An introduction to Big Data

What we are going to do

- Define big data
- Why the hype behind big data
- Architectures
    - Hardware
    - Software
        - Data storage
        - Data processing (batch, interactive, streaming)
    - Reference architectures

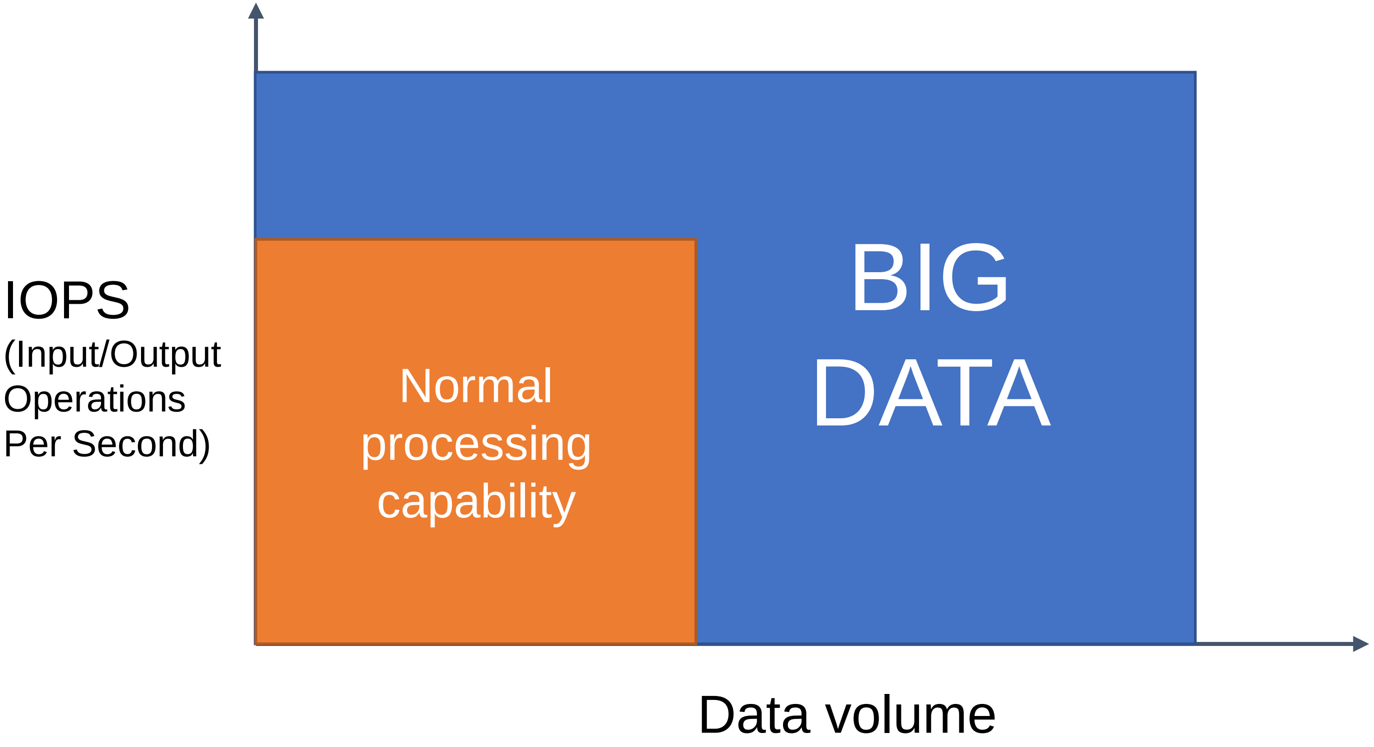
Define *big data*

The term *big data* comes in two flavors

- Noun: “We have *big data*"
- Adjective: “We use *big data* tools"

Big data as a noun

When does data become *big*?



Big data as a noun

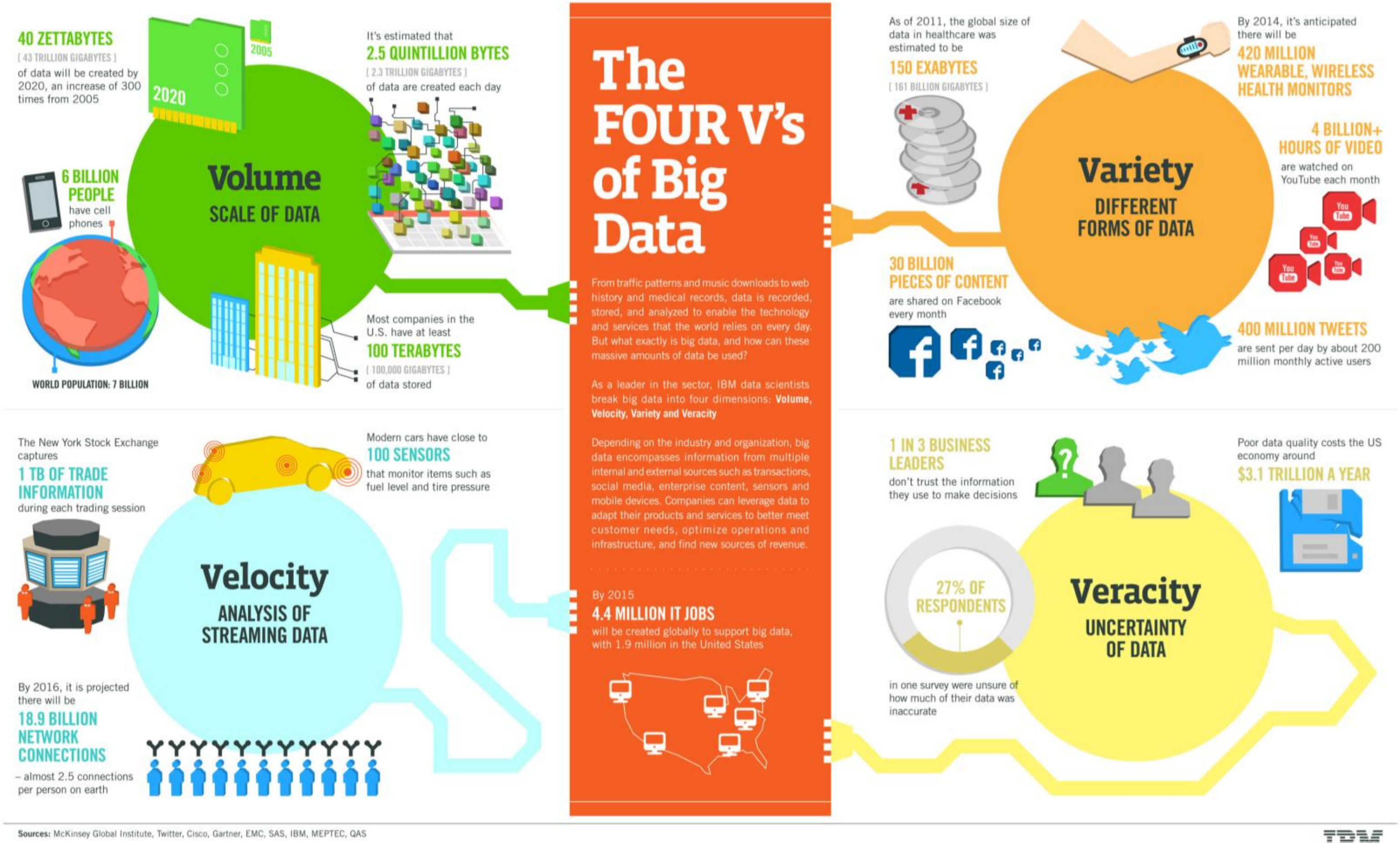
Common definitions

- "Big data **exceeds the reach of commonly used hardware** environments **and software** tools to capture, manage, and process it with in a tolerable elapsed time for its user population."
  Teradata Magazine article, 2011

- "Big data refers to data sets whose **size is beyond the ability of typical** database **software** tools to capture, store, manage and analyze."
  The McKinsey Global Institute, 2012

- "Big data is data sets that are **so voluminous and complex that traditional** data processing application **softwares are inadequate** to deal with them."
  Wikipedia

Big data as a noun – The V's



From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015 **4.4 MILLION IT JOBS** will be created globally to support big data, with 1.9 million in the United States

40 ZETTABYTES [43 TRILLION GIGABYTES] of data will be created by 2020, an increase of 300 times from 2005

6 BILLION PEOPLE have cell phones

It's estimated that **2.5 QUINTILLION BYTES** [2.3 TRILLION GIGABYTES] of data are created each day

Most companies in the U.S. have at least **100 TERABYTES** [100,000 GIGABYTES] of data stored

The New York Stock Exchange captures **1 TB OF TRADE INFORMATION** during each trading session

Modern cars have close to **100 SENSORS** that monitor items such as fuel level and tire pressure

By 2016, it is projected there will be **18.9 BILLION NETWORK CONNECTIONS** – almost 2.5 connections per person on earth

As of 2011, the global size of data in healthcare was estimated to be **150 EXABYTES** [161 BILLION GIGABYTES]

By 2014, it's anticipated there will be **420 MILLION WEARABLE, WIRELESS HEALTH MONITORS**

4 BILLION+ HOURS OF VIDEO are watched on YouTube each month

30 BILLION PIECES OF CONTENT are shared on Facebook every month

400 MILLION TWEETS are sent per day by about 200 million monthly active users

1 IN 3 BUSINESS LEADERS don't trust the information they use to make decisions

Poor data quality costs the US economy around **$3.1 TRILLION A YEAR**

27% OF RESPONDENTS in one survey were unsure of how much of their data was inaccurate

Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS

http://www.ibmbigdatahub.com/sites/default/files/infographic_file/4-Vs-of-big-data.jpg

Big data as a noun – The V’s

Volume

- Large quantity of data

Velocity

- Refers to the speed of data production…
- …and to the speed of consumption and analysis

Variety

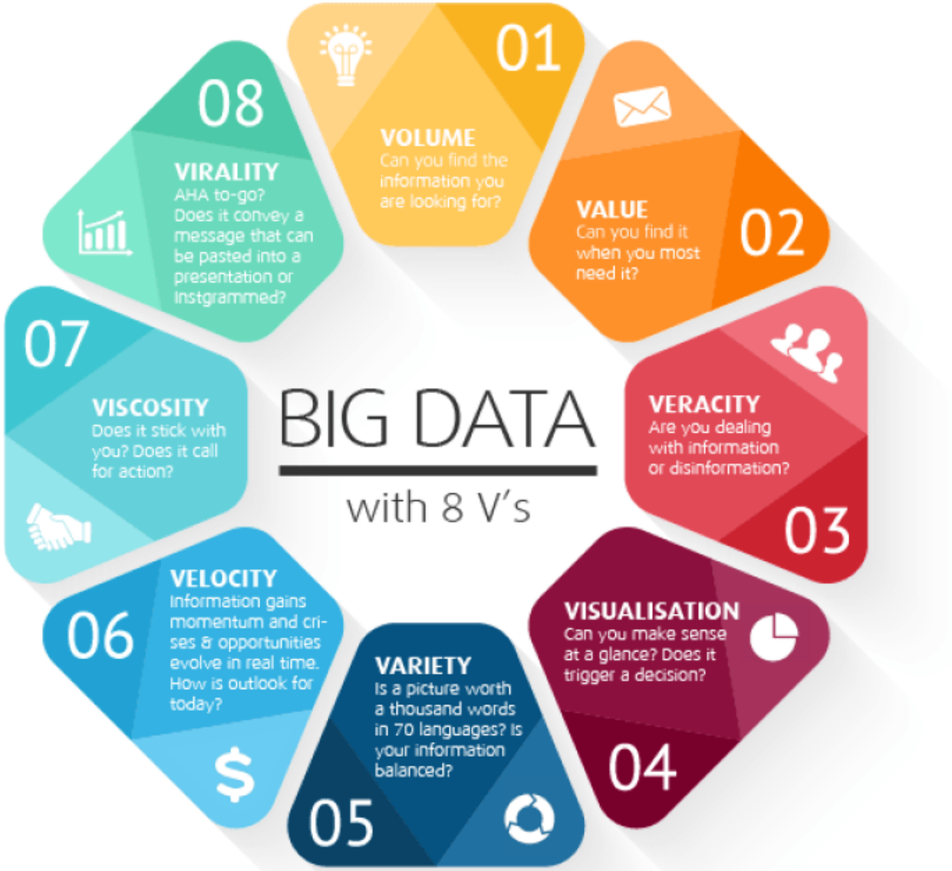
- Structured, unstructured, multimedia

Veracity

- Refers to the trustworthiness of data
- Potentially inconsistent, incomplete, ambiguous, obsolete, false, approximate

Big data as a noun – The V’s

How many V’s?



BIG DATA

with 8 V’s

01 **VOLUME** Can you find the information you are looking for?

02 **VALUE** Can you find it when you most need it?

03 **VERACITY** Are you dealing with information or disinformation?

04 **VISUALISATION** Can you make sense at a glance? Does it trigger a decision?

05 **VARIETY** Is a picture worth a thousand words in 70 languages? Is your information balanced?

06 **VELOCITY** Information gains momentum and crises & opportunities evolve in real time. How is outlook for today?

07 **VISCOSITY** Does it stick with you? Does it call for action?

08 **VIRALITY** AHA to-go? Does it convey a message that can be pasted into a presentation or Instgrammed?

Big data as an adjective

When used as a noun, the boundary between *normal* and *big* data is vague

When used as an adjective, its meaning is more specific

- Big data architecture (e.g., the Lambda architecture)
- Big data tools (e.g., Apache Spark)
- Big data paradigm (e.g., Map-Reduce)

Why the hype

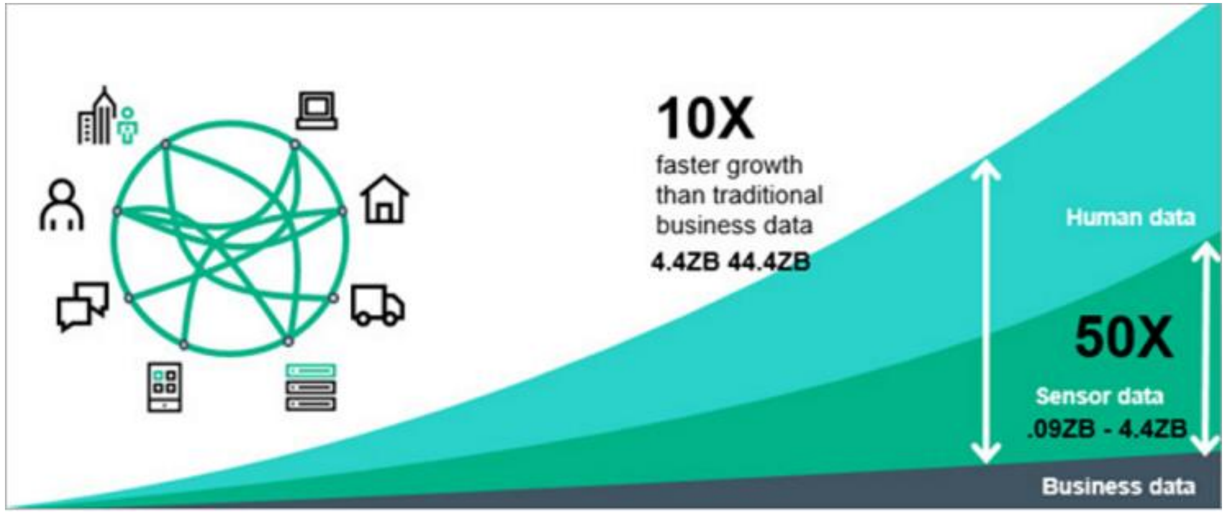
We have always known that data is powerful

What has changed?

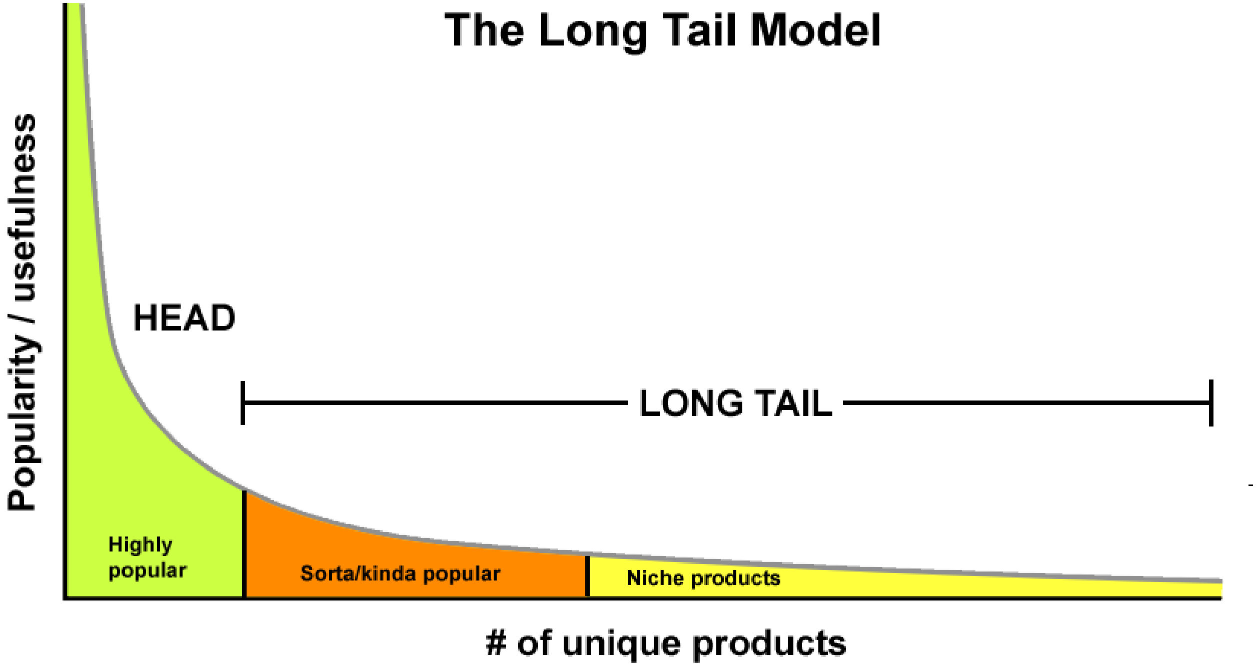
Data growth

New sources that drive an exponential growth of data

- Opens to new analytical opportunities



The long tail



> "We sold more items today that didn't sell at all yesterday than we sold today of all the items that did sell yesterday" - *Amazon employee*

The highest value does not come from the small set of highly popular items, but from the long list of niche items

- Put together, the *insignificant* data is actually the most valuable
- The inverse of the 80-20 Pareto rule

Bigger = Smarter

Google Translate

- You collect snippets of translations
- You match sentences to snippets
- You continuously debug your system

Why does it work?

- There are tons of snippets on the Web
- The accuracy improves as the training set grows

Architectures

Hardware-wise

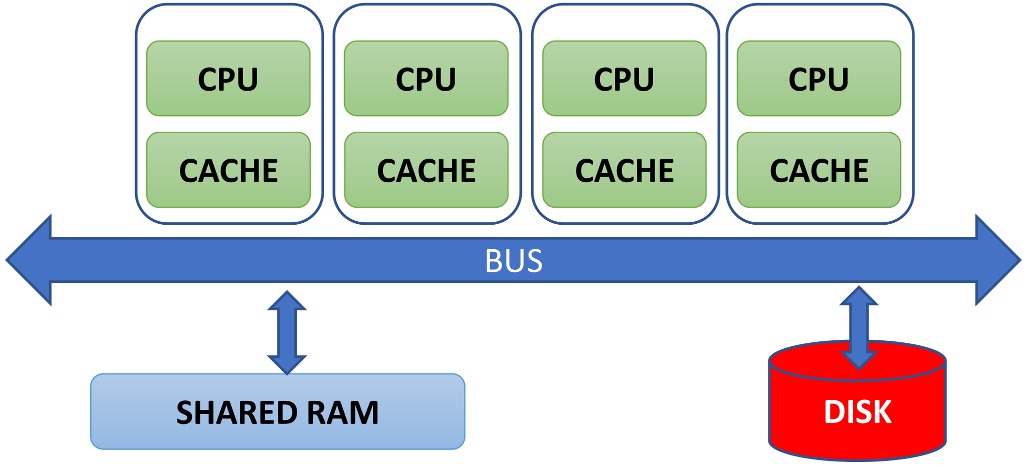
Software-wise

- Data storage (files, databases)
- Data processing (batch, interactive, streaming)
- Reference architecture (components, lambda vs kappa)

Scaling

First of all, big data is *big*

- It doesn't fit a single drive
- It doesn't fit a single (typical) machine
  - *Symmetric Multi Processing* (SMP): several processors share the same RAM, the same I/O bus and the same disk(s)
  - Limited number of mountable physical devices
  - BUS bottleneck



Second, processing big data requires a lot of computing resources

- Simply scaling only the disk is not an option

What do we do?

- Scale up
- Scale out

Scale up

Generally refers to adding more processors and RAM, buying a more expensive and robust server

Pros

- Less power consumption than running multiple servers
- Cooling costs are less than scaling horizontally
- Generally less challenging to implement
- Less licensing costs
- Less networking equipment

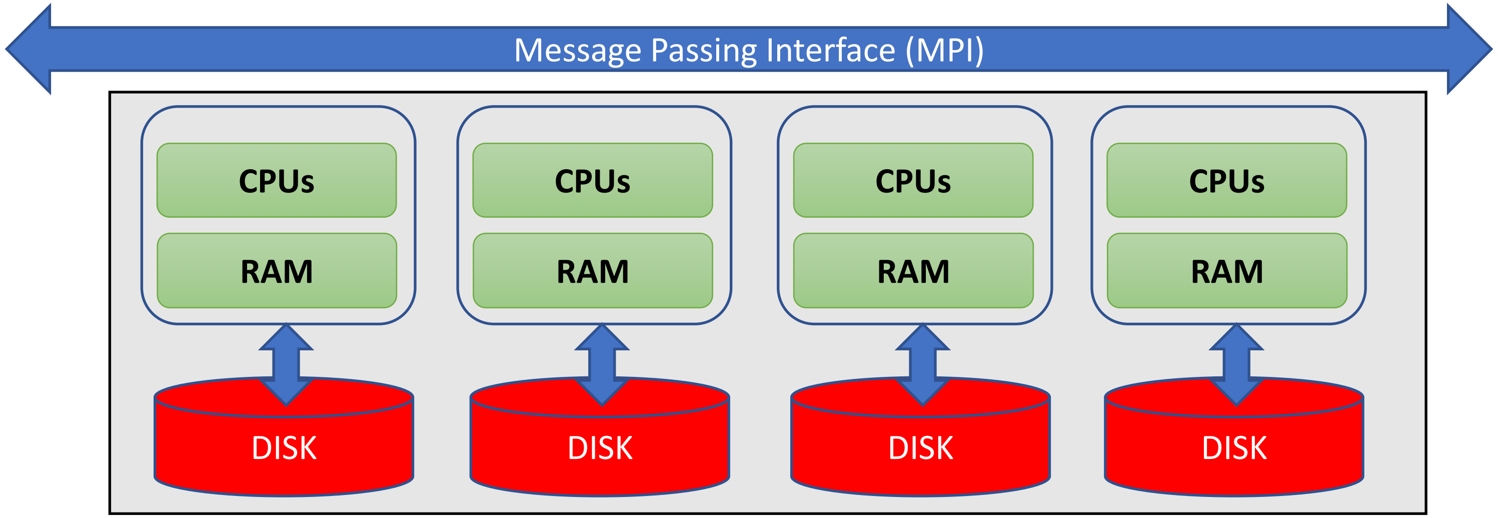
Cons

- PRICE
- Greater risk of hardware failure causing bigger outages
- Generally severe vendor lock-in
- Not long-term: limited upgradeability in the future

MPP

In a *Massively Parallel Processing* (MPP) architecture there are several processors, equipped with their own RAM and disks, collaborating to solve a single problem by splitting it in several independent tasks

- It is also called shared-nothing architecture
- Mainly used for Data Warehouse applications (e.g., Teradata, Nettezza, Vertica)



Scale out

Generally refers to adding more servers with less processors and RAM

Pros

- Much cheaper than scaling vertically
- New technologies simplify fault-tolerance and systems monitoring
- Easy to upgrade
- Usually cheaper
- Can literally scale infinitely

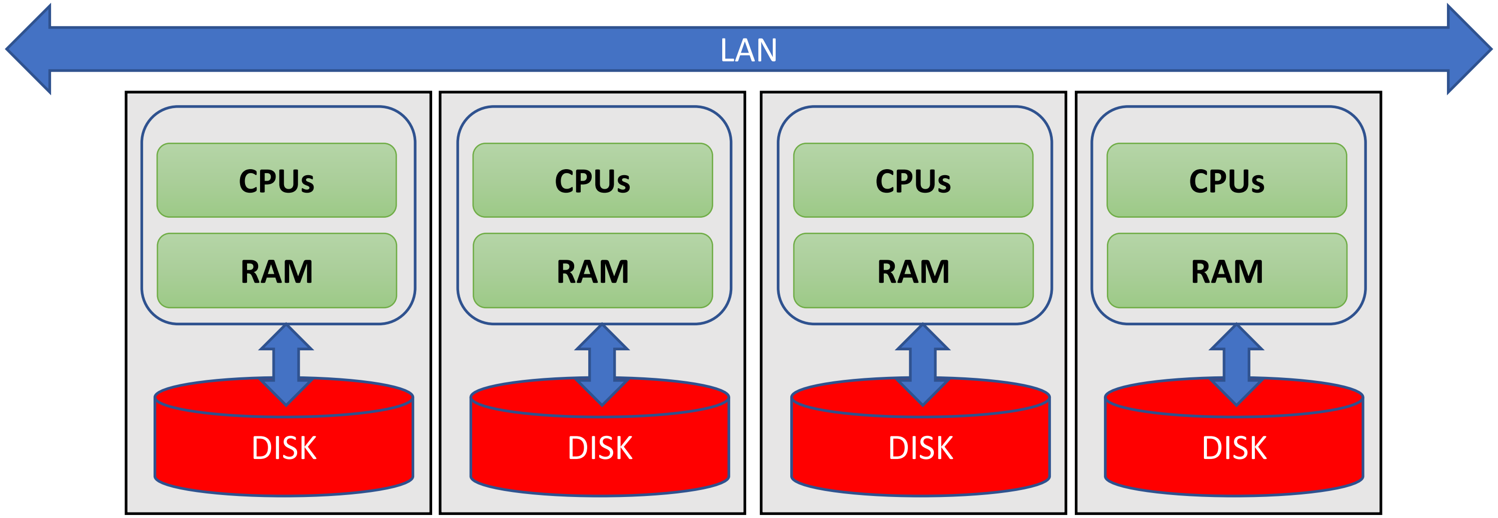
Cons

- More licensing fees
- Bigger footprint in the Data Center
- Higher utility cost (electricity and cooling)
- More networking equipment (switches/routers)

Cluster

A computer cluster is a group of linked computers (nodes), working together closely so that in many respects they form a single computer

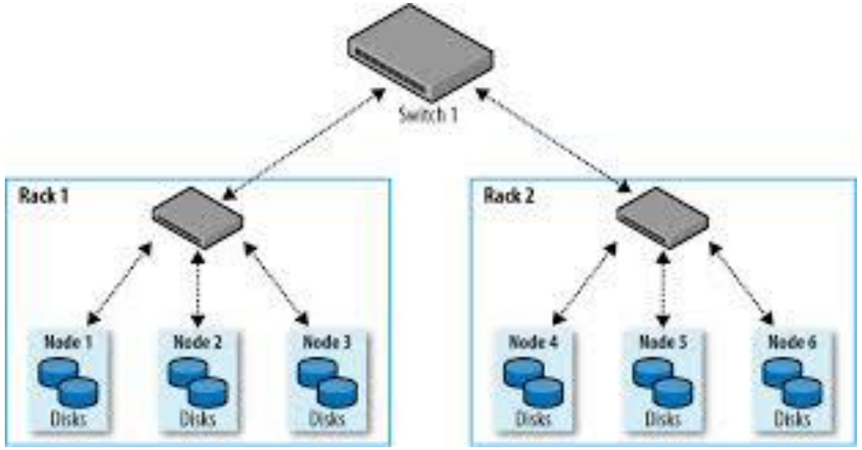
- Typically connected to each other through fast LAN (slower than MPI)
- Still shared-nothing, but every node is a system on its own, capable of independent operations
  - Unlimited scalability, no vendor lock-in
- Number of nodes in the cluster >> Number of CPUs in a node



Cluster

Compute nodes are stored on racks

- 8–64 compute nodes on a rack
- There can be many racks of compute nodes
- The nodes on a single rack are connected by a network (typically gigabit Ethernet)
- Racks are connected by another level of network (or a switch)
    - Intra-rack bandwidth >> inter-rack bandwidth



Commodity hardware

You are not tied to expensive, proprietary offerings from a single vendor

You can choose standardized, commonly available hardware from a large range of vendors to build your cluster

Commodity ≠ Low-end!

- Cheap components with high failure rate can be a false economy

Commodity hardware

Example of commodity hardware specifications:

- Processor: 12-core i7-8700 CPU @ 3.20GHz
- Memory: 64 GB RAM
- Storage: 3 × 4TB SATA disks
- Network: Gigabit Ethernet

Yahoo!’s Hadoop installation:

- > 100,000 CPUs in > 60,000 computers (as of 2017)
- Used to support research for Ad Systems and Web Search
- Also used to do scaling tests to support development of Hadoop

Multiple clusters

Having a single large cluster that is tantalizing to many organizations

- No data silos, simpler governance

Multiple clusters are inevitable within medium-large enterprise settings

- Resiliency
  - Every cluster sits within a single point of failure due to geography
- Software development
  - Mitigate the risk of impacting critical production environments by isolating configuration, integration, or evolution testing and deployment
- Workload isolation
  - Hardware resources tuned for specific workloads, less resource contention
- Legal separation
- Independent Storage and Compute

Multiple clusters

With the success of cloud services, the “independent storage and compute” solution for big data clusters is on the rise

- Network is no more a problem
  - Data locality is (?)
- Scalability is modularized
  - Disk and computation needs are different
- Cost-efficient
  - Data must be persistent, 24/7
  - Compute nodes are activated only on demand

Data storage

Distributed file system

Distributed databases

Distributed FS

Different implementations, relying on similar principles

- Apache Hadoop
- GCP Colossus
- AWS EFS
- Azure Data Lake

HDFS

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware

- A typical file in HDFS is gigabytes to terabytes in size. There are Hadoop clusters running today that store petabytes of data
- Applications that run on HDFS need streaming access to their data sets. HDFS is designed more for batch processing rather than interactive use. The emphasis is on high throughput of data access rather than low latency of data access
- HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access
- Hardware failure is the norm rather than the exception. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS

HDFS: blocks

Block: minimum amount of data read/written

- Disk blocks are normally 512B
- Filesystem blocks are typically a few KB

HDFS blocks range between 64MB and 1GB (default: 128MB)

- If a file smaller, it will occupy the necessary disk space, not the full block size

Why blocks?

- Files can be larger than disks. This way, storage management is simplified and replication is easier

Why this big?

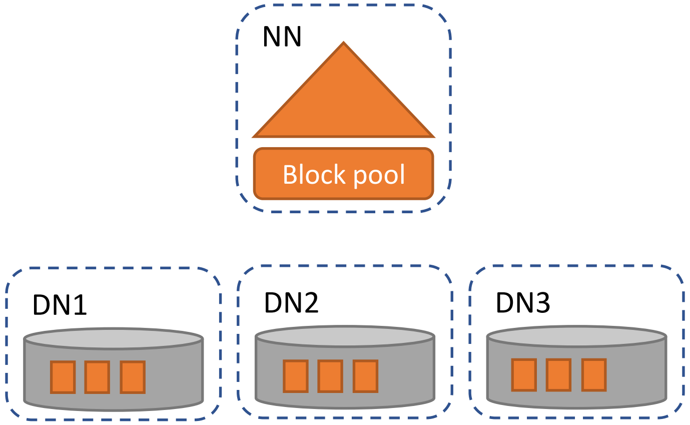
- Large files split into many small blocks require a huge number of seeks
- E.g.: block size = 4KB; file = 1GB; number of seeks = 250.000

HDFS: namenodes and datanodes

Nodes in an HDFS cluster operate in a master-slave pattern

Namenode (NN) - the master

- Persistently maintains the filesystem tree and all files' and directories' metadata
- Keeps in memory the location of each block for a given file (*block pool*)



Datanodes (DNs) - the slave

- Store and retrieve blocks
- Periodically report to the NN with the list of blocks they are storing; *heartbeats* are sent to the DN to signal their active state (every 10 minutes by default)

HDFS: the SPoF

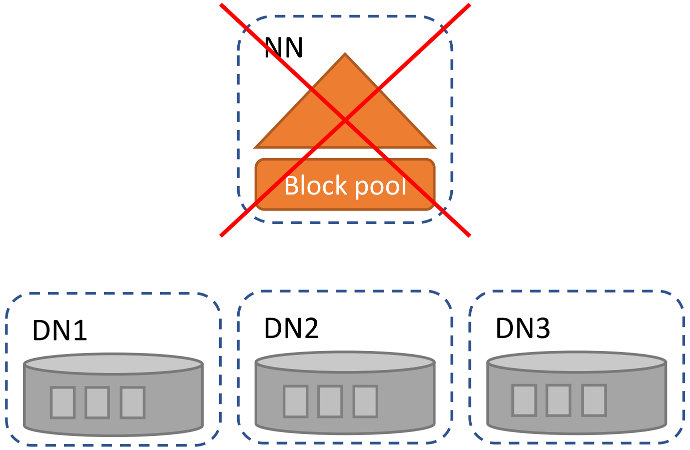
The NN is a single point of failure: without it, the filesystem cannot be used (no way to reconstruct the files from the blocks in the DNs)

Backup solution

- NN writes its persistent state to multiple filesystems, preventing loss of data

Secondary NN solution

- A separate machine regularly connects with the primary NN to build snapshots of its persistent data and saves them to local or remote directories.
- These *checkpoints* can be used to restart a failed NN without having to replay the entire journal of file-system actions.



High-Availability

High-Availability (HA) indicates a system that can tolerate faults

- The service never stops while the fault (be it hardware or software) is detected, reported, masked, and repaired off-line

Backups and secondary NNs protect against data loss..

- ..but restarting a NN could take 30 minutes or more on large clusters!

HA is supported by configuring two separate machines as NNs

- One is in an *active* state, the other is in a *standby* state
- In the event of a failure of the active NN, the standby NN takes over

The NN in standby keeps its metadata up-to-date by:

- Reading the *edit log* files written by the active NN in a shared storage
  - Edit logs are typically stored in *Journal Nodes*, which are also replicated
- Receiving the data block locations and the heartbeats directly from the DNs, which are configured to report to both NNs

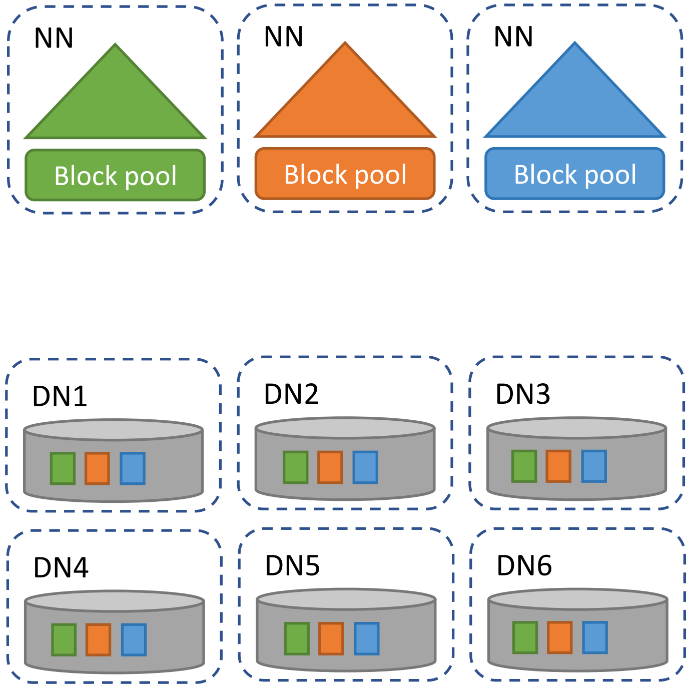
HDFS: federation

The size of the block pool is limited by the memory size of the NN

- May incur in scaling issues on very large clusters with many files

Solution: configure additional NNs

- Each NN manages a portion of the filesystem, i.e., a *namespace*
- NNs are independent of each other
- Introduced with Hadoop 2.0



HDFS: federation

Scalability

- Because the NN keeps all the namespace and block locations in memory, the size of the NN heap limits the number of files and also the number of blocks addressable. This also limits the total cluster storage that can be supported by the NN

Performance

- NN is the single point for storage and management of meta-data, it can become a bottleneck for supporting a huge number of files, especially a large number of small files

Availability

- Separate namespaces of different applications improve the overall availability of the cluster

Maintainability, Security & Flexibility

- Block pool abstraction allows other services to use the block storage with perhaps a different namespace structure. Each namespace is isolated and not aware of the others

Applications can read/write on more than one namespace

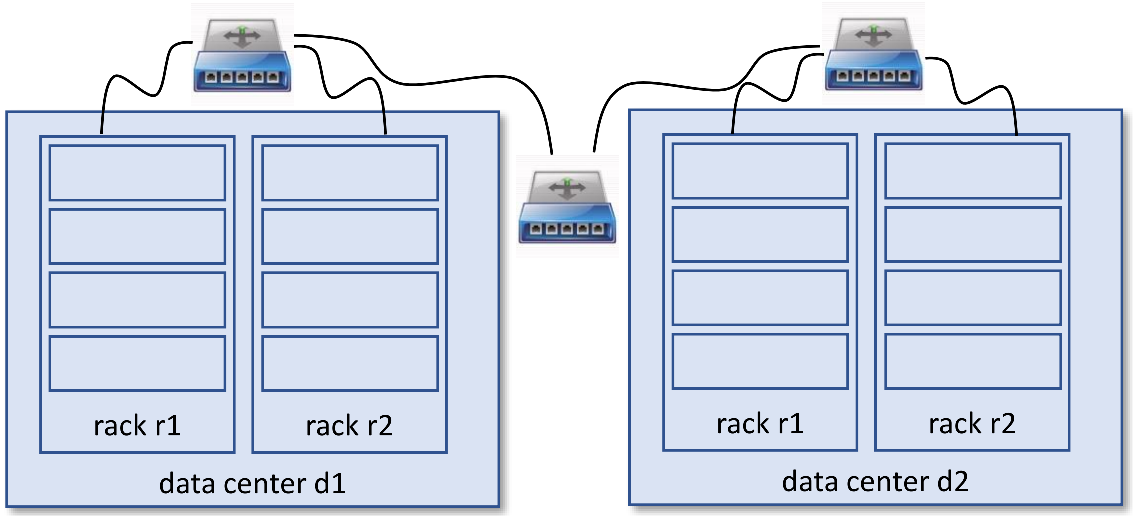
HDFS: cluster topology

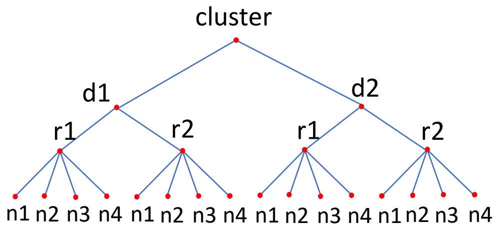
In order to carry out proper choices, Hadoop must be aware of the cluster topology that is defined during the cluster setting phase.

- Block storage and process allocation (data locality) need such information

Nodes are organized in racks, racks are organized in data centers

- Hadoop models such concepts in a tree-like fashion and computes the distance between nodes as their distance on the tree.





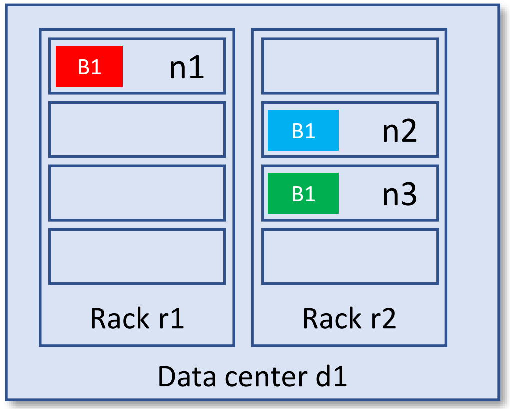
HDFS: replication (1)

Each data block is independently replicated at multiple DNs in order to improve performance and robustness

- Replication is aware of the cluster topology
- For each data block, the NN stores the list of DNs storing it

The default replication factor is 3:

- Replica 1 is stored on the node (n1) where the client issued the write command (if the client resides within the cluster)
- Replica 2 is stored on a node (n2) in a rack (r2) different from the one of n1 (off-rack)
- Replica 3 is stored on a node, different from n2, but that belongs to r2



Replicas can be rebalanced when nodes are added or unavailable

HDFS: replication (2)

Hadoop 3: support to Erasure Coding (EC) as an alternative to simple replication

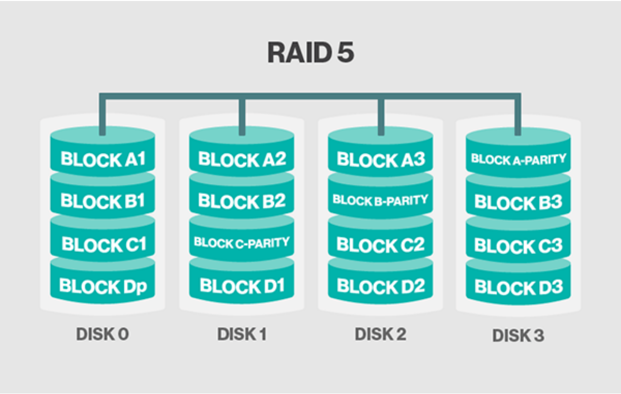
- I.e., a mechanism similar to Redundant Array of Inexpensive Disks (RAID) 5-6
- Each block is split (*striped*) across each data node
- Works best for warm and cold datasets with relatively low I/O activities

Main advantages

- Sensibly reduce data redundancy (from 200% to 50% with default setups)
- Faster writes

Main disadvantages

- Higher CPU cost
- Longer recovery time in case of failure
- **Loss of data locality**



HDFS: not always the best fit

Although this may change in the future, there are areas where HDFS is not a good fit today

Low-latency data access

- HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. Applications that require access to data in the tens of milliseconds range will not work well with HDFS.

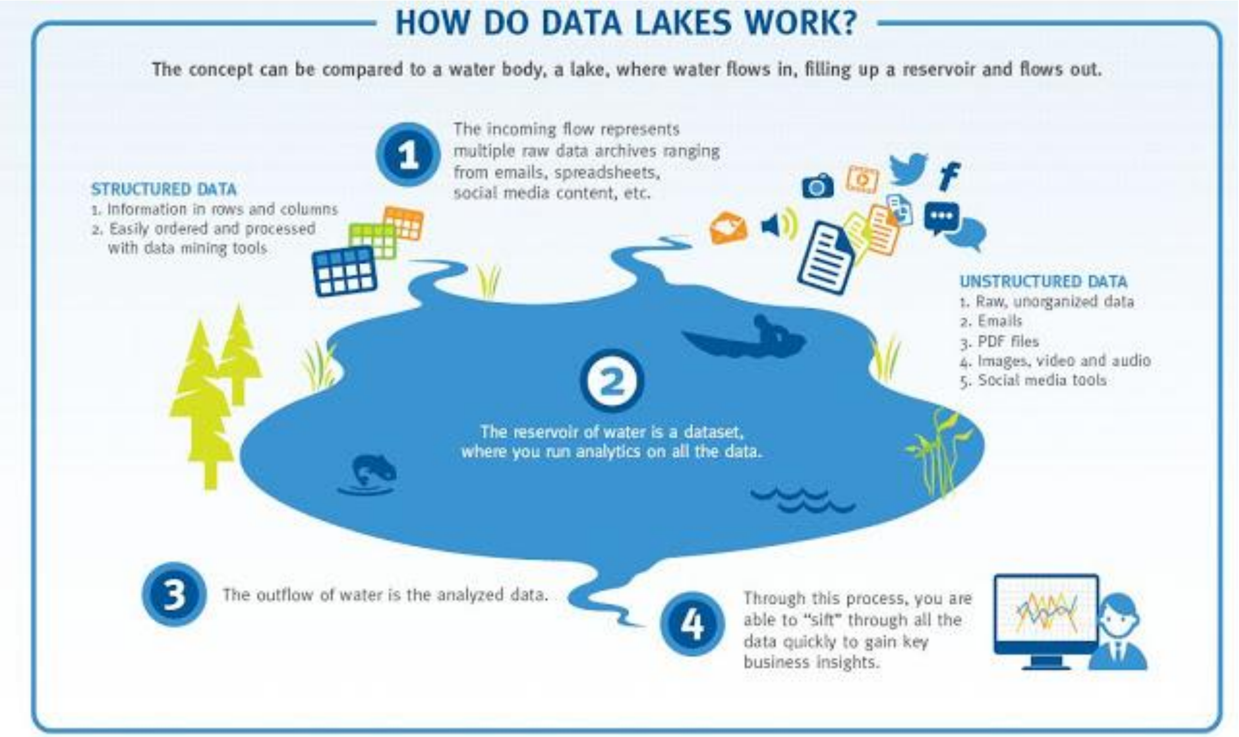
Lots of small files

- The limit to the number of files in a filesystem is governed by the amount of memory on the NN, because it holds filesystem metadata in memory.
- Although storing millions of files is feasible, billions is beyond the capability of current hardware.

The data lake

Big data stored in a DFS usually become a *data lake*

- A central location that holds a large amount of data in its native, raw format
- More on this the day after tomorrow...



HOW DO DATA LAKES WORK?

The concept can be compared to a water body, a lake, where water flows in, filling up a reservoir and flows out.

1 The incoming flow represents multiple raw data archives ranging from emails, spreadsheets, social media content, etc.

STRUCTURED DATA
1. Information in rows and columns
2. Easily ordered and processed with data mining tools

UNSTRUCTURED DATA
1. Raw, unorganized data
2. Emails
3. PDF files
4. Images, video and audio
5. Social media tools

2 The reservoir of water is a dataset, where you run analytics on all the data.

3 The outflow of water is the analyzed data.

4 Through this process, you are able to "sift" through all the data quickly to gain key business insights.

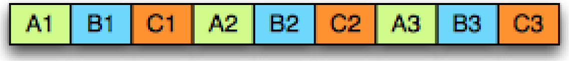
File formats

Raw files

- From simple text to multimedia files
- No support besides reading/writing the whole file (e.g., no record compression, no splittability)

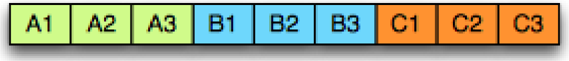
Row-oriented file formats

- Apache Avro, Google’s Protocol Buffers, Facebook’s Apache Thrift
- Better suited for row-level access to the data (e.g., OLTP)



Column-oriented file formats

- Apache Parquet, ORC
- Better suited for column-level access to the data (e.g., OLAP)
  - Better compression
  - Reduced I/O



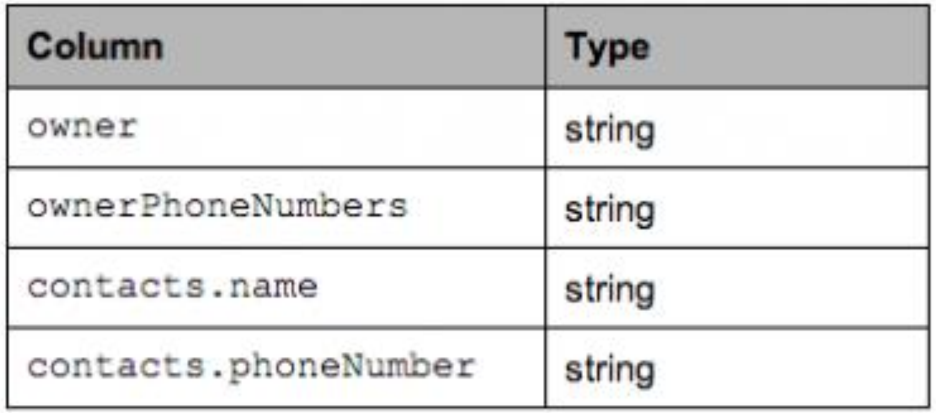
Parquet

A columnar storage format for efficient querying of **nested structures** in a **flat format**

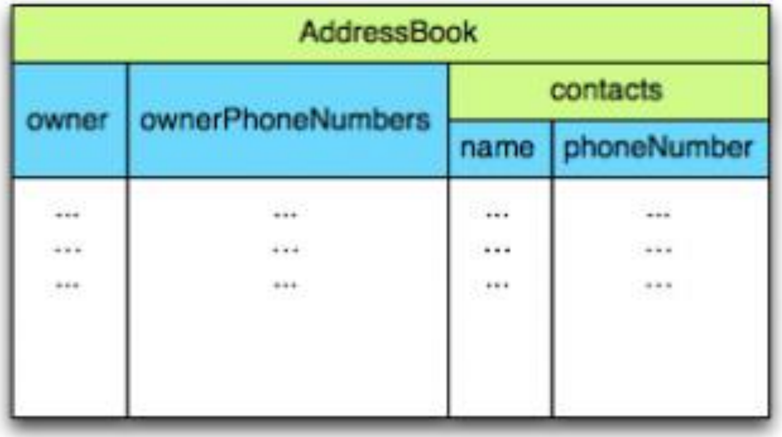
- Supported by most frameworks and query engines

One column per primitive type

- The structure of the record is captured for each value by two integers: *repetition level* and *definition level*
- Enough to fully reconstruct the nested structures



Column	Type
owner	string
ownerPhoneNumbers	string
contacts.name	string
contacts.phoneNumber	string

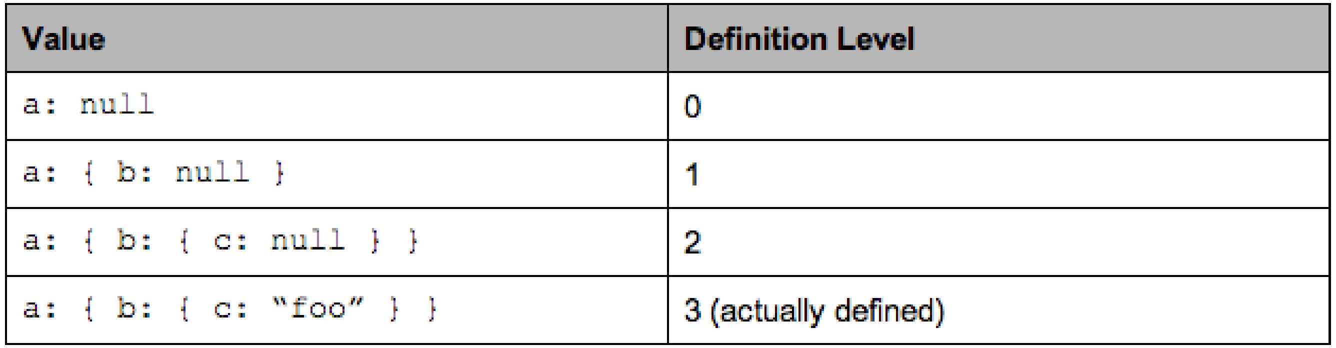


Parquet: definition level

Needed in presence of optional nested fields

- If there is a *null* value, at which level is it?

```
message ExampleDefLevel {
  optional group a {
    optional group b {
      optional string c;
    }
  }
}
```

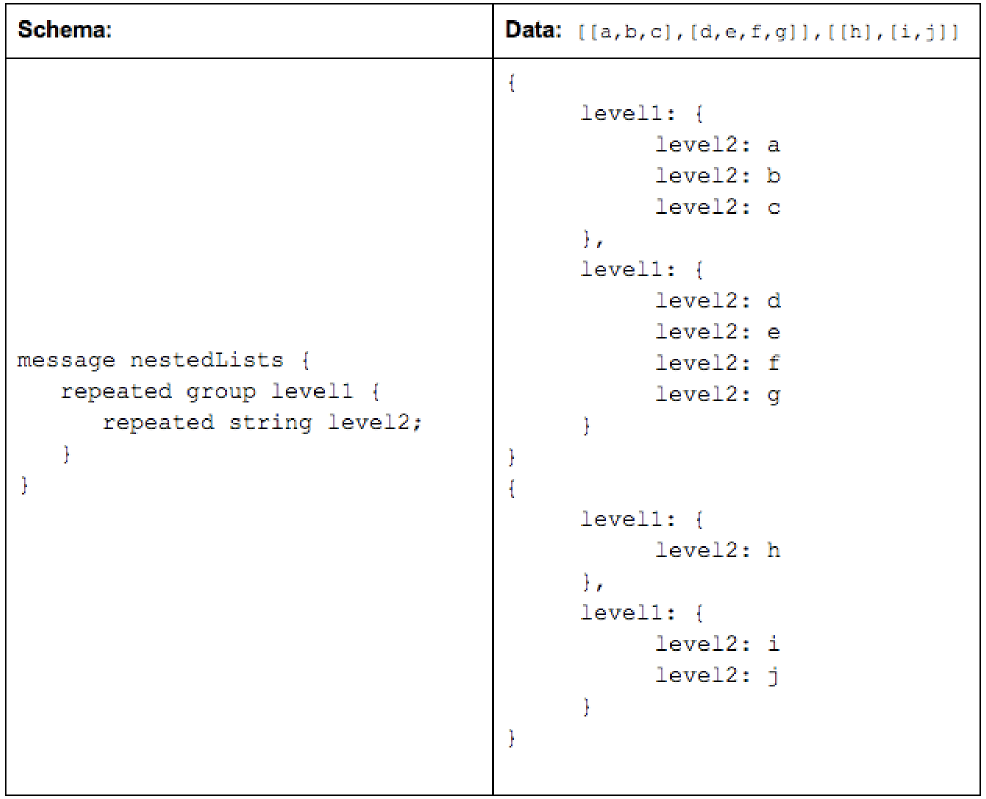


Value	Definition Level
`a: null`	0
`a: { b: null }`	1
`a: { b: { c: null } }`	2
`a: { b: { c: "foo" } }`	3 (actually defined)

Parquet: repetition level

Needed in presence of repeated fields (i.e., arrays)

- To which array/record does the value belong to?



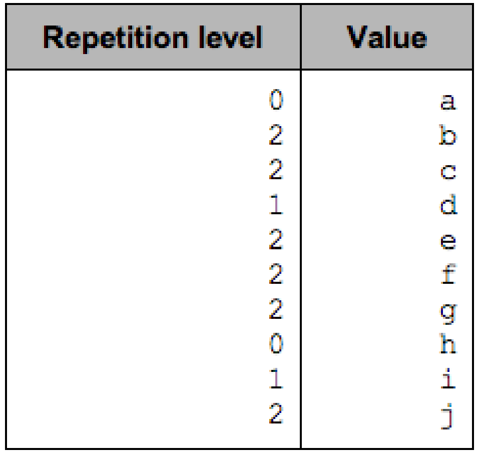
Schema:

```
message nestedLists {
   repeated group level1 {
      repeated string level2;
   }
}
```

Data: [[a,b,c],[d,e,f,g]],[[h],[i,j]]

```
{
     level1: {
          level2: a
          level2: b
          level2: c
     },
     level1: {
          level2: d
          level2: e
          level2: f
          level2: g
     }
}
{
     level1: {
          level2: h
     },
     level1: {
          level2: i
          level2: j
     }
}
```

The repetition level can be seen as a marker of when to start a new list and at which level.



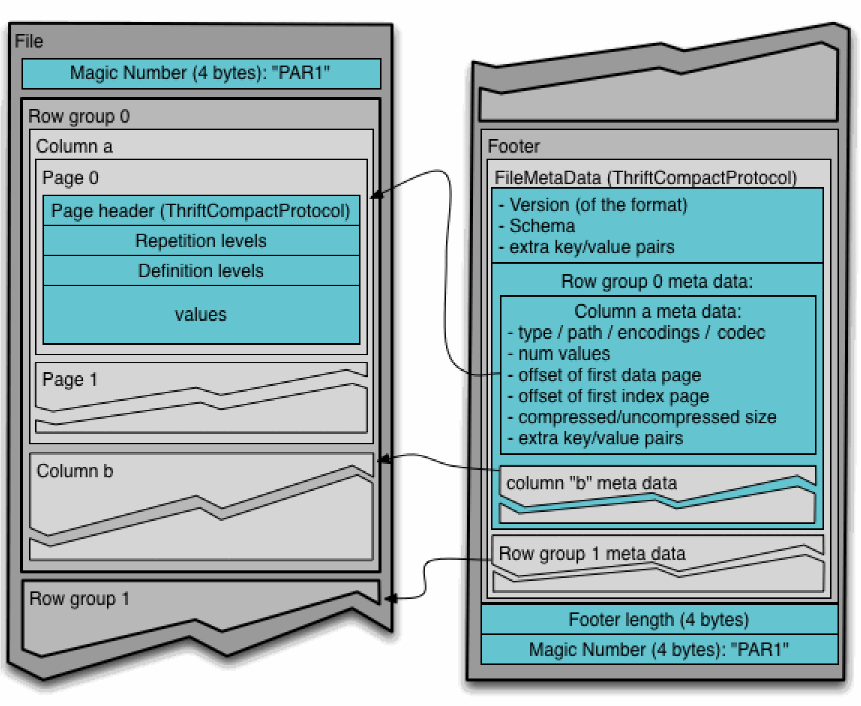
Repetition level	Value
0	a
2	b
2	c
1	d
2	e
2	f
2	g
0	h
1	i
2	j

0 marks every new record;
implies creating a new level1 and level2 list

1 marks every new level1 list;
implies creating a new level2 list as well

2 marks every new element in a level2 list

Parquet: storage and compression



File

Magic Number (4 bytes): "PAR1"

Row group 0

Column a

Page 0

Page header (ThriftCompactProtocol)

Repetition levels

Definition levels

values

Page 1

Column b

Row group 1

Footer

FileMetaData (ThriftCompactProtocol)

- Version (of the format)
- Schema
- extra key/value pairs

Row group 0 meta data:

Column a meta data:

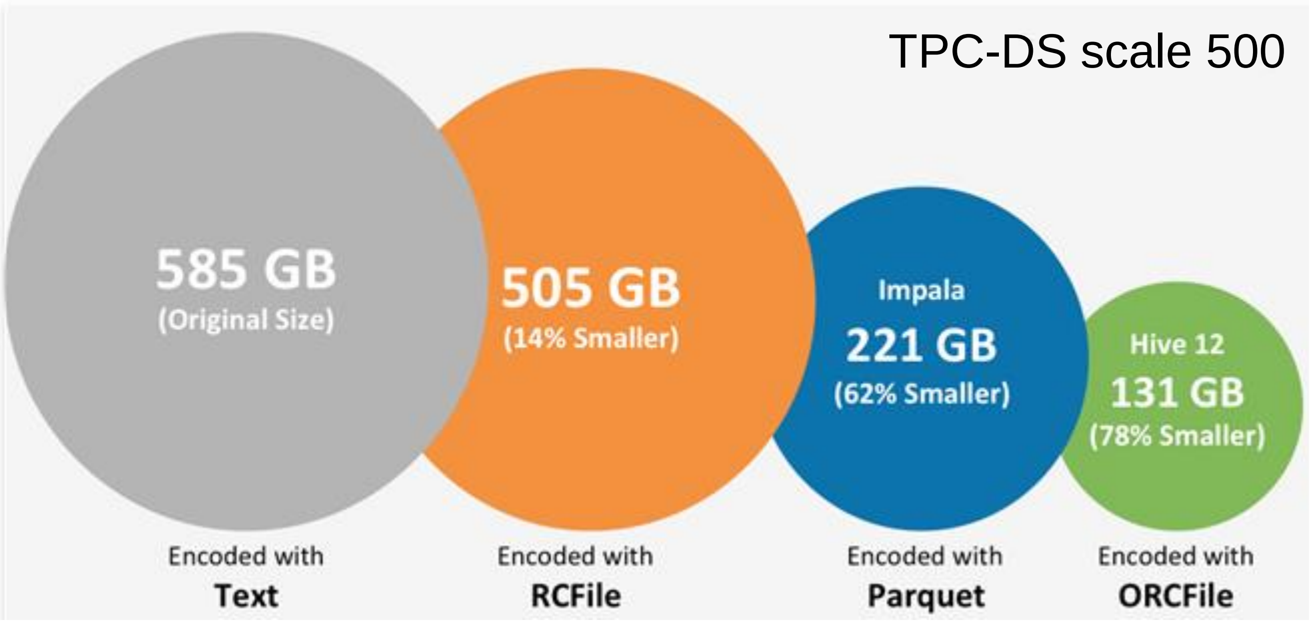
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs

column "b" meta data

Row group 1 meta data

Footer length (4 bytes)

Magic Number (4 bytes): "PAR1"



Distributed DB

New types of databases have emerged

RDBMSs are full of strengths

ACID properties

- Provides guarantees in terms of consistency and concurrent accesses

Data integration and normalization of schemas

- Several application can share and reuse the same information

Standard model and query language

- The relational model and SQL are very well-known standards
- The same theoretical background is shared by the different implementations

Robustness

- Have been used for over 40 years

RDBMSs have weaknesses as well

Impedance mismatch

- Data are stored according to the relational model, but applications to modify them typically rely on the object-oriented model
- Many solutions, no standard
  - E.g.: Object Oriented DBMS (OODBMS), Object-Relational DBMS (ORDBMS), Object-Relational Mapping (ORM) frameworks

Painful scaling-out

- Not suited for a cluster architecture
- Distributing an RDBMS is neither easy nor cheap (e.g., Oracle RAC)

Consistency vs latency

- Consistency is a must – even at the expense of latency
- Today's applications require high reading/writing throughput with low latency

Schema rigidity

- Schema evolution is often expensive

What is "NoSQL"

The term has been first used in '98 by Carlo Strozzi

- It referred to an open-source RDBMS that used a query language different from SQL

In 2009 it was adopted by a meetup in San Francisco

- Goal: discuss open-source projects related to the newest databases from Google and Amazon
- Participants: Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, MongoDB

Today, NoSQL indicates DBMSs adopting a different data model from the relational one

- NoSQL = Not Only SQL
- According to Strozzi himself, NoREL would have been a more proper noun

The first NoSQL systems

LiveJournal, 2003

- Goal: reduce the number of queries on a DB from a pool of web servers
- Solution: Memcached, designed to keep queries and results in RAM

Google, 2005

- Goal: handle Big Data (web indexing, Maps, Gmail, etc.)
- Solution: BigTable, designed for scalability and high performance on Petabytes of data

Amazon, 2007

- Goal: ensure availability and reliability of its e-commerce service 24/7
- Solution: DynamoDB, characterized by strong simplicity for data storage and manipulation

NoSQL common features

Not just rows and tables

- Several data model adopted to store and manipulate data

Freedom from joins

- Joins are either not supported or discouraged

Freedom from rigid schemas

- Data can be stored or queried without pre-defining a schema (*schemaless* or *soft-schema*)

Distributed, shared-nothing architecture

- Trivial scalability in a distributed environment with no performance decay
- Each workstation uses its own disks and RAM

Not a farewell to SQL

NoSQL data models

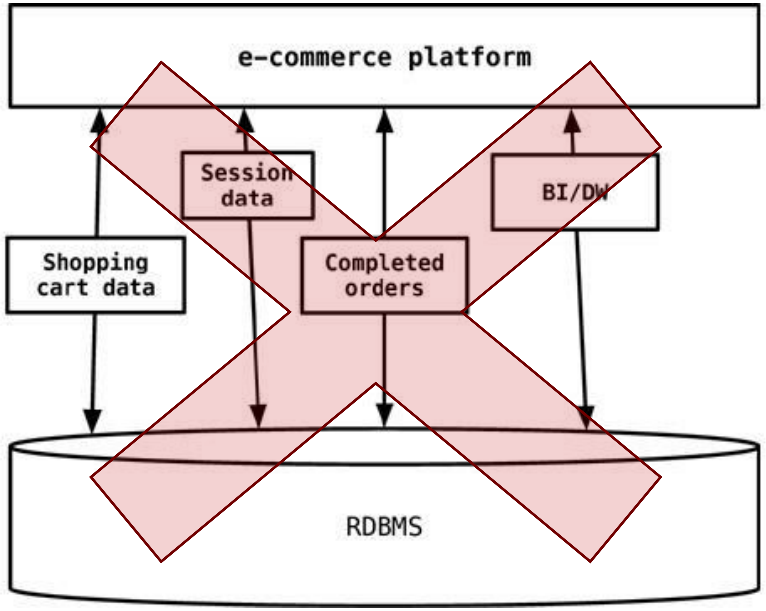
NoSQL databases mainly differ by the supported data model

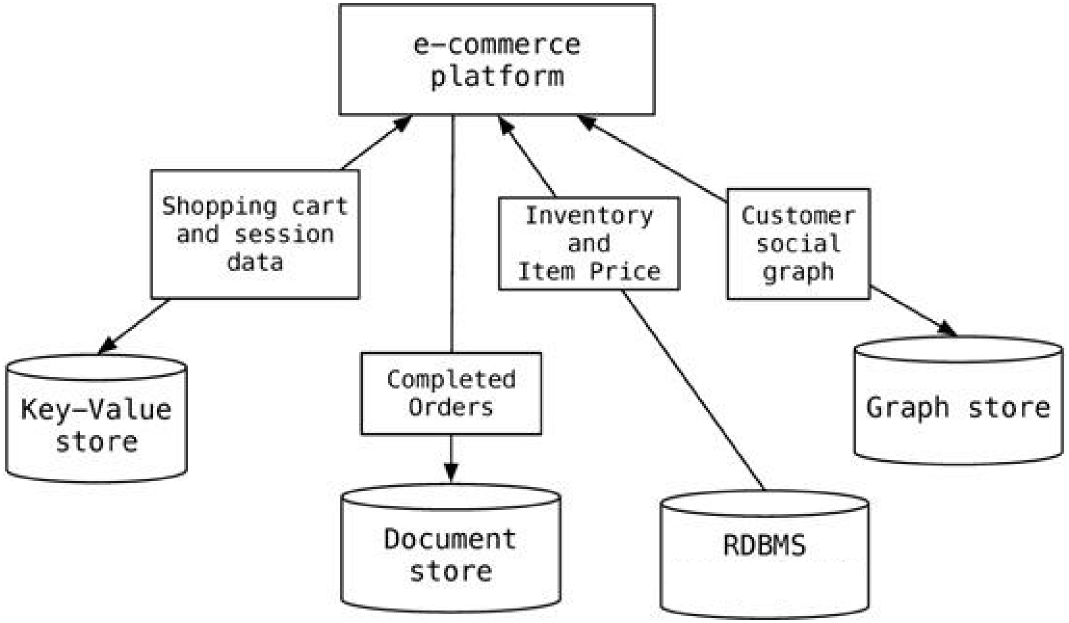
Model	Description	Use cases
Key-value	Associates any kind of value to a string	Dictionary, lookup table, cache, file and images storage
Document-based	Stores hierarchical data in a tree-like structure	Documents, anything that fits into a hierarchical structure
Wide-column	Stores sparse matrixes where a cell is identified by the row and column keys	Crawling, high-variability systems, sparse matrixes
Graph	Stores vertices and arches	Social network queries, inference, pattern matching

Polyglot persistence

The *one-size-fits-all* is “no more”

- More on this tomorrow...





Data processing

Three main kinds of data processing

- **Batch**
- Interactive
- Streaming

Batch

Simple/complex computations over large amounts of stored data

- Execution takes minutes to hours

What we need

- A cluster manager to negotiate resources
- An execution engine

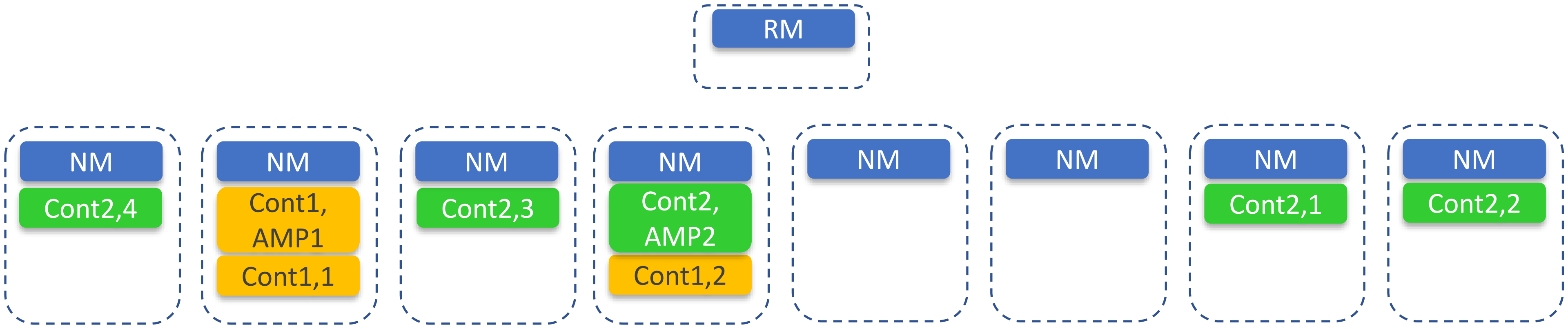
YARN

Yet Another Resource Negotiator

- Usage: assign resources for applications to run their processes

Two daemons

- Resource Manager (RM) - the master
  - The ultimate authority that arbitrates resources among all the applications
- Node Managers (NM) - the slaves
  - Responsible for the allocation and monitoring of containers



Data locality

Exploit cluster topology and data block replication to apply the data locality principle

When computations involves large set of data,
it is cheaper (i.e. faster) to move code to data
rather than data to code

The following cases respect the order the resource manager prefers:

1. Process and data on the same node
2. Process and data on the different node of the same rack
3. Process and data on different racks of the same data center
4. Process and data on different racks of the different data centers

YARN: Scheduler

YARN provides a choice of schedulers and configurable policies

FIFO Scheduler

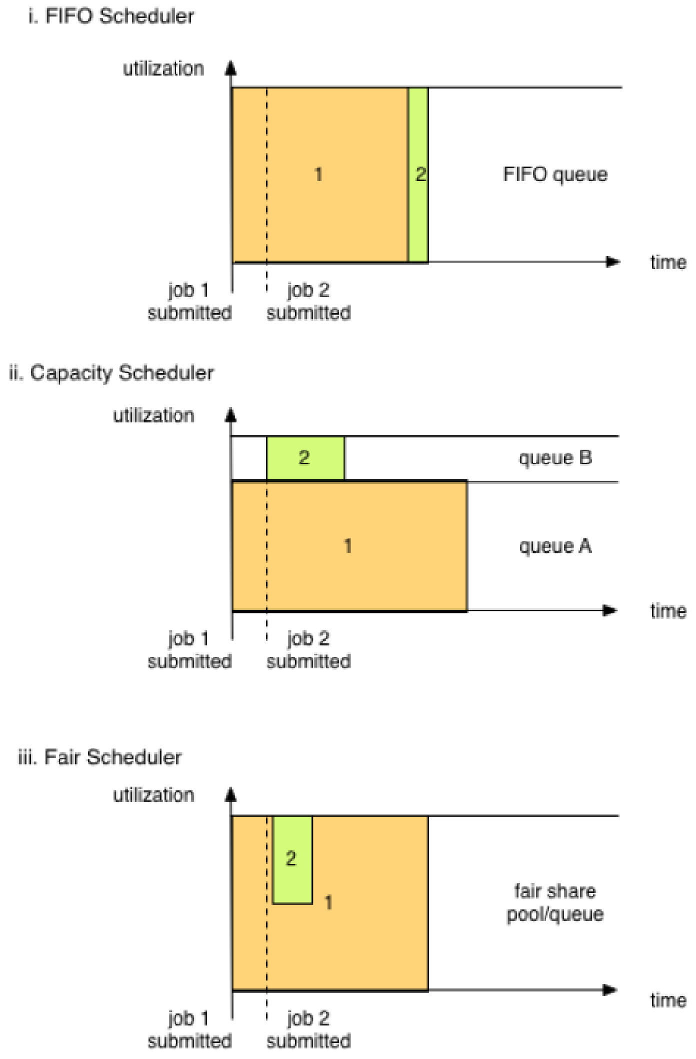
- Simple to understand, no configuration needed
- Not suitable for shared clusters

Capacity Scheduler (default)

- Reserves a fixed amount of capacity to each job

Fair Scheduler

- Dynamically balances the available resources between all running jobs



Typical Large-Data Problem

Iterate over a large number of records

Extract something of interest from each

Map

Shuffle and sort intermediate results

Aggregate intermediate results

Reduce

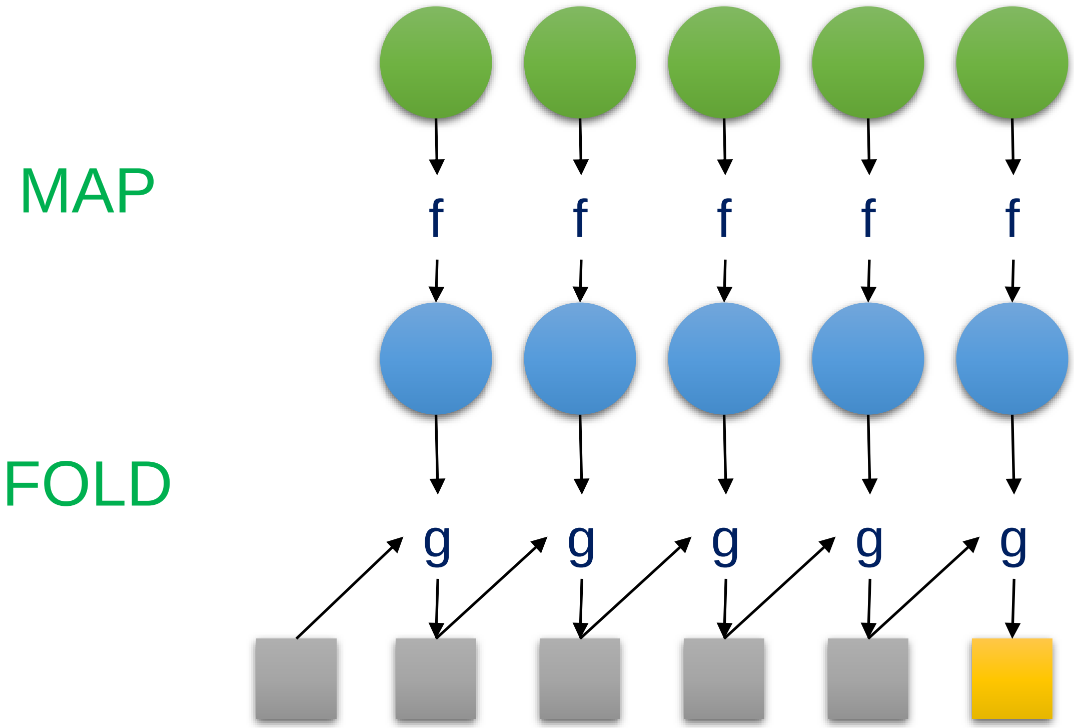
Generate final output

Key idea: provide a functional abstraction for these two operations

Roots in Functional Programming

MAP takes a function f and applies it to every element in a list,

FOLD iteratively applies a function g to aggregate results



Parallelization of Map and Reduce

The map operation (i.e., the application of *f* to each item in a list) can be parallelized in a straightforward manner, since each functional application happens in isolation

- In a cluster, these operations can be distributed across many different machines

The reduce operation has more restrictions on data locality

- Elements in the list must be "brought together" before the function *g* can be applied

However, many real-world applications do not require *g* to be applied to all elements of the list. If elements in the list can be divided into groups, the fold aggregations can proceed in parallel.

Disambiguation of MapReduce

"MapReduce is a programming model and an associated implementation for processing and generating large data sets.

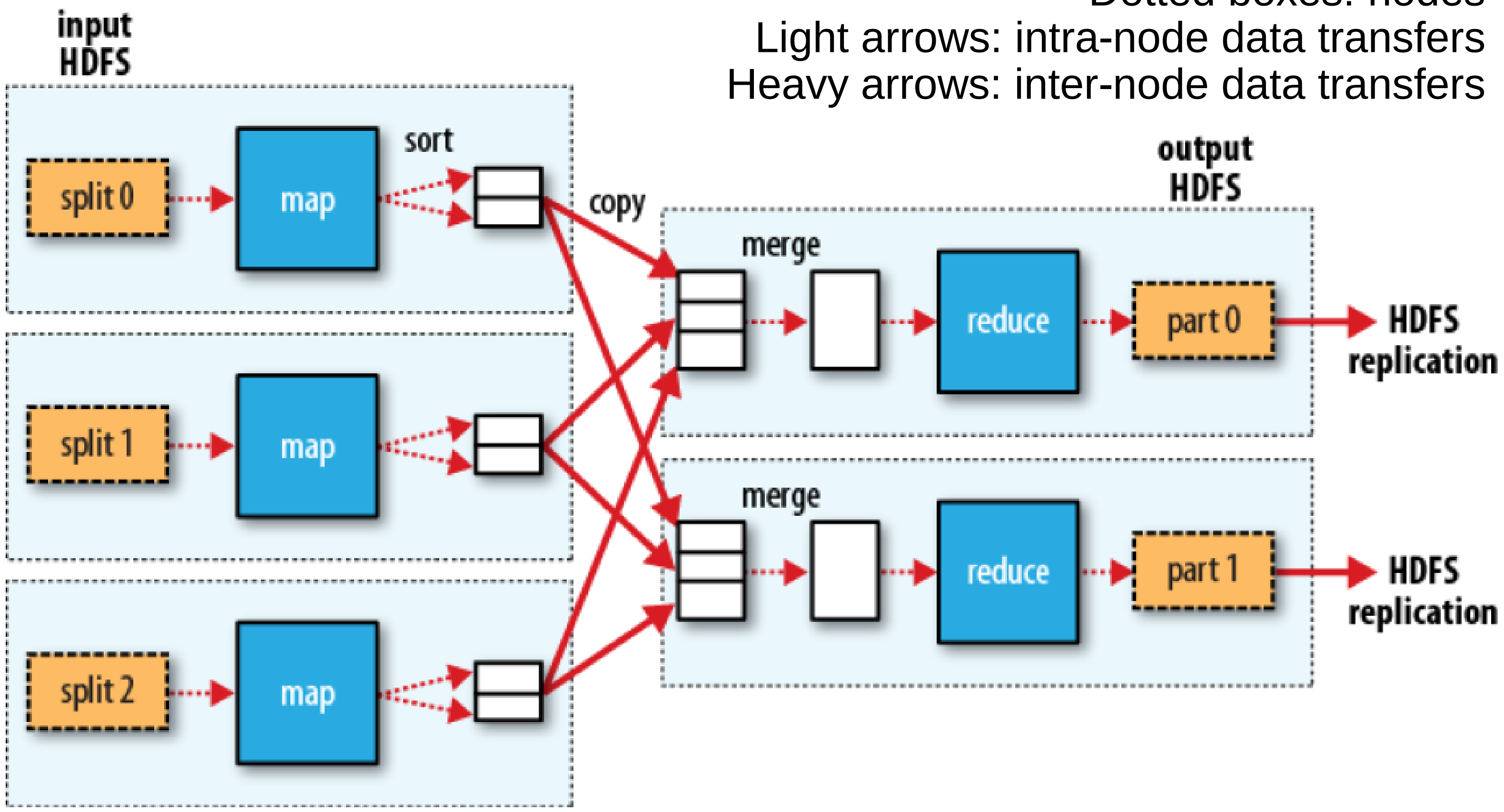
Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key."

-- Dean J., Ghemawat S. (Google)

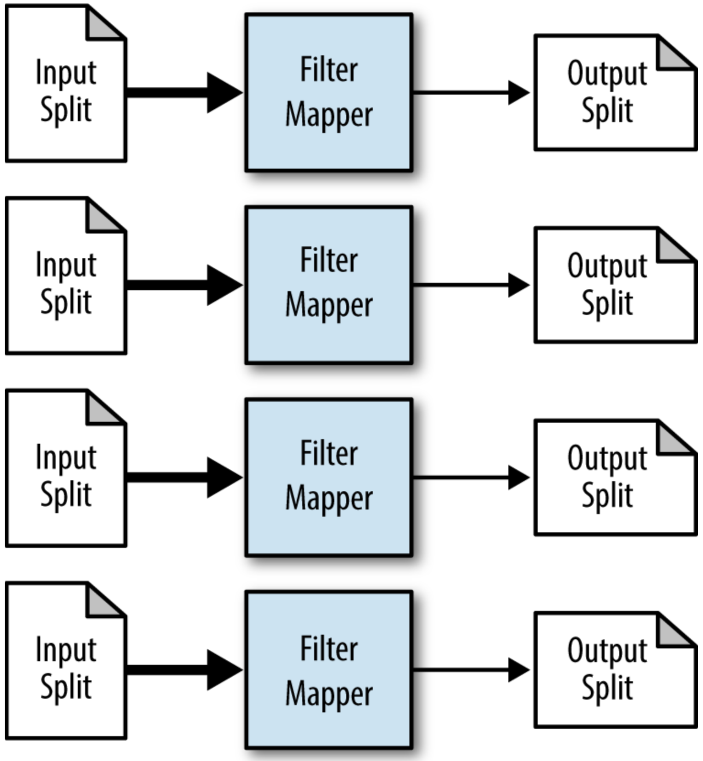
Hadoop MapReduce is an open-source implementation of the MapReduce programming model

MapReduce process

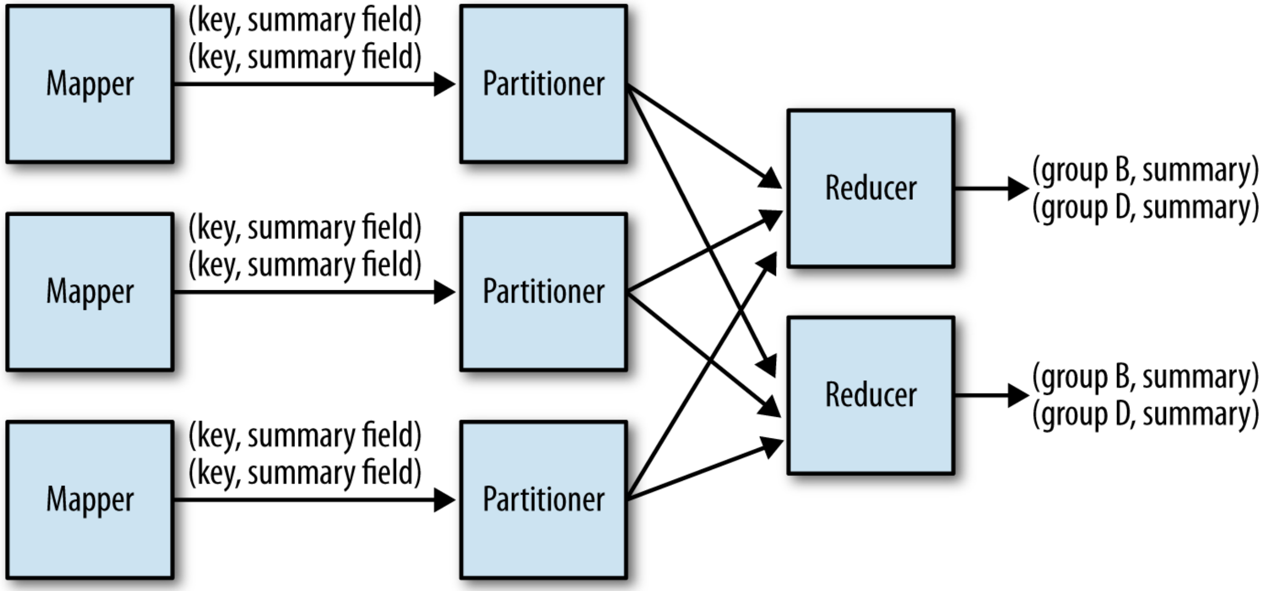
Dotted boxes: nodes
Light arrows: intra-node data transfers
Heavy arrows: inter-node data transfers



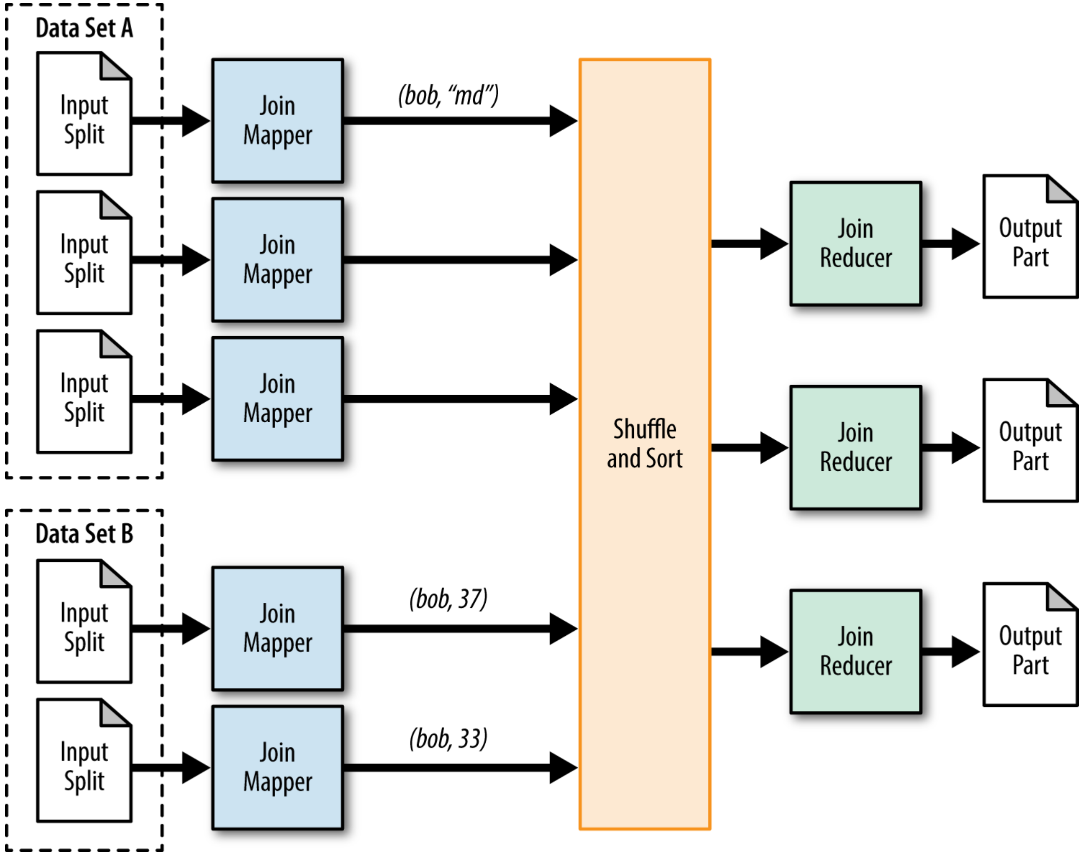
Filtering pattern



Summarization pattern



Join pattern



Sort

Goal: sort input

Examples:

- Return all the domains indexed by Google and the number of pages in each, ordered by the number of pages

The programming model (map, then reduce) does not support this per se

- But the implementations do: the shuffle stage performs grouping and ordering!

General pattern:

- map (key, record) → emit (sortKey, record)
- reduce (sortKey, records) → emit (sortKey, records[1]), ...

The Map and the Reduce do nothing

- With 1 reducer, we get sorted output
- With many reducers, we get partly sorted output (unless: TotalOrderPartitioner)

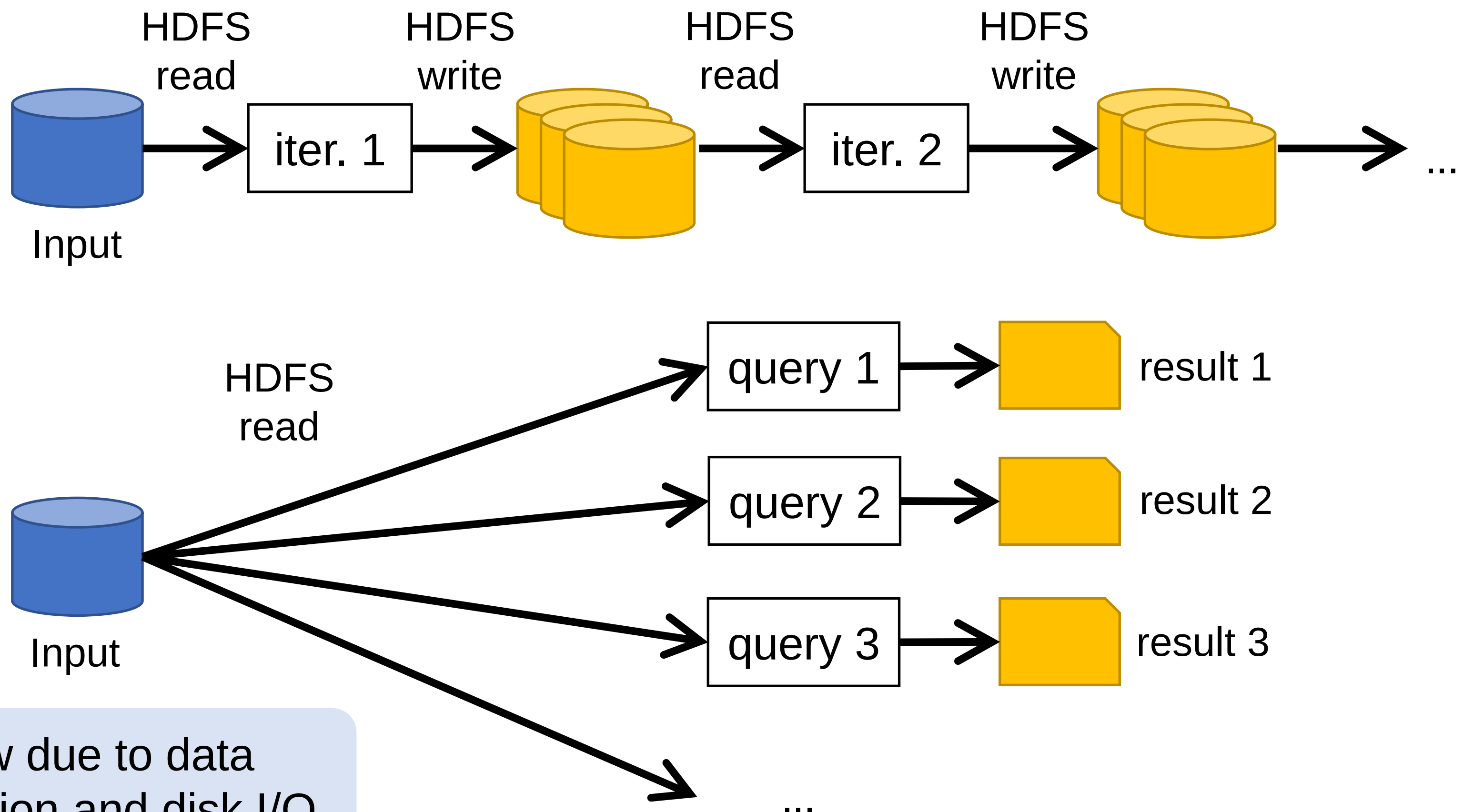
Two stage MapReduce

As map-reduce calculations get more complex, it’s useful to break them down into stages

- The output of the first stage serves as input to the next one
- The same output may be useful for different subsequent stages
- The output can be stored in the DFS, forming a materialized view

Early stages of map-reduce operations often represent the heaviest amount of data access, so building and saving them once as a basis for many downstream uses saves a lot of work!

Data sharing in MapReduce



Slow due to data replication and disk I/O

Spark

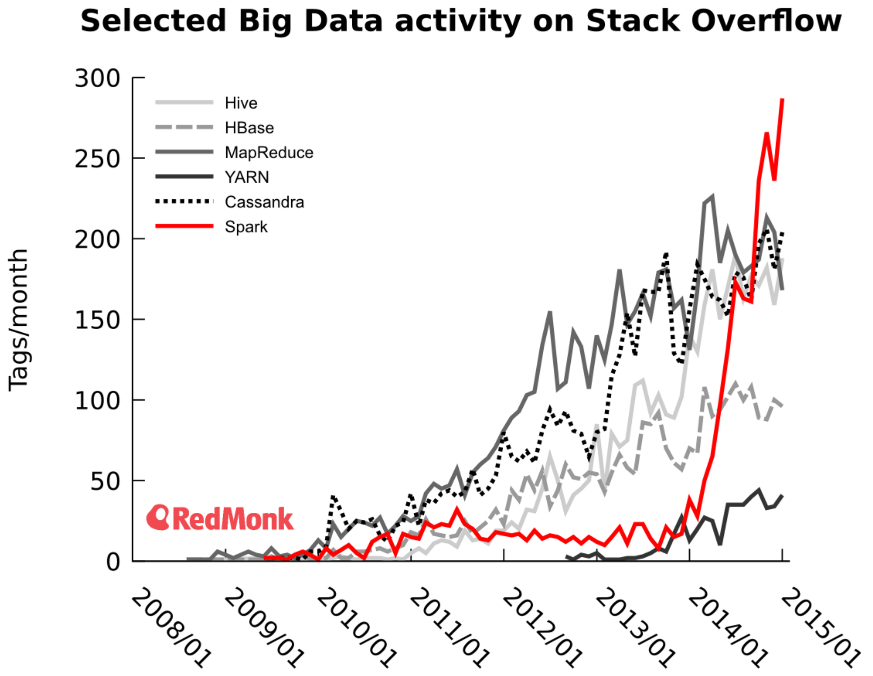
Spark project started in 2009

- Developed at UC Berkeley’s AMPLab by Matei Zaharia
- Originally built to test how Apache Mesos works

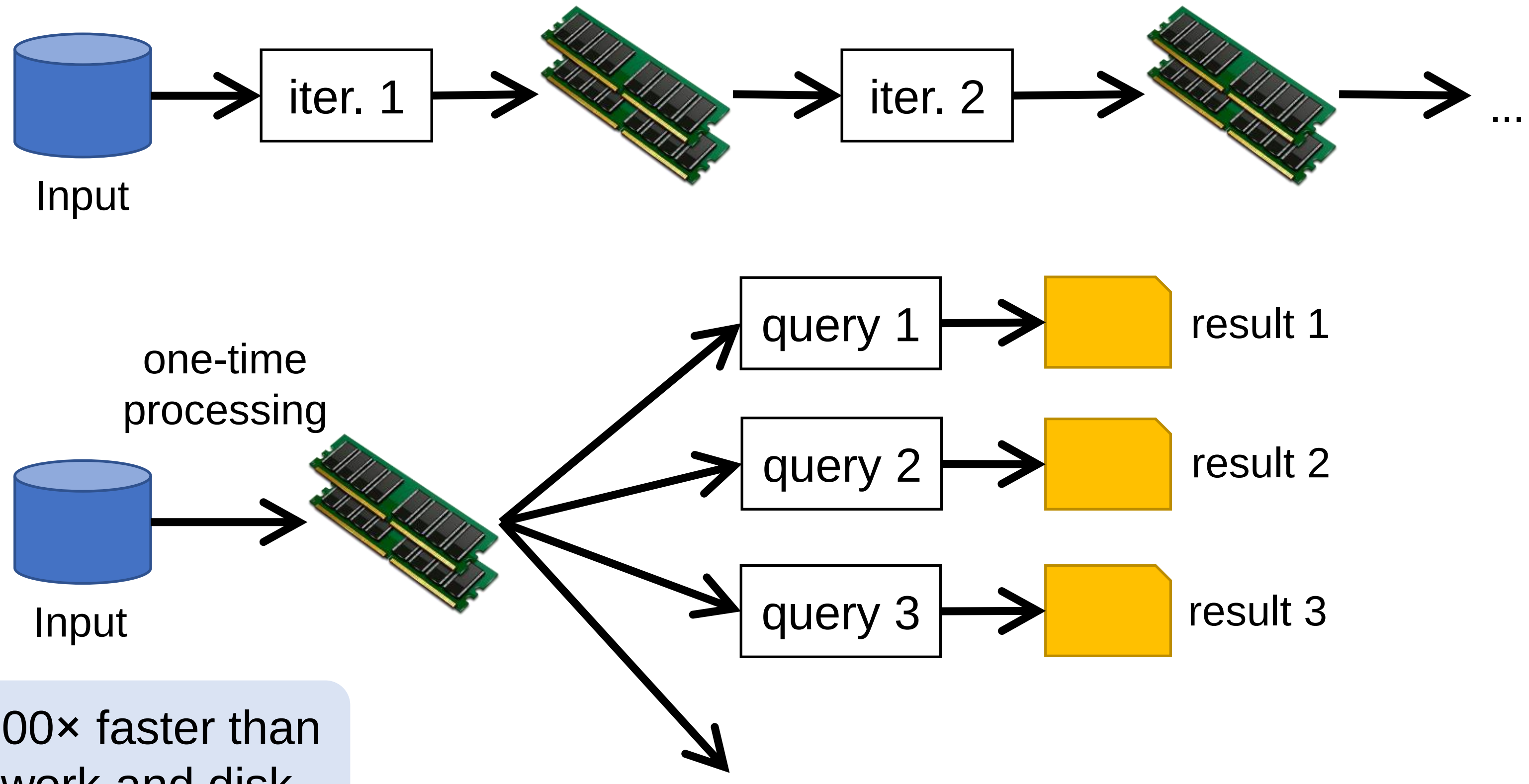
Open sourced 2010, Apache project from 2013

- In 2014, Zaharia founded Databricks
- Current version: 3.2
- Written in Scala; supports Java and Python

Currently the most used tool for batch analyses



Data sharing in Spark



10-100× faster than network and disk

Spark pillars: RDD

RDDs are immutable distributed collection of objects

- **Resilient**: automatically rebuild on failure
- **Distributed**: the objects belonging to a given collection are split into *partitions* and spread across the nodes
  - RDDs can contain any type of Python, Java, or Scala objects
  - Distribution allows for scalability and locality-aware scheduling
  - Partitioning allows to control parallel processing

Fundamental characteristics (mostly from *pure functional programming*)

- **Immutable**: once created, it can't be modified
- **Lazily evaluated**: optimization before execution
- **Cacheable**: can persist in memory, spill to disk if necessary
- **Type inference**: data types are not declared but inferred (≠ dynamic typing)

Spark pillars: DAG

Based on the user application and on the lineage graphs,
Spark computes a logical execution plan in the form of a DAG

- Which is later transformed into a physical execution plan

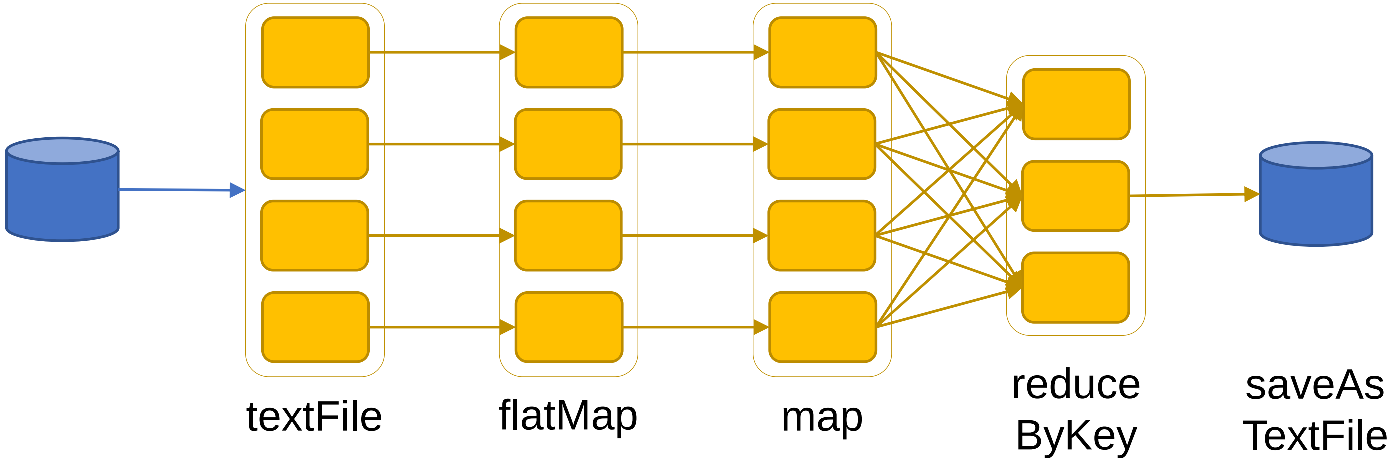
The DAG (Directed Acyclic Graph) is a sequence of computations performed on data

- Nodes are **RDDs**
- Edges are operations on RDDs
- The graph is Directed: transformations from a partition A to a partition B
- The graph is Acyclic: transformations cannot return an old partition

Spark sample DAG

Word count in Scala

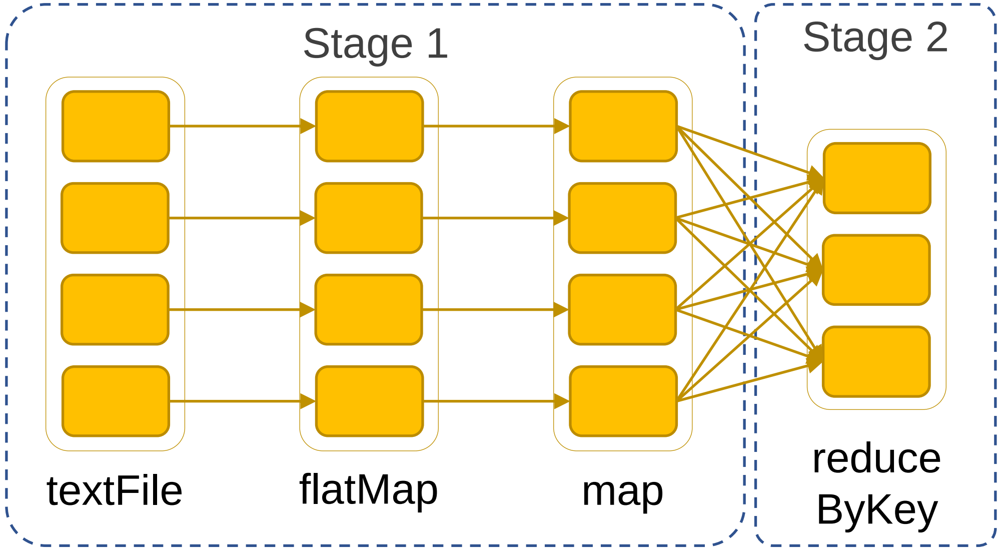
- textFile = sc.textFile("hdfs://...")
- counts = textFile.flatMap(line => line.split(" "))
  .map(lambda word: (word, 1))
  .reduceByKey(lambda a,b: a + b)
- **counts.saveAsTextFile("hdfs://...")**



DAG decomposed into stages

The execution plan is compiled into physical **stages**

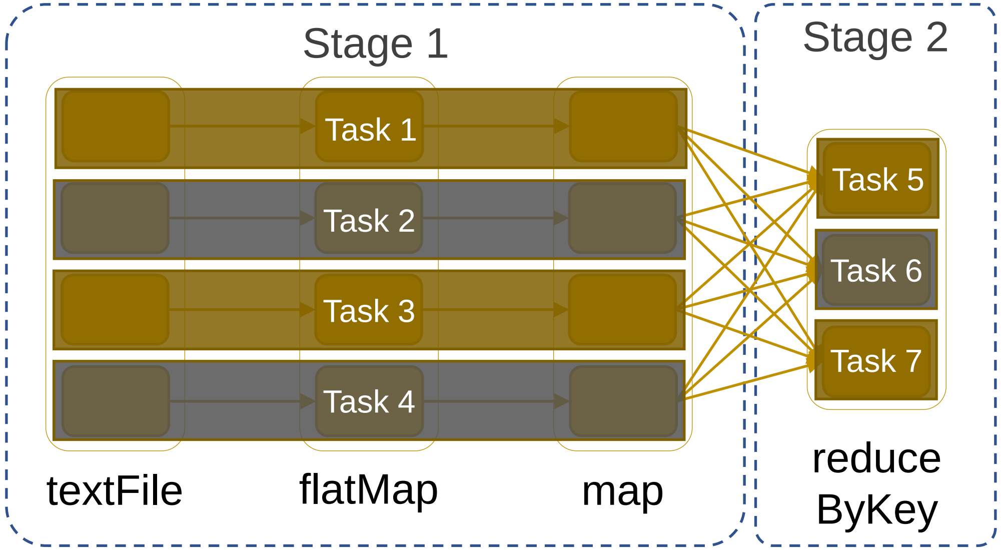
- Stages’ boundaries are defined by shuffle operations
- Operations with narrow dependencies are pipelined as much as possible



Stages decomposed into tasks

The fundamental unit of execution is the one of **tasks**

- A task is created for each partition in the new RDD
- Tasks are scheduled and assigned to the worker nodes based on data locality
- The scheduler can run the same task on multiple nodes in case of stragglers (i.e., slow nodes)

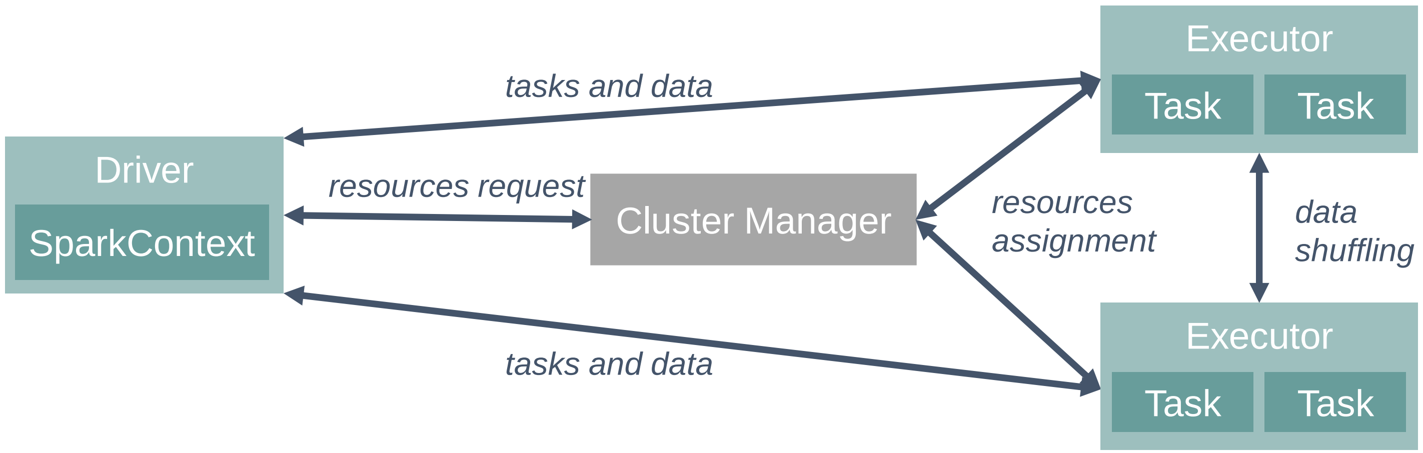


Spark architecture

Spark uses a *master/slave architecture* with one central coordinator (*driver*) and many distributed workers (*executors*)

- The driver and each executor are independent Java processes
- Together they form a Spark *application*

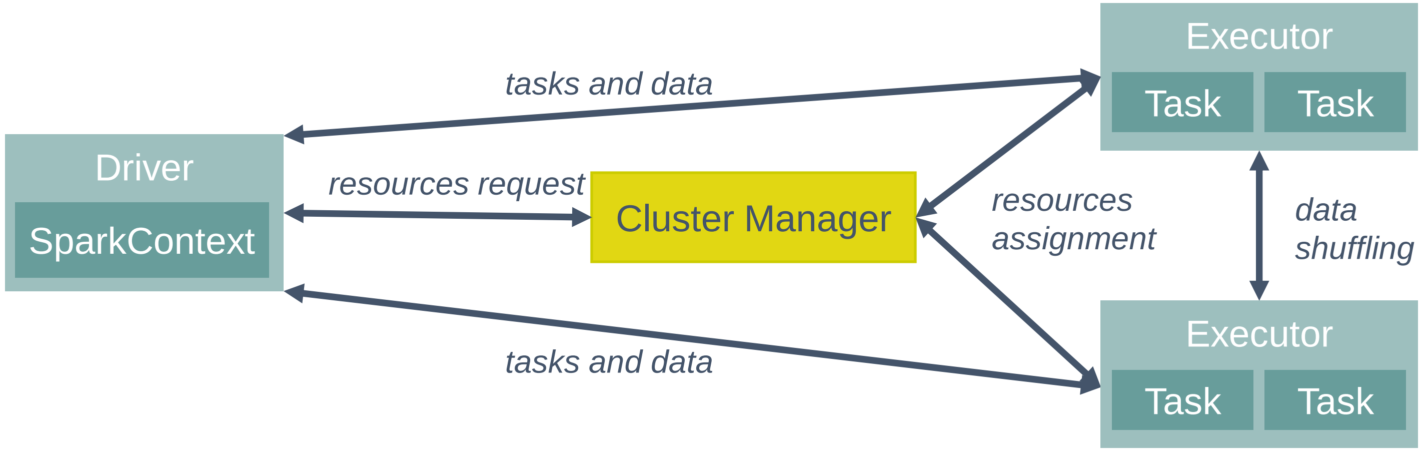
The architecture is independent of the cluster manager that Spark runs on



Spark architecture

Cluster Manager: the component responsible for assigning and managing the cluster's resources (e.g., memory, processor time, …)

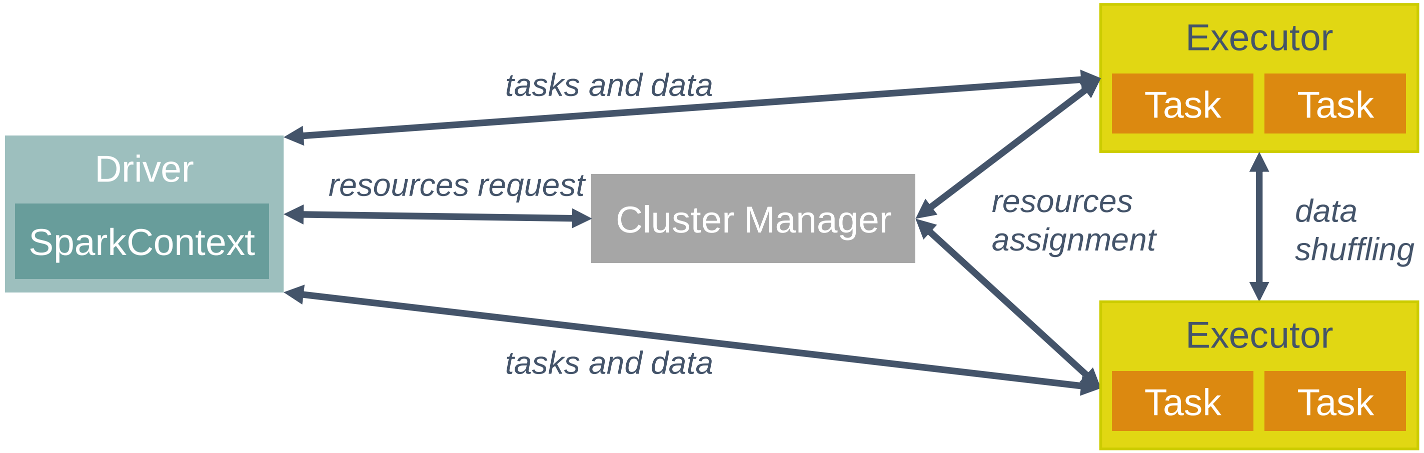
- The cluster manager can be either the Spark standalone manager or any other compatible manager (e.g., YARN)
- Launches executor processes on behalf of the driver



Spark architecture

Executor: a process responsible for executing the received tasks

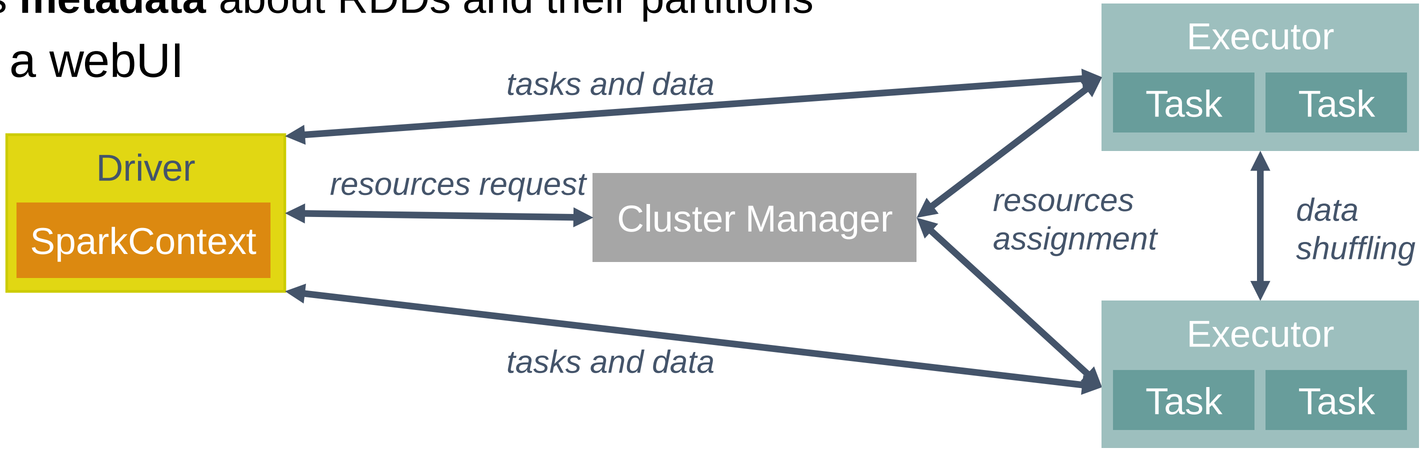
- Each spark application can have (and usually has) multiple executors, and each worker node can host many executors
- Typically runs for the entire duration of the application
- Stores (caches) RDD data in JVM heap
- Tasks are the smallest unit of work and are carried out by executors



Spark architecture

Driver Program (a.k.a. *Spark Driver*, or simply *Driver)*

- Each spark application can only have one driver (entry point of Spark Shell)
- Converts user program into tasks
  - Creates the **SparkContext**, i.e., the object that handles communications
  - Computes the logical **DAG** of operations and converts it into a physical **execution plan**
- Schedules tasks on executors
  - Has a **complete view** of the available executors and schedules tasks on them
  - Stores **metadata** about RDDs and their partitions
- Launches a webUI



Spark architecture (in YARN)

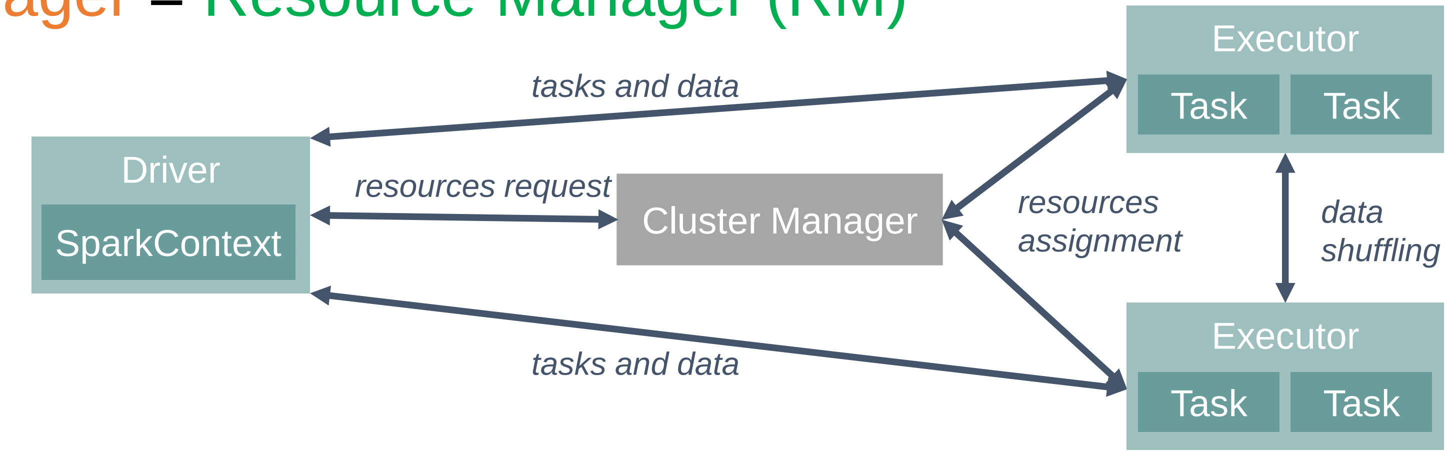
Driver Program ≅ Application

- Driver can be run *externally* from the client (e.g., spark-shell) or *internally* from the AMP (e.g., for production jobs)
- The Application master process (AMP) is not shown for simplicity

Executor = Container

- Executors are run and monitored by the Node Manager (NM)

Cluster Manager = Resource Manager (RM)



Interactive

Simple computations over small/large amounts of stored data

- Execution takes milliseconds to minutes

What we need

- An MPP-like execution engine

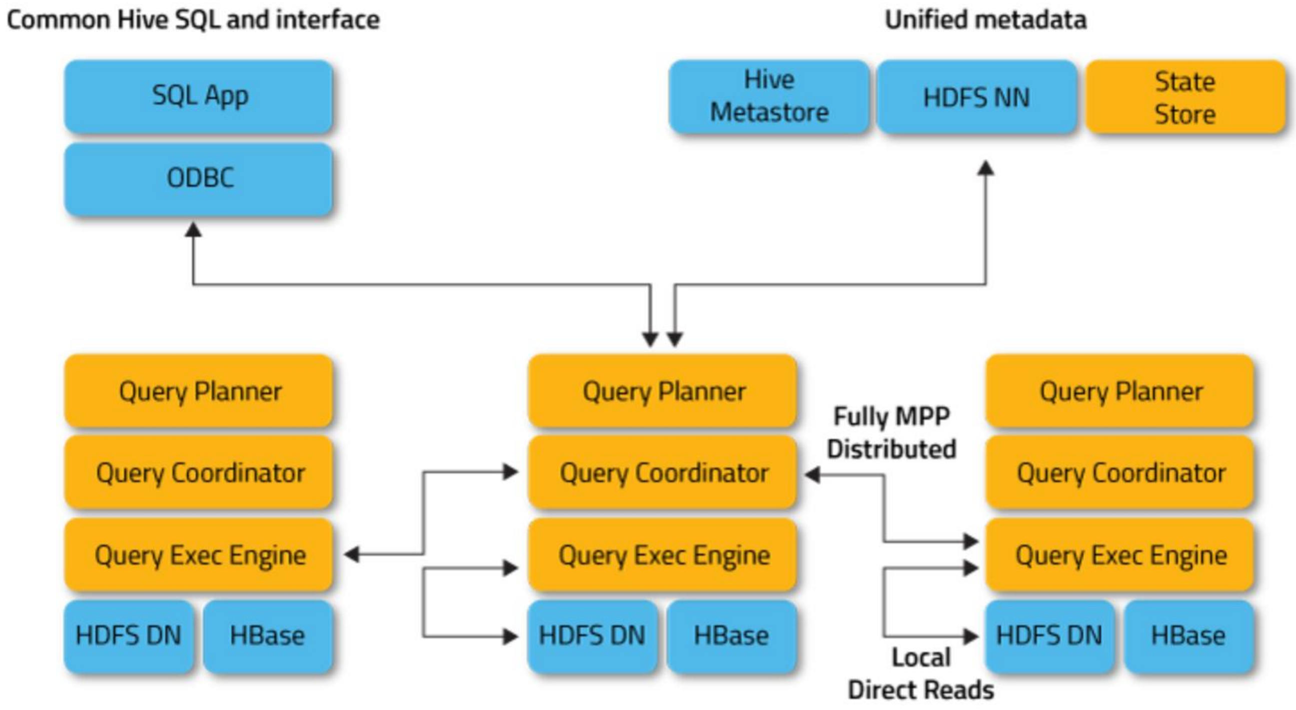
Interactive

MPP-like architecture

- Always-running daemons
- Distributed engine: submit queries to any node
- Highly rely on caching and in-memory computation
- Optimistic query execution

Several tools

- Apache Impala (from Cloudera)
- Apache Presto (from Facebook)
- Apache Drill



Streaming

Simple computations over small amounts of continuously incoming data

- Execution is in near-real-time

What we need

- A message queueing service
- An execution engine

Streaming system definition

The term "streaming" has been used to mean a variety of different things

- Video streaming
- Streaming systems
- Streaming algorithms

In our context, a *system for data streaming* is

***a type of* data processing engine**
that is designed with* infinite datasets *in mind

Remember that

- Batch engines can be (and have been) used to process infinite datasets
- Streaming engines can be (and have been) used to process finite datasets

Data streaming characteristics

Infinite dataset

- Data is always being generated
- No control over the order in which data elements arrive

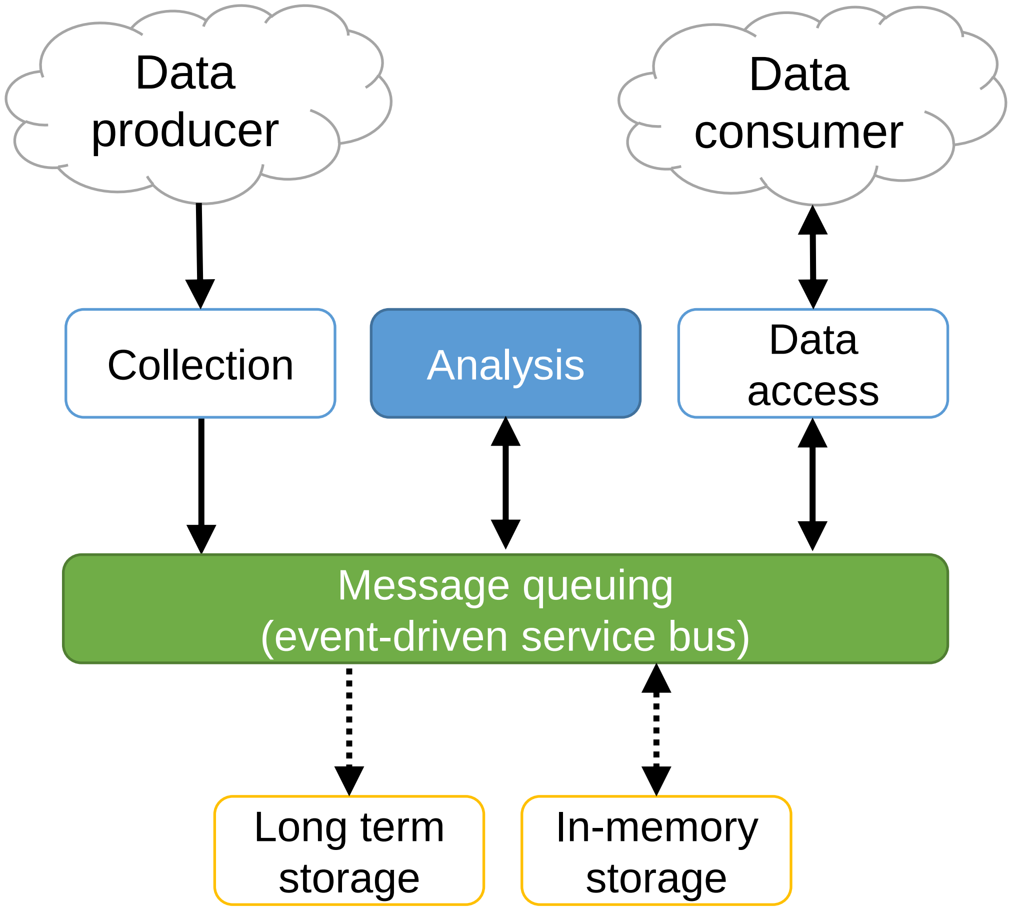
Infinite computation

- The system must be always on and must be able to keep up with the data
  - There must be a plan to avoid *overflowing* (e.g., auto-scalability)

Low-latency, approximate and/or speculative results

- Data can usually be processed a single time (*one pass*)
- Only a fraction of the dataset can be kept in memory for analyses
- Approximation may be required to accommodate the low-latency requirement

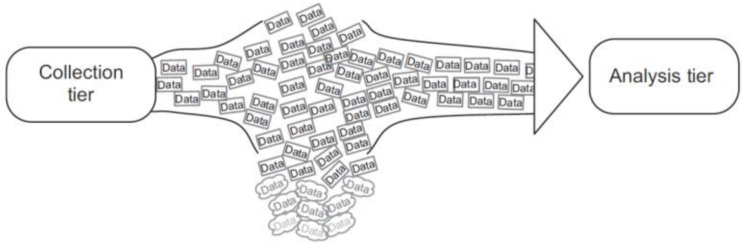
Architecture



Message queuing tier

The message queuing tier handles the transportation of data between different tiers

- Most importantly, between collection to analysis tiers



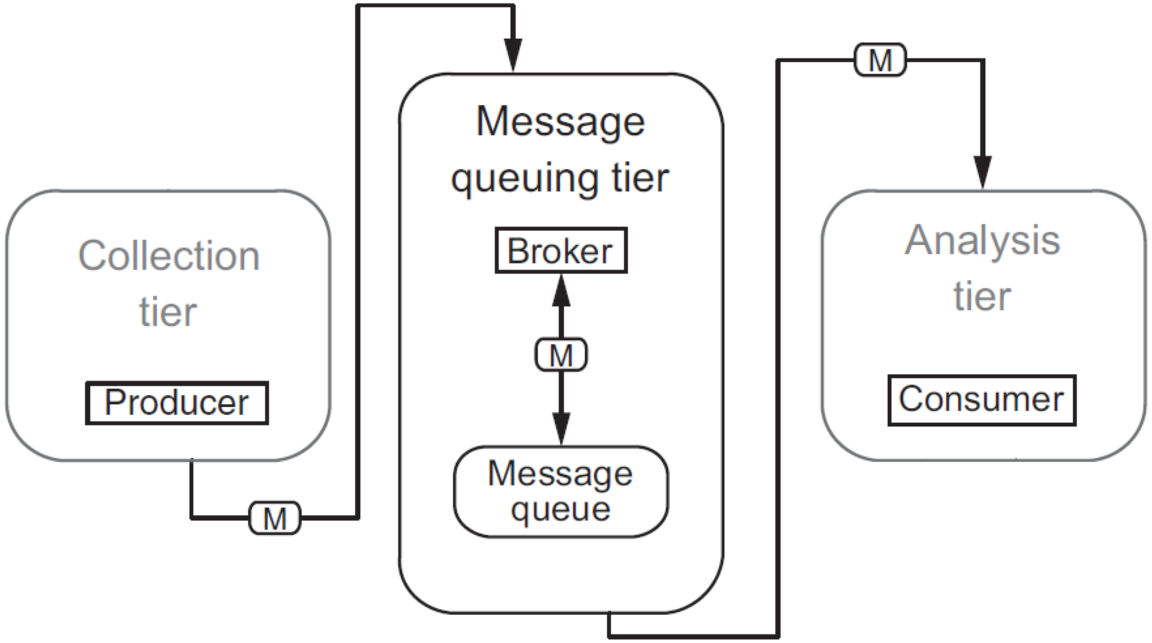
Why message queuing?

- By decoupling the pipeline of operations (collection, analysis, data access), each node in the cluster will do *one* job only
- Message queuing provides a solid framework for a safe communication between such nodes
- Message queuing handles funneling of *n* data streams to *m* consumers

Producer-Broker-Consumer

Core concepts

- The Producer and the Consumer of data
- The Broker, which manages one or more queues of data



Message delivery semantics

Exactly once: a message is never lost and is read once and only once

- Required in applications where data means money (financial/ad systems)
- Performance is sacrificed to provide safety mechanisms

At most once: a message may get lost, but it will never be read twice

- Allowed where not all data is required (monitoring systems, down-sampling)
- Fast and trivial

At least once: a message will never be lost, but it may be read twice

- Balances the two previous semantics
- Easier to provide than *exactly once*
- The consumer can still check for duplicates to adopt *exactly once*

Beware: guarantees depends on the chosen tools + application logic on the whole pipeline

Analysis: continuous queries

A continuous query is a query that is issued once and then is continuously executed against the data

- In contrast, traditional queries are simply executed once when issued
- A continuous query may need to maintain a **state**

State: an intermediate result that is continuously updated by the query

- A query is *stateless* if each execution is independent from the other

What makes continuous queries different from traditional ones?

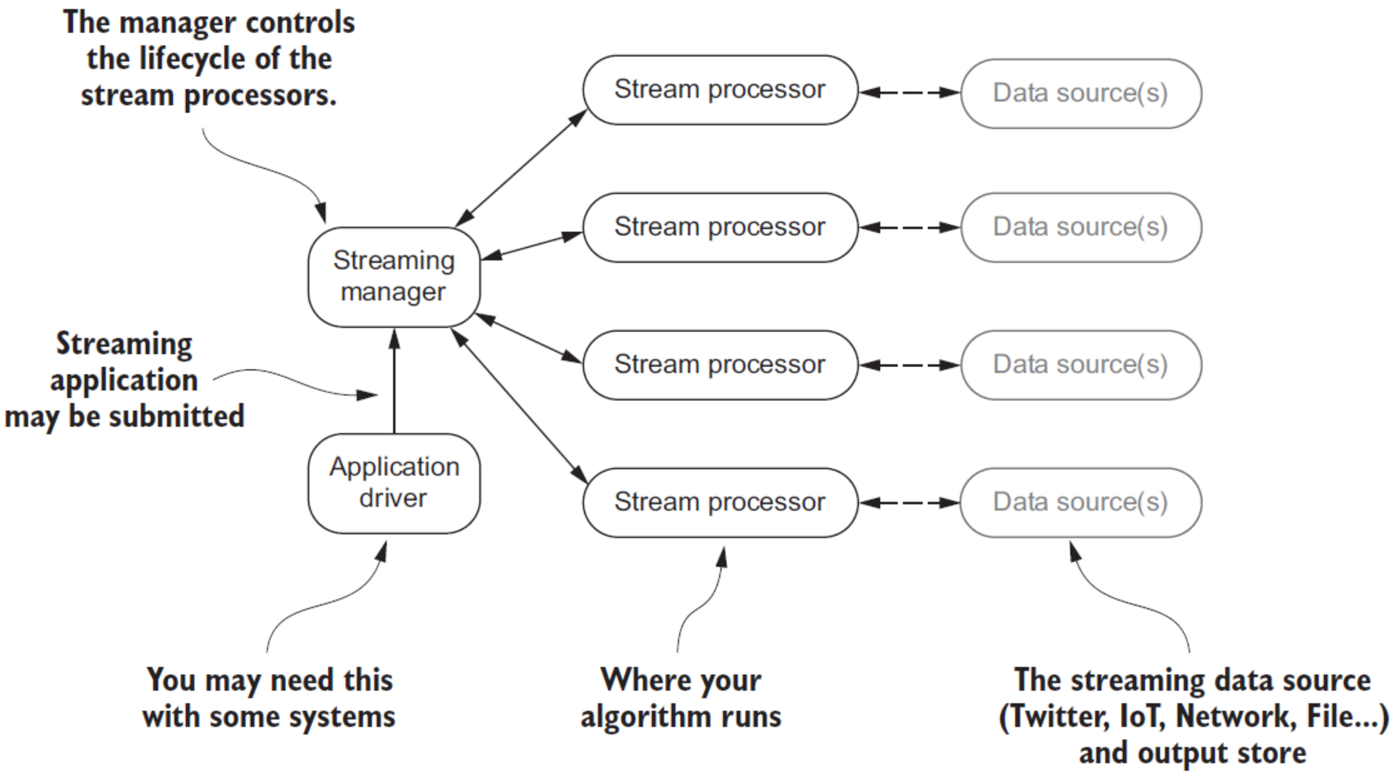
- Memory constraint (for both data processing and state maintenance)
  - Data cannot be processed altogether (***one pass* algorithms**)
  - Not much space to store the state
- Time constraint
  - Data that can't be processed in time may have to be dropped (***load shedding***)
  - Algorithms (e.g., predictive models) may lose efficacy over time (***concept drift***)

Continuous query model

	Typical query	Continuous query
Query model	Queries are based on a one-time model and a consistent state of the data **Pull model**: the user executes a query and gets an answer, and the query is forgotten	The query is continuously executed based on the data that is flowing into the system **Push model**: a user registers a query once, and the results are regularly pushed to the client
Query state	If the system crashes while a query is being executed, it must be re-issued on the whole dataset	Registered continuous queries may or may not need to continue where they left off; in the first case, a **state** has to be maintained

Distributed execution

An architecture that is common to different frameworks


The manager controls the lifecycle of the stream processors.
Stream processor
Data source(s)
Stream processor
Data source(s)
Streaming manager
Stream processor
Data source(s)
Streaming application may be submitted
Application driver
Stream processor
Data source(s)
You may need this with some systems
Where your algorithm runs
The streaming data source (Twitter, IoT, Network, File...) and output store

Sliding windows

Sliding windows define length and period in terms of stream time

- Beware of the difference between event time and stream time

Depending on the comparison of length and period, specialized versions of sliding windows can be defined

- Fixed windows: when length and period are the same
  - E.g.: analyze the last 5 minutes of data every 5 minutes
  - *Tumbling windows* are a special case of fixed windows, where length and period are expressed in terms of number of items (instead of time)
- Overlapping windows: when the length is greater than the period
  - E.g.: analyze the last 5 minutes of data every 2 minutes
- Sampling windows: when the length is smaller than the period
  - E.g.: analyze the last 2 minutes of data every 5 minutes

Data-driven windows

Data-driven windows define the length in terms of the content that comes with the data

- In this case, the period determines the *update interval*

Typical use case: sessions

- A session is a sequence of events terminated by a gap of inactivity grater than some timeout
  - Goal: determine the average amount of traffic generated by sessions

Main characteristic: the lengths cannot be defined apriori

- How can I know when the session of a user has ended?
- How can I know when the event of a user's session are going to stop coming?

Algorithms

Consider *n* the space of events captured by the data stream

Streaming algorithm have the following requirements:

- *One-pass*; once examined, items must be discarded
- Use small space for the internal state: (O(polylog(n)))
- Fast update of the internal state: O(1) to O(polylog(n))
- Fast computation of answers
- Provide approximated answers with (ε, δ)-guarantees
  - Let R be the exact result, ε and δ > 0
  - With probability at least $1 − δ$, the algorithm outputs R' such that $(1 − ε)R ≤ R' ≤ (1 + ε)R$
  - For instance, take ε=0.02 and δ=0.01
  - Then, I have a 99% probability that the obtained result R’ equals the real result ± 2%

Algorithms (examples)

Given any list as an input

- Count the number of elements
- Find the *n*th element

Given a list of numbers

- Find the *k* largest or smallest elements (*k* given in advance)
- Find the sum/mean/variance/st.dev. of the elements of the list

Given a list of symbols from an alphabet of k symbols (given in advance)

- Count the number of times each symbol appears in the input (*frequency*)
- Find the most or least frequent elements (*heavy hitters*)
- Sort the list according to some order on the symbols
- Find the maximum gap between two appearances of a given symbol

Algorithms (examples)

Some problems are **not** solvable by one-pass algorithms

Given any list as an input

- Find the middle element of the list

Given a list of numbers

- Find the median
- Find the most frequent symbol
- Sort the list

Approximated algorithms

Analyzing a data stream is challenging due to:

- Space constraints
- Time constraints
- Algorithm unfeasibility in one-pass

One solution is to rely on *approximated algorithms*

Approximation can be achieved by relying on two main concepts:

- Sampling
- Random projections

Approximation by sampling

Many different sampling methods have been proposed

- *E.g., Distinct sampling, Quantile sampling, Reservoir sampling*

Sampling algorithms are known on time series and cash register models to

- Find the number of distinct items
- Find the quantiles
- Find frequent items

Notes

- Some systems (e.g., IP packet sniffers) already do sampling
- Sampling is not a powerful primitive for many problems
    - Too many samples for performing sophisticated analyses
- Sampling is more challenging in the turnstile model

Approximation with random projections

An approach that relies on dimensionality reduction, using projection along random vectors

- Project high-dimensional data into a suitable lower-dimensional space..
- ..in a way which approximately preserves the distances between the points
- Projections are called *sketches*

Random projection algorithms are known on (also) turnstile models to

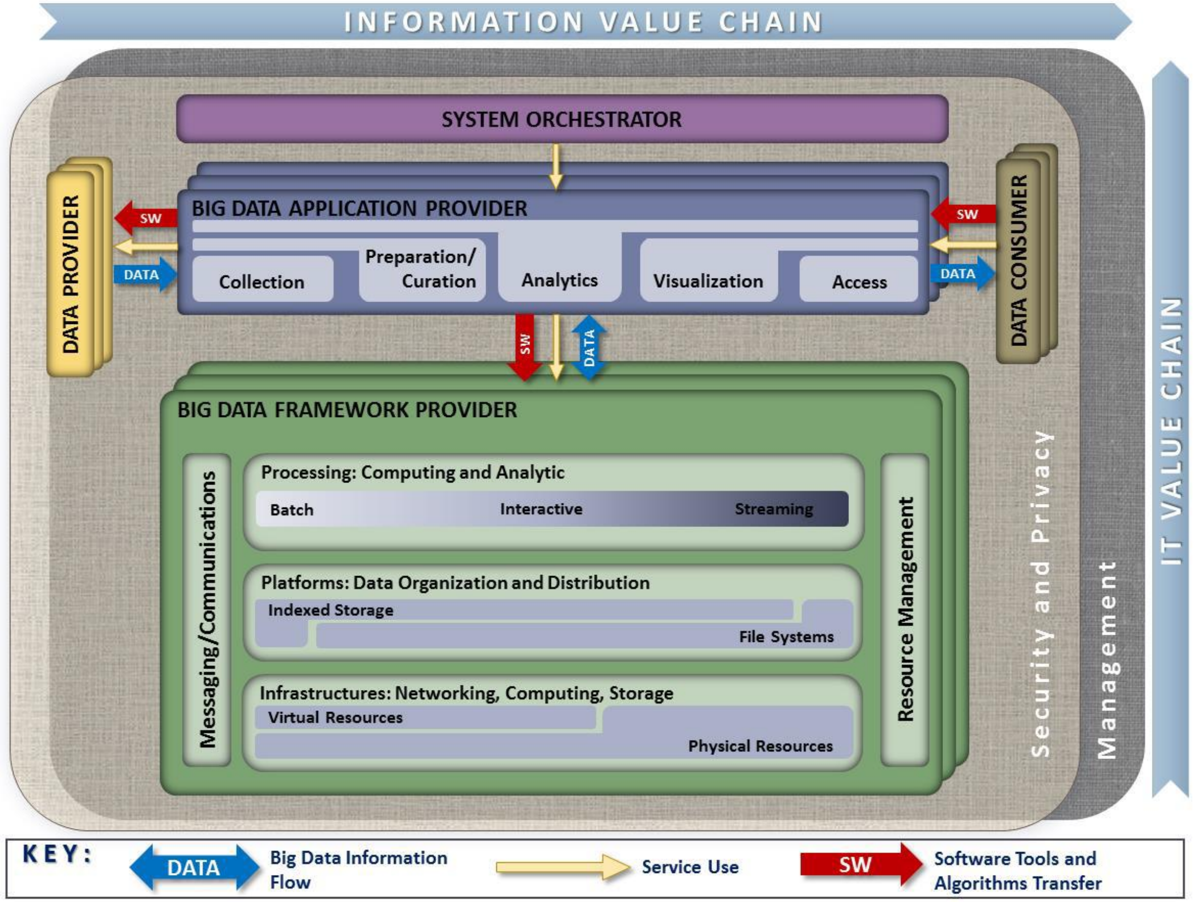
- Estimate the number of distinct elements at any time
- Estimate the quantiles at any time
- Track most frequent items, wavelets, histograms, etc.
- Random subset sums, counting sketches, Bloom filters

Reference architectures

A stack of many components

- NIST’s reference
- Microsoft’s reference
- Coexistence of batch and streaming (Lambda vs Kappa)
- The technological stack

NIST's reference



NIST's reference

System Orchestrator

- **Integrate the required application activities into an operational vertical system**
- Configure and manage the other components of the Big Data architecture to implement one or more workloads
- Monitor workloads and system to verify the meeting of quality requirements
- Elastically assign and provision additional physical or virtual resources
- Performed by human and/or software components

NIST's reference

Data Provider

- **Introduces new data or information feeds into the Big Data system**
- It can be anything from a sensor to a human or another Big Data system
- Includes the following activities:
  - Collecting and persisting the data
  - Providing transformation functions for scrubbing sensitive information
  - Creating the metadata describing the data source, usage policies/access rights, etc.
  - Enforcing access rights on data access and establishing (in)formal contracts for data access authorizations
  - Making the data accessible through suitable programmable push or pull interfaces and mechanisms
  - Publishing the availability of the information and the means to access it

Data Consumer

- **Uses the interfaces or services provided by the Big Data Application Provider to get access to the information of interest**
- It can be an actual end user or another Big Data system

NIST's reference

Big Data Application Provider

- **Executes a specific set of operations along the data life cycle**
  - Meet the requirements established by the System Orchestrator
  - Meet security and privacy requirements
- Includes the following activities:
  - Collection, Preparation, Analytics, Visualization, Access
- Each activity
  - Is specific to the application
  - Can be implemented by independent stakeholders and deployed as a stand-alone service
- There may be multiple and differing instances of each activity, or a single program may perform multiple activities
- Each of the functions can run on a separate Big Data Framework Provider or all can use a common Big Data Framework Provider

NIST's reference

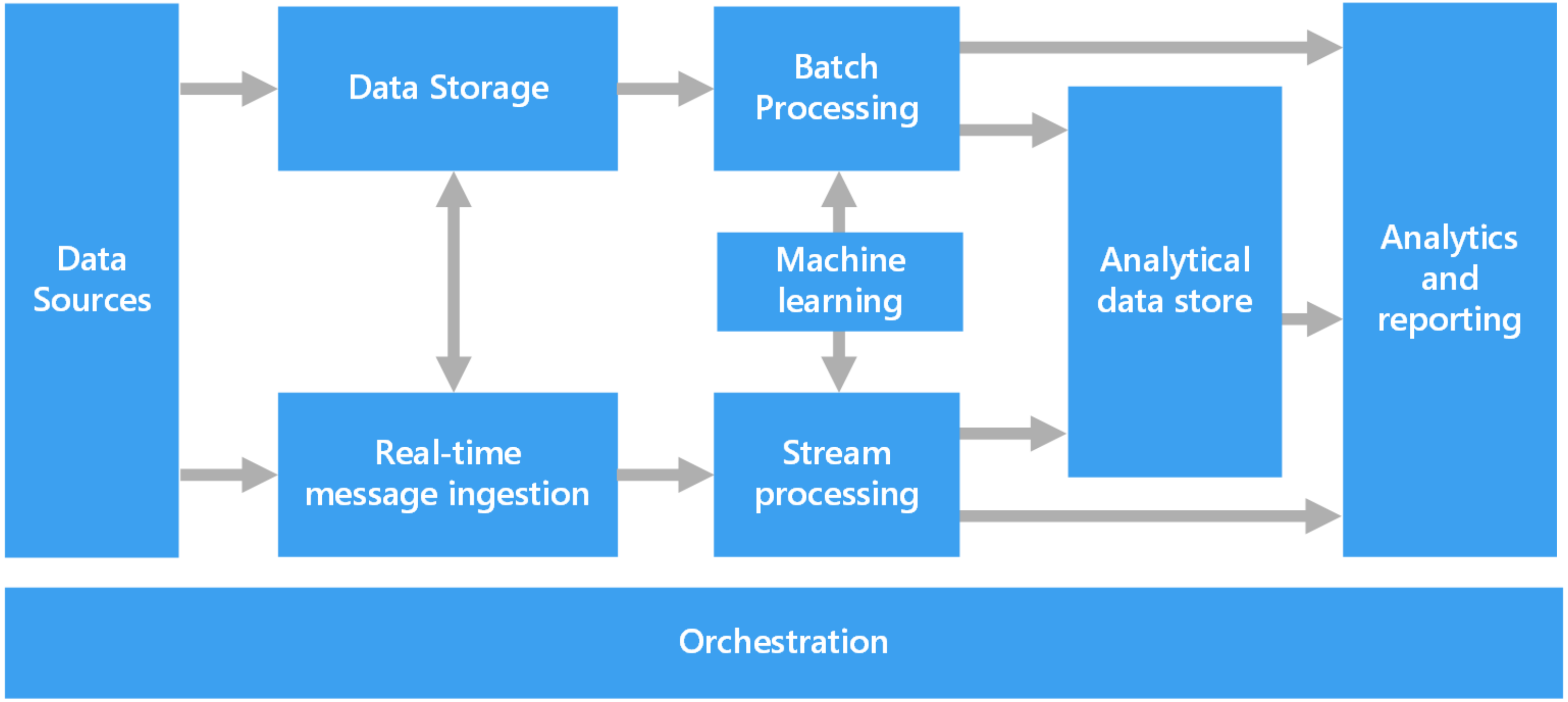
Big Data Framework Provider

- **Provides general resources or services to be used by the Big Data Application Provider in the creation of the specific application**
  - Handles the communication with the Big Data Application Provider and the assignment of physical resources to the respective activities
- Consists of Infrastructure frameworks..
  - Support the underlying computing, storage, and networking functions required to implement the overall system
  - Associated with physical or virtual infrastructure resources
- .. Data Platform frameworks..
  - Manage the organization and distribution of data (file System, databases, etc.)
- .. and Processing frameworks
  - How data will be processed in support of Big Data applications (e.g., batch/streaming framework)

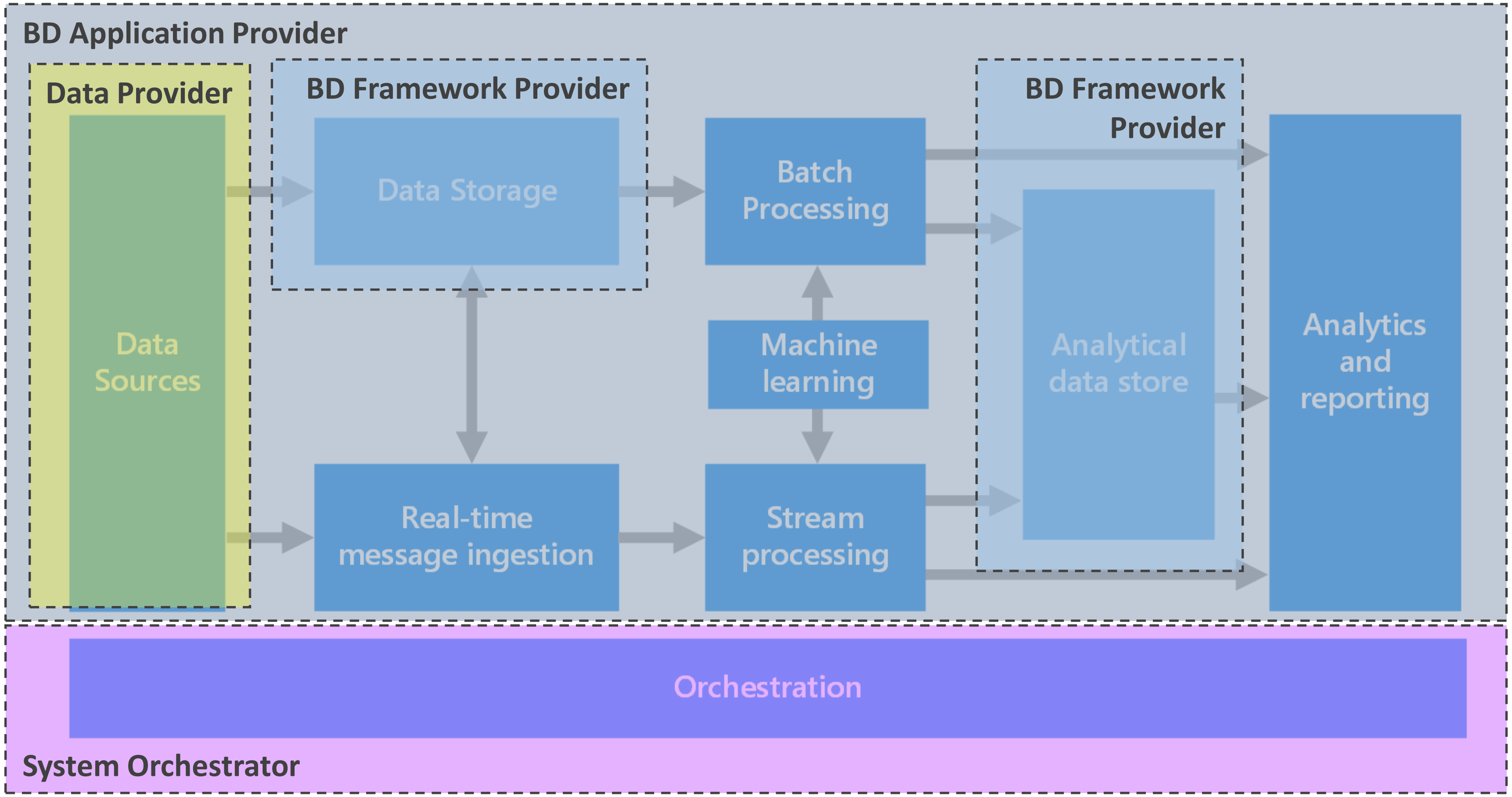
Microsoft's reference

Logical components that fit into a big data architecture

- Individual solutions may not contain every item



Microsoft's reference



Microsoft's reference

Data sources

- Data stores, static files, real-time sources

Data storage

- Distributed file store (data lake) or database

Batch processing

- Long-running batch jobs to filter, aggregate, and prepare the data for analysis

Analytical data store

- Serve the processed data in a structured format that can be queried using analytical tools

Analysis and reporting

- Traditional OLAP and BI tools, interactive exploration, analytical notebooks

Microsoft's reference

Real-time message ingestion

- Capture and store real-time messages for stream processing
- Act as a buffer for messages
- Support scale-out processing, reliable delivery, message queuing semantics

Stream processing

- Filter, aggregate, and prepare the data for analysis
- Processed stream data is then written to an output sink

Orchestration

- Automation of repeated data processing operations, encapsulated in workflows
- E.g., transforming source data, moving data between multiple sources and sinks, loading into an analytical data store, pushing results to a dashboard

Lambda vs Kappa

Batch and streaming analyses follow different principles

- Batch: large amounts of data at-rest with fault-tolerance
- Streaming: small amount of continuously incoming data without fault-tolerance

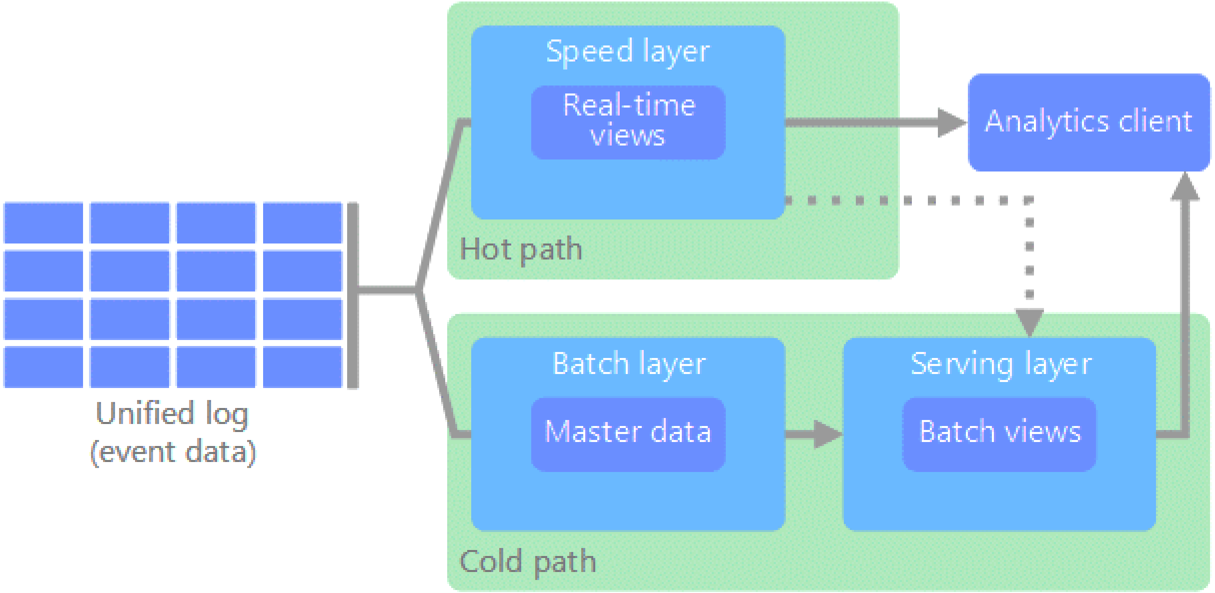
Two philosophies to make them coexist

- Lambda
- Kappa

Lambda architecture

All data coming into the system goes through two paths

- *Hot path*: for timely, yet potentially less accurate data in real time
- *Cold path*: for less timely but more accurate data



Lambda architecture

Pros of Lambda

- Emphasizes retaining the input data unchanged
- Highlights the problem of reprocessing data (bugs, evolution, etc.)

Cons of Lambda

- Parallel development and maintenance of two parallel pipelines
- Same goal, different languages, different frameworks

Streaming: a superset of batch

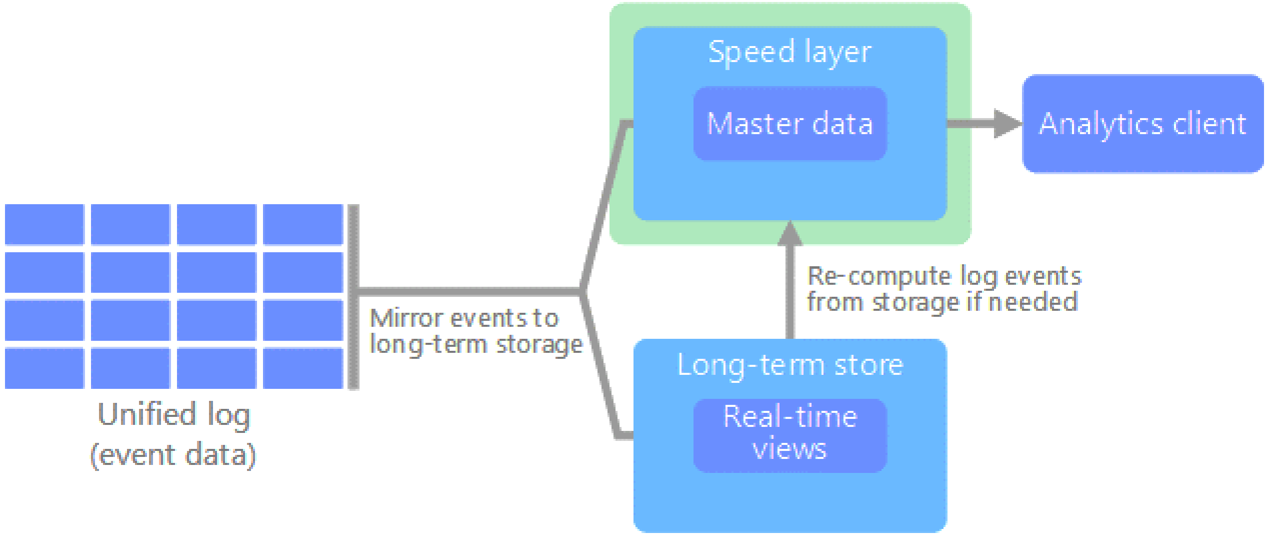
Can we do better?

- Well-designed streaming systems provide a strict superset of batch functionality
- Introduce correctness (exactly-once semantics, strong consistency) and the result of a streaming job is the same of batch job
- A streaming engine can handle a bounded dataset

Kappa architecture

All data flows through a single path, using a stream processing system

- Similar to a lambda architecture's speed layer, all event processing is performed on the input stream and persisted as a real-time view
- To recompute the entire data set (equivalent to what the batch layer does in lambda), simply replay the stream



Kappa architecture

~~Modeling everything under the streaming paradigm is not trivial~~

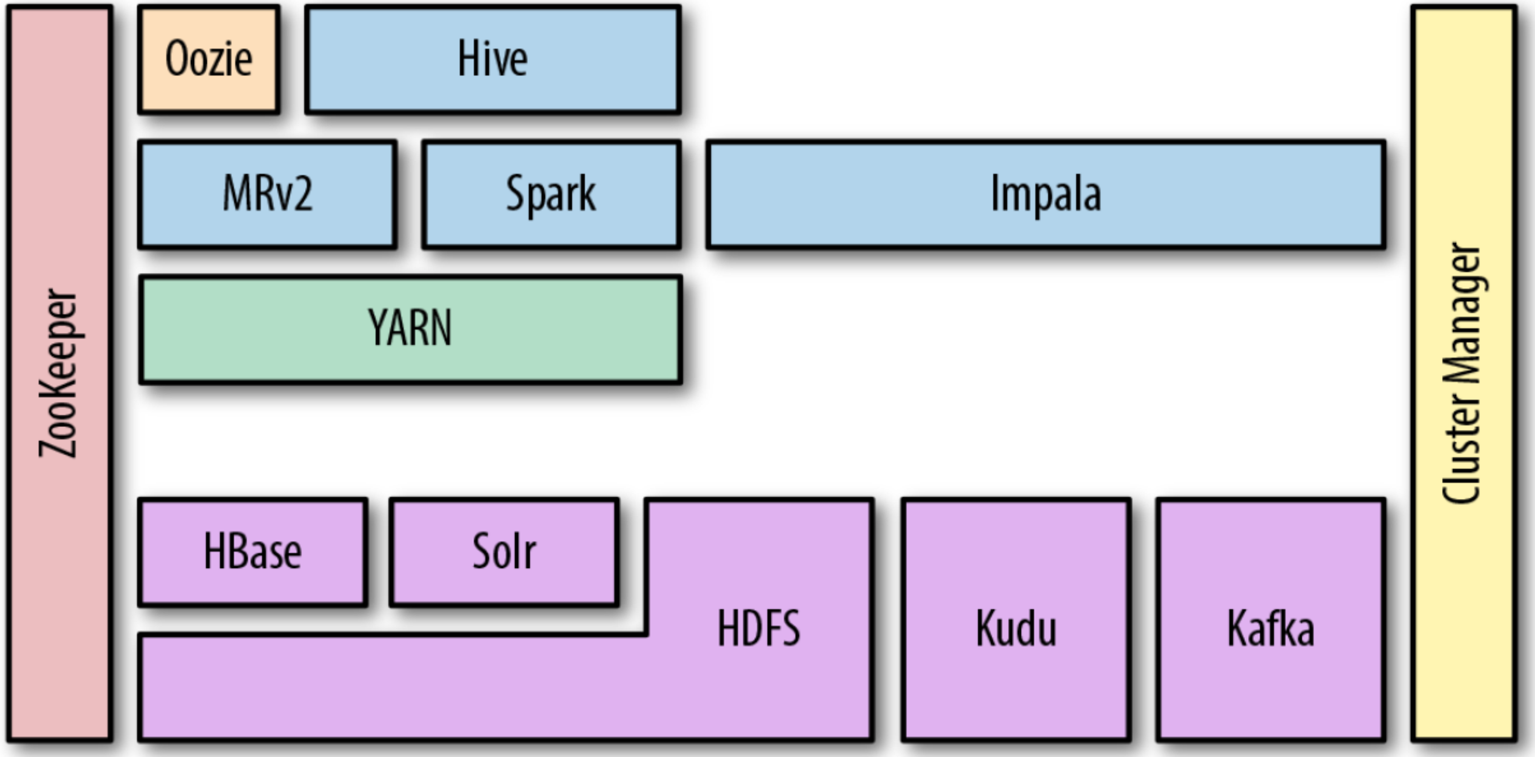
Advanced tools provide a single framework and unified APIs for writing and executing both batch and streaming jobs

- Google Cloud Dataflow
- Apache Flink

“Broad maturation of streaming systems combined with robust frameworks for unbounded data processing will in time allow for the relegation of the Lambda Architecture to the antiquity of big data history where it belongs”

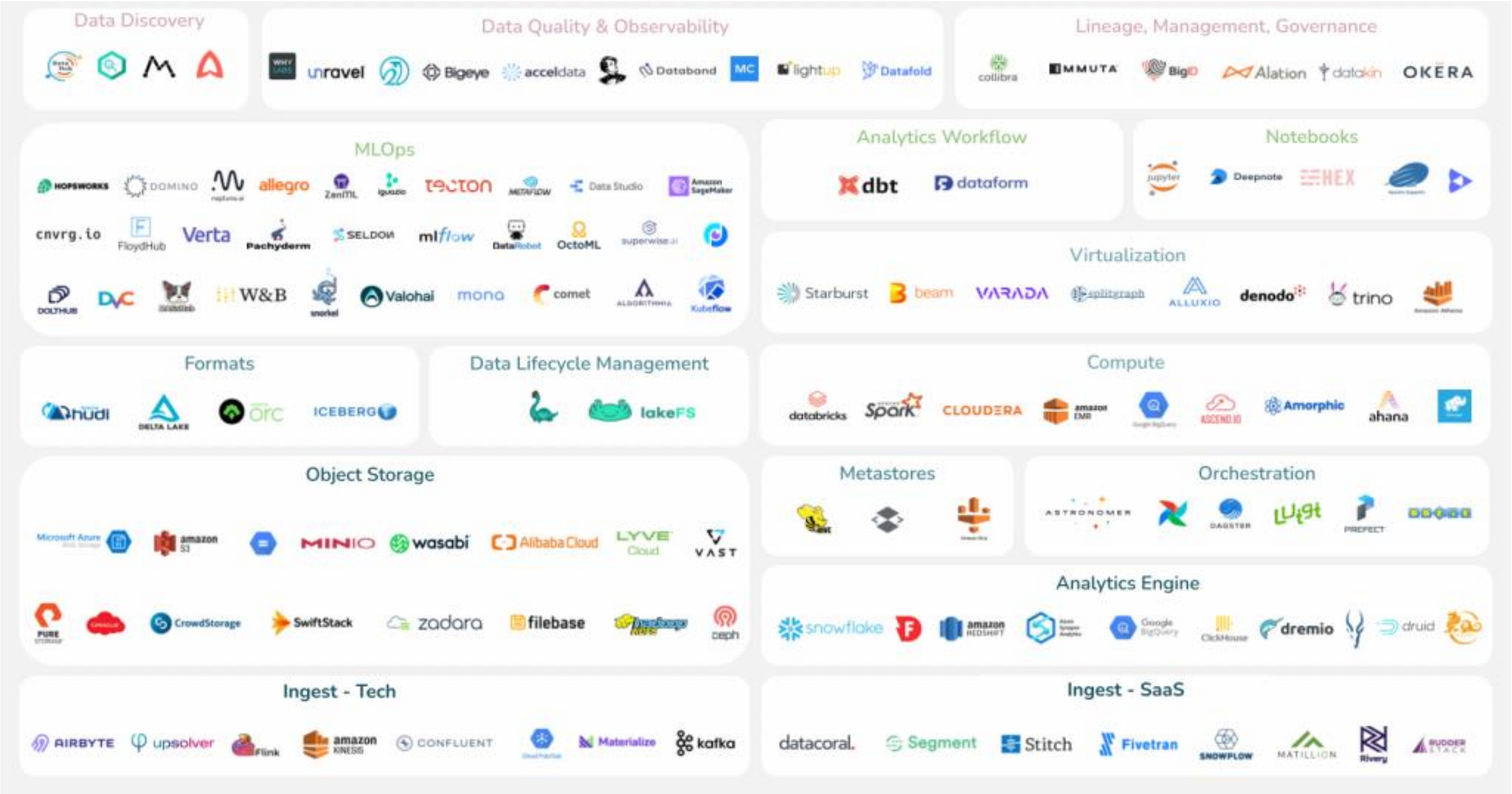
The technological stack

Sample Hadoop-based deployment



The technological stack

Beyond Hadoop



On-premises vs cloud

First installations where primarily on-premises

- **Installation**: how do I set up a new machine?
- **Networking**: how do I cable dozens of machines?
- **Management**: how do I replace a broken disk?
- **Upgrade**: how do I extend the cluster with new services/machines?
- (energy and cooling, software licenses, insurance...)

Today, cloud providers are mature and offer comprehensive support for big data ecosystems

- Forget all that is above

Cloud computing

A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, services) that can be rapidly provisioned and released with minimal management effort or service provider interaction

- On-demand self-service (consume services when you want)
- Broad network access (consume services from anywhere)
- Broad network deployment (deploy services anywhere)
- Resource pooling (infrastructure, virtual platforms, and applications)
- Rapid elasticity (enable horizontal scalability)
- Measured service (pay for the service you consume as you consume)
- Reliability (fault-tolerance handled by the provider)

Conclusions

Learned the basis of big data (a summary of the Master’s degree course)

- Definitions and motivations
- Main components – focus on storage and processing
- Hardware and software architectures

Up next: an overview of some of the main challenges

- Polyglot persistence
- The metadata challenge