analysis

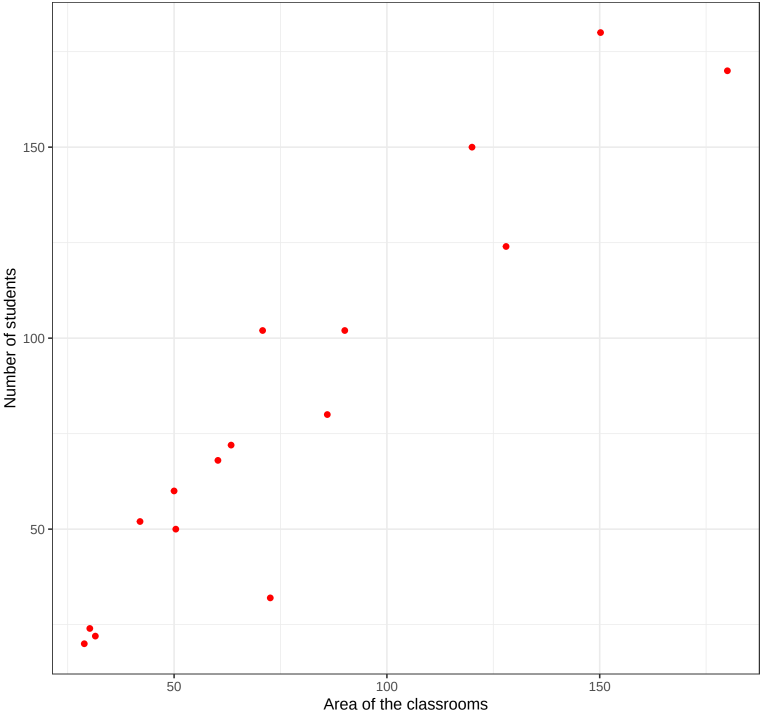
Basic idea behind Principal Component Analysis (PCA):

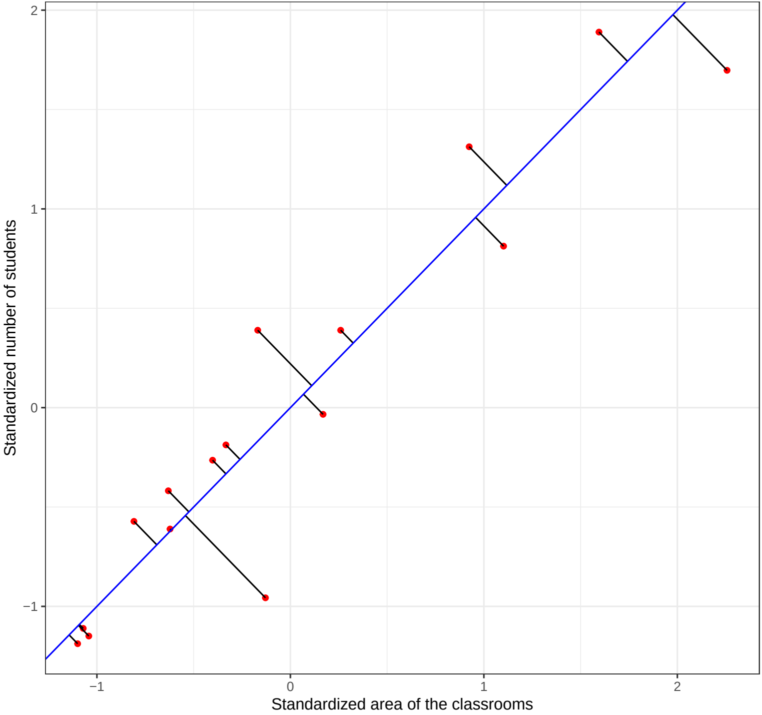
- Provides the best "perspective" that emphasises similarities and differences in the data
- This new perspective combines the original "characteristics" in order to best summarize the data

Example. Data that can be collected about teaching classrooms:

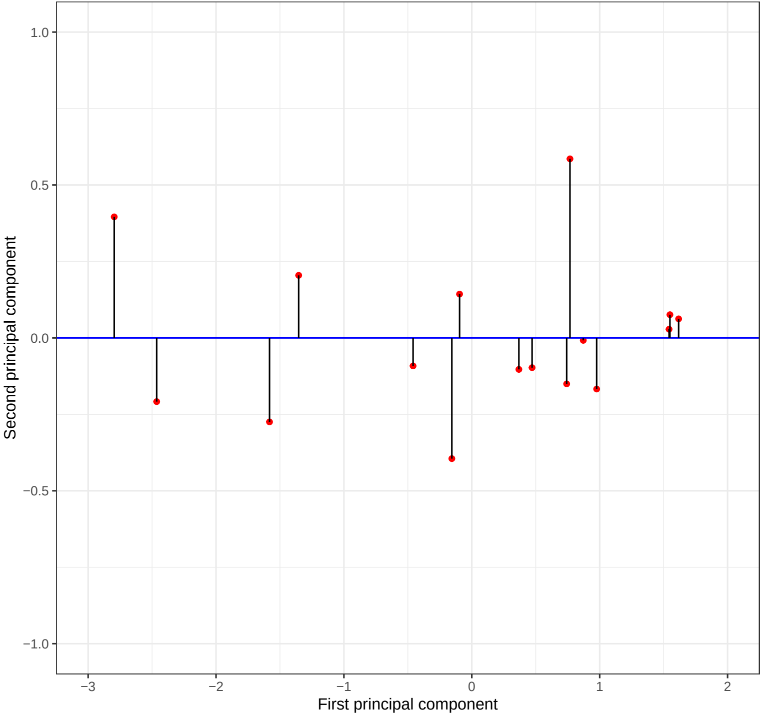
- The area
- The number of students who can seat in
- The number of blackboards
- The number of desktop computers

Which are useful, useless, or redundant?


150
100
50
Number of students
50
100
150
Area of the classrooms



Standardized number of students
2
1
0
−1
−1
0
1
2
Standardized area of the classrooms


1.0
0.5
0.0
−0.5
−1.0
Second principal component
−3
−2
−1
0
1
2
First principal component

Principal component analysis

Process:

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

$$Z_{i,j} = \frac{X_{i,j} - \bar{X}_i}{\sigma_{X_i}}$$

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

Numerical stability

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 δM 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

$$\|\delta D\| \leq \|P^{-1}\|\|P\|\|\delta M\|, \tag{4.1}$$

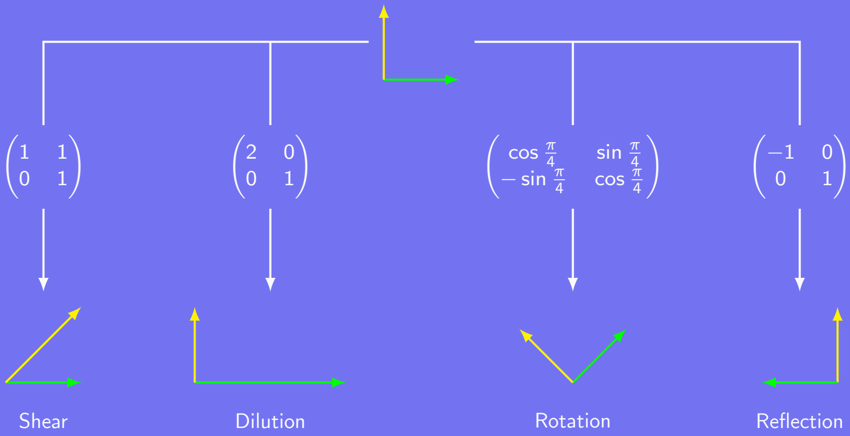
where the p-norm of an $m \times n$ matrix $A = (a_{i,j})_{\substack{1\leq i\leq m\\ 1\leq j\leq n}}$ is defined as

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

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

Linear transformations

For the sake of simplicity we consider the dimension 2 case



$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} \cos \frac{\pi}{4} & \sin \frac{\pi}{4} \\ -\sin \frac{\pi}{4} & \cos \frac{\pi}{4} \end{pmatrix}$$

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

Singular value decomposition

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

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

with σ_1 and σ_2 in $\mathbb{R}$. For a vector $x = \langle x, v_1\rangle v_1 + \langle x, v_2\rangle v_2$, we have

$$\begin{aligned} Mx &= \langle x, v_1\rangle Mv_1 + \langle x, v_2\rangle Mv_2 \\ &= \langle x, v_1\rangle u_1\sigma_1 + \langle x, v_2\rangle u_2\sigma_2. \end{aligned}$$

Recalling that $\langle x, v_1\rangle = x^\top v_1 = v_1^\top x$, 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

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

Singular value decomposition

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

Size of the matrices:

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

The elements on the diagonal of Σ 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:

- Σ 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 or $XX^\top$

Singular value decomposition

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. $U^\top U = \mathrm{I}_m$ and $V^\top V = \mathrm{I}_n$. 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

QR decomposition

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.

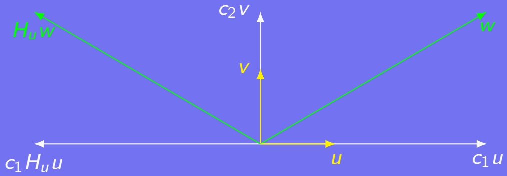
Given a vector u, a *Householder reflection* is a linear transformation $H_u = I - \frac{2uu^\top}{u^\top u}$. For instance in an orthonormal basis $\{u, v\}$ we can write $w = c_1 u + c_2 v$, with $c_1 = \frac{\langle w,u \rangle}{\|u\|_2^2}$ and $c_2 = \frac{\langle w,v \rangle}{\|v\|_2^2}$.

QR decomposition

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.



QR decomposition

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 $Q_1 c_1 = (\gamma_1, 0, \cdots, 0)^\top$, where γ_1 has similar norm as c_1. At this stage we have

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

SVD and QR decomposition

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

$$R_2 = Q_1 R_1 = \left(\begin{array}{cc|c} \gamma_1 & \multicolumn{2}{l}{* \cdots\cdots\cdots\cdots *} \\ 0 & \gamma_2 & * \cdots\cdots * \\ \hline 0 & 0 & \\ \vdots & \vdots & X_2 \\ 0 & 0 & \end{array}\right).$$

- 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:

1. Write $X = QR$
2. Perform SVD on $R = U_1 \Sigma V^\top$
3. Obtain the SVD for

$$X = QU_1 \Sigma V^\top = U \Sigma V^\top$$

PCA and SVD

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

$$\left.\underbrace{\qquad\qquad X \qquad\qquad}_{n \text{ attributes}}\right\} m \text{ cases}$$

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

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

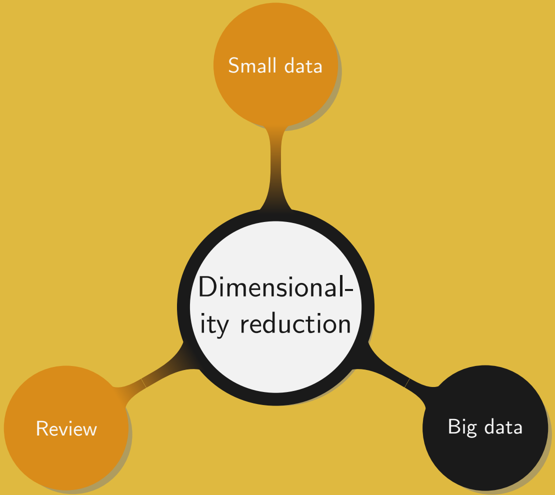
Principal components are given by $XV = U\Sigma V^\top V = U\Sigma$ and the singular values are the square of the eigenvalues.

Dimensionality reduction

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

Chapter organisation



Distributed systems

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

Tall and skinny

Setup:

- X does not fit in memory
- X 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?

Tall and skinny

From slide 4.146, we know that

$$C = X^T X = V \Sigma^2 V^\top . \tag{4.2}$$

Also C can fit in memory since it has dimension $n \times n$ and by assumption a single machine has n^2 memory. Thus we *hope to compute* $X^\top X$, without any dependence on m. Then using equation 4.2 we can find the eigenvalues of C and retrieve V and Σ. The main challenge is to efficiently compute $X^\top X$, while ensuring 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

Tall and skinny

Simple MapReduce approach to compute X^TX:

- Mapper: for all pairs (x_{ij}, x_{ik}) on row i, return the $(key, value)$ pair
$$\left((c_j, c_k), x_{ij}x_{ik}\right),$$
where c_j and c_k correspond to columns j and k, respectively.
- Reducer: consider all the $(key, value)$ pairs $\left((c_i, c_j), \langle v_1, \cdots, v_R\rangle\right)$, 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 similarity

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 exert of d_1
- What is likely to happen?

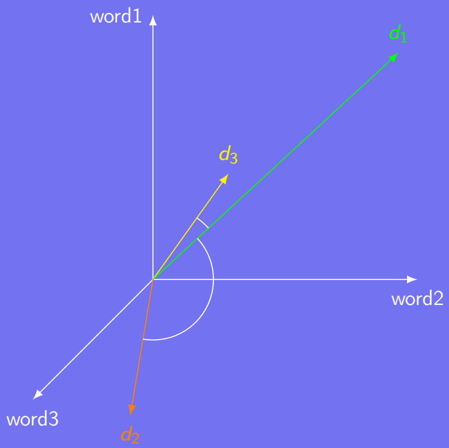
Cosine similarity

For two vectors d_i and d_j we define their *cosine similarity* as

$$\cos(d_i, d_j) = \frac{\langle d_i, d_j \rangle}{\|d_i\| \|d_j\|}.$$

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

Tall and skinny with cosine similarity

We fix γ, 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

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

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

- Reducer: consider all the $(key, value)$ pairs $\left((c_i, c_j), \langle v_1, \cdots, v_R\rangle\right)$, 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 $\frac{1}{\gamma}\sum_{i=1}^R v_i$

Tall and skinny with cosine similarity

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 $X^\top X \approx DX'D$.

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.

Tall and skinny with cosine similarity

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 γ can be adjusted depending on what is expected:
  - Preserve similar entries in $X^\top X$: $\gamma = \Omega\left(\frac{\log n}{s}\right)$, where s is the lowest cosine similarity
  - Preserve singular values of $X^\top X$: $\gamma = \Omega\left(\frac{n}{\varepsilon^2}\right)$, where ε is the relative error

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

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

Distributed QR

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 “tiled 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 bocks $X_{i,j}$ on their right
- Repeat for all blocks on the diagonal

Looking at the big picture

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?

Random projections

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,

$$\|v_i\|_2 \approx \|u_i\|_2 \text{ and } \|v_i - v_j\|_2 \approx \|u_i - u_j\|_2 \tag{4.3}$$

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 $1 \pm \varepsilon$.

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.

Random projections

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 selected following a symmetric distribution about zero with unit variance

Random projections

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 ε-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

Dimensionality reduction for big data:

Preserving important structural properties of the data:

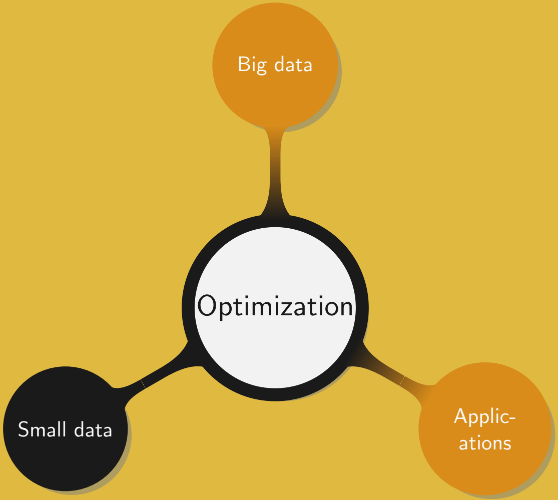
- Tall and skinny can be handle using SVD
- For other cases random projections are a good alternative
- PCA will ensure a good result, random projections might not
- Randomized PCA:
    - Randomly project X to obtain X'
    - Perform a QR decomposition on $X' = QR$
    - Compute $B = Q^\top X$
    - Run SVD on $B = U_1 \Sigma V^\top$
    - Approximate the SVD of X by $QQ^\top X = Q\left(U_1 \Sigma V^\top\right) = U \Sigma V^\top$
    - Complete PCA the usual way

Key points

- How to perform dimensionality reduction?
- What is the relationship between SVD and PCA?
- What is cosine similarity?
- How to deal with tall and skinny big data?
- What are random projections?

5. Optimization

Chapter organisation



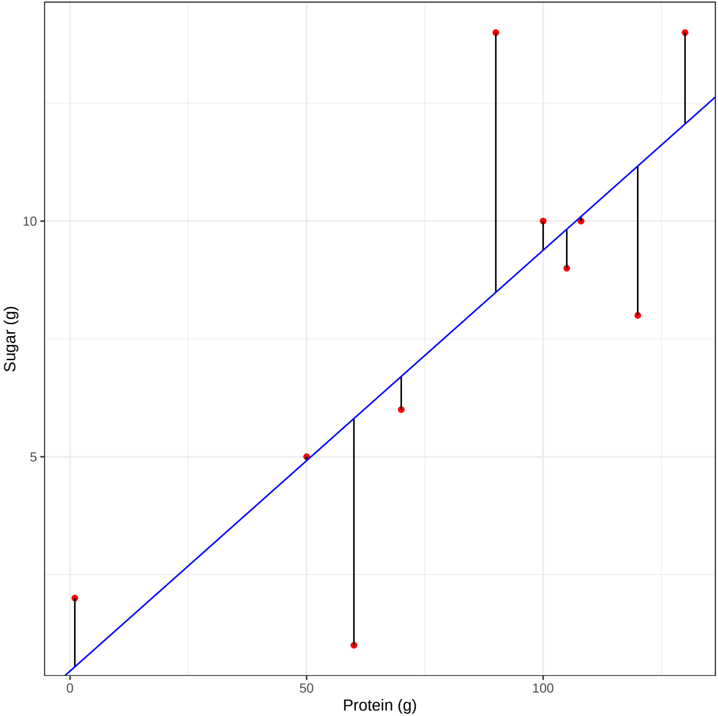
Regression analysis

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



10
5
0
50
100
Protein (g)
Sugar (g)

Optimization problems

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

Notes on optimization problems

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

Gradient descent

Description is usually done in term of minimization

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

- 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 $\alpha \in [0,1]$,

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

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

Gradient descent

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

Comparison of gradient descent methods

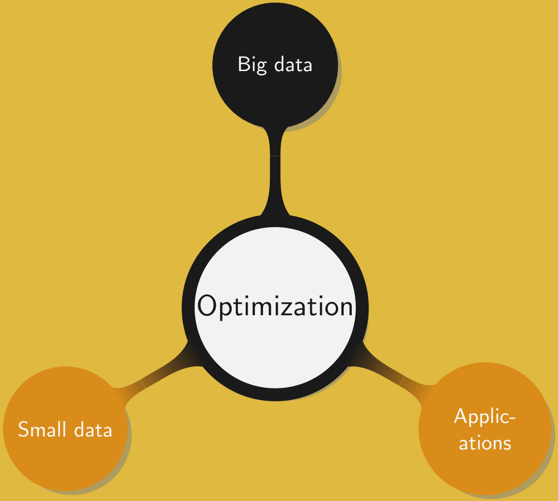
Gradient descent:

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

Stochastic gradient descent:

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

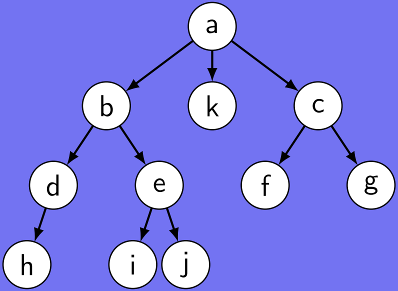
Chapter organisation



Directed Acyclic Graph

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



Work-depth model

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_∞

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_∞ tend to zero?

Bounds on T_p

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_∞ helps define how far we are from the ideal case

Remarks.

- Together T_1 and T_∞ 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

Back to gradient descent

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 α. 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 ∇F is L-Lipschitz continuous. In such a case taking $\alpha < \frac{1}{L}$ leads to an exponential convergence rate to a global minimum!

Gradient descend – Sum of squares

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?

Parallel summation

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 $a$
$s \leftarrow 0$ ;
for $i \leftarrow 1, \cdots, n$ do
    $s \leftarrow s + a[i]$ ;
end for
return $s$
```

Basic summation:

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

How to achieve:

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

Gradient descend – Work and depth

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?

Stochastic gradient descend

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

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

Notes on stochastic gradient descent:

- Error ε 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

Stochastic vs. batch gradient descent

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?

Parallelizing 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?

Parallelizing Stochastic gradient descent

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:

- Both $x_i = (x_i^{(1)}, \cdots, x_i^{(n)})$ and $w = (w^{(1)}, \cdots, w^{(n)})$ are n-dimensional vectors
- If x_i is sparse, then only a few $w^{(k)}$ will be updated

In a big data setup the probability of collision will likely be low

Going Hogwild!

Hogwild! strategy for p processors:

- Proceed in parallel over all available CPUs
- Until the expected error condition is met:
  - Select a random index j from $\{1, \cdots, m\}$
  - Concurrently compute $F_j(w)$ and $\nabla F_j(w)$ for the w currently in the shared memory
  - For all k such that $x_i^{(k)} \neq 0$, update $w^{(k)}$ with $w^{(k)} - \alpha \left[\nabla F_j(w)\right]^{(k)}$

Remarks.

- No locks are used, leading to speed up nearly linear in term of CPUs
- The locking overhead should be avoided when processing big data
- Hogwild! requires the cost function to be sparse

Stochastic vs. batch gradient descent

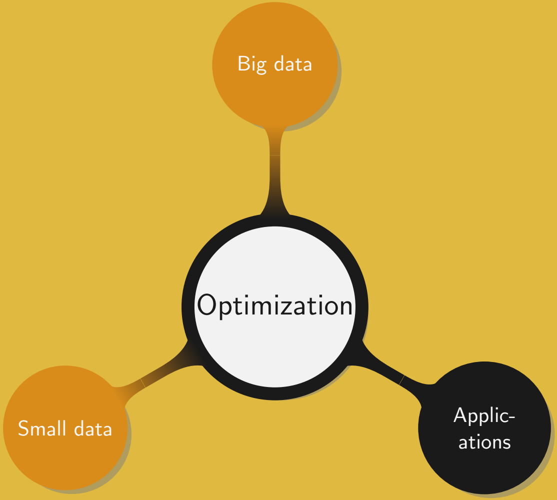
Looking back at gradient descent in PRAM:

- Stochastic has lower depth but is hard to parallelize
- Batch is easier to parallelize but slower
- Hogwild! renders stochastic almost *embarrassingly parallel*

Stochastic and batch on a distributed system:

- Communication based on the number of iterations:
  - Batch: $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$
  - Stochastic: $\mathcal{O}\left(\frac{1}{\varepsilon}\right)$
- Stochastic is often preferred on a single computer with many GPUs
- Batch is more common on distributed systems

Chapter organisation



Implementing gradient descent

Questions to consider first:

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

In our setup we expect to:

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

How to store the data on the cluster?

Batch gradient descent in Spark

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

Batch gradient descent in Spark

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?

Stochastic gradient descent in Spark

Reminders on stochastic gradient descent:

- Total depth is much larger than for batch gradient descent
- Apply Hogwild! to speed up the process when data is sparse

Hogwild! on Spark:

- Broadcast needs to be completed to start the mappers
- All mappers and reducers must be done before broadcasting again
- Spark achieves fault-tolerance through synchronisation barriers

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

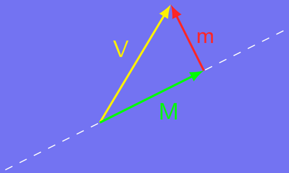
Gradient descent and PCA

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 $XX^\top$ 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

Linear programming

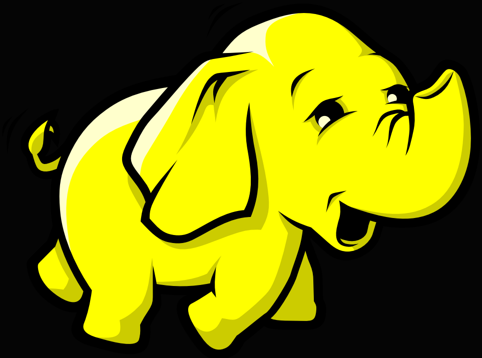
> *Given a mathematical model where an objective is represented as a linear function and some constraints are expressed as equalities or inequalities, maximize the objective while respecting the constraints.*

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" finds 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

Key points

- What are batch and stochastic gradient descents?
- How to batch and stochastic gradient descent compare?
- What are PRAM, work, and depth?
- Explain how Hogwild! works
- Discuss gradient descent implementation on Spark



Thank you, enjoy the Summer break!

References I

1.30 https://upload.wikimedia.org/wikipedia/commons/0/0e/Hadoop_logo.svg