The Big Data Ecosystem

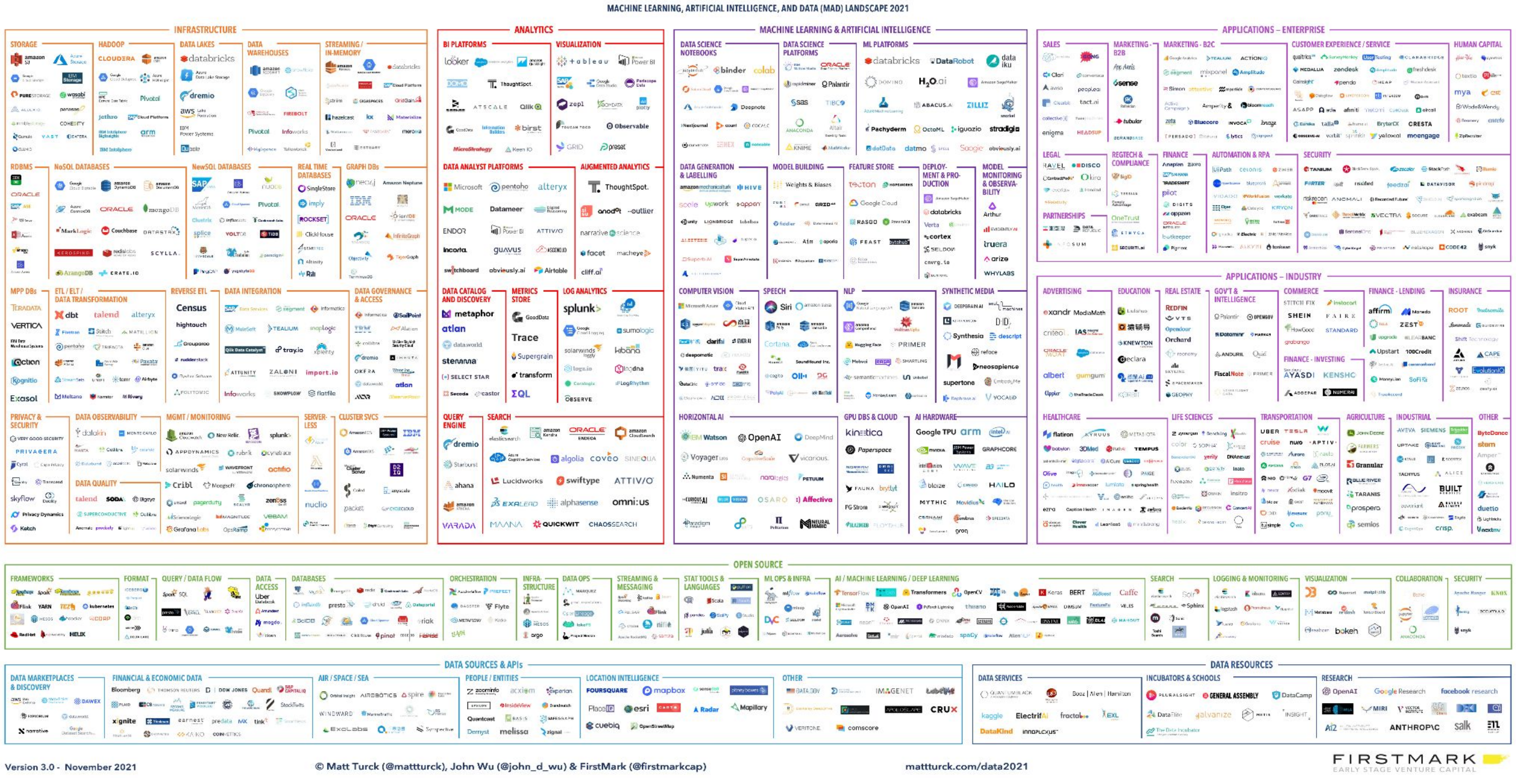
Julián Arenas Guerrero
julian.arenas.guerrero@upm.es

With some slides from Jesús Montes

Nov. 2024

Do you know any of these?

And all these?



MAD landscape: https://mattturck.com/mad2024/

Technology for Big Data

Big Data technologies can be grouped in three categories:

1. **Infrastructure**
    - Collecting/storing the data
    - Processing the data
2. **Analytics + ML/AI**
    - Extracting knowledge from data
    - Visualizing the data/knowledge
3. Applications

A multidisciplinary profile is required (computer systems, statistics, AI, visualization…)

Big Data infrastructure

- Collecting/storing the data
    - Information needs to be properly collected and stored
    - New technologies have appeared that are better suited for the nature of Big Data problems
        - Fast generating data
        - Very large volume
        - Heterogeneous sources and complex data schemas
    - **NoSQL**

Big Data infrastructure

- Processing the data
    - Data needs to be efficiently processed.
    - We need to take as much advantage as possible from distributed and parallel techniques.
    - At the same time, it is important to maintain focus on data and its analysis.
    - **MapReduce**

Basic concepts: scale up & scale out

- Scale up (vertically)

- Scale out (horizontally) - distributed

Basic concepts: OLAP & OLTP

- OLTP (OnLine Transaction Processing)
  - ‘Operational’: focuses on current, real time data supporting regular operations
  - Handles large amount of small transactions
  - Simple queries
  - Optimized for small read/write/updates

- OLAP (OnLine Analytical Processing)
  - ‘Analytical’: focuses on historical data supporting complex analysis and reporting
  - Handles small amount of operations involving large amounts of data
  - Complex queries
  - Optimized for read/query and bulk operations

Are not traditional RDBMS enough?

- RDBMS are successfully used in most professional applications that require proper data handling.
- They provide high performance and the very convenient ACID properties:
    - **A**tomicity
    - **C**onsistency
    - **I**solation
    - **D**urability

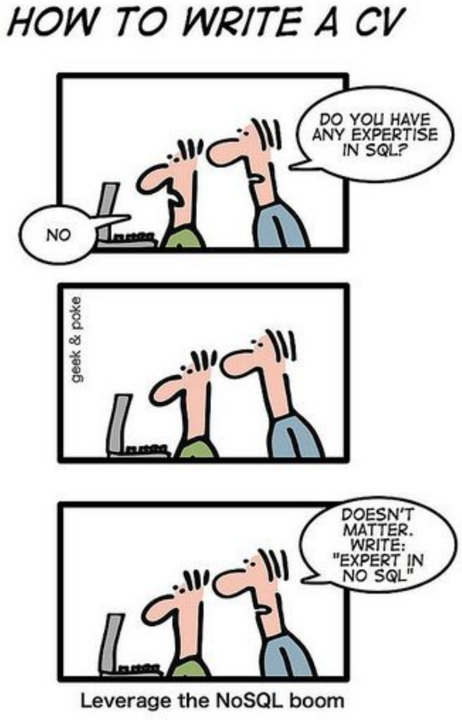
Are not traditional RDBMS enough?

- It is generally accepted that traditional RDBMS are not enough for several reasons. Mainly:
  - Scalability: They do not handle extremely large datasets well.
  - Flexibility: They do not adapt easily to the complexity and requirements of some modern applications.
- Still, there are some voices that argue that most of these claims have not been properly justified, and that traditional RDBMS are suitable for many Big Data applications (example: Oracle Exadata).
- It is up to us to decide what solution is better for each problem.

NoSQL databases

NoSQL (Not Only SQL): Is an extended group of database technologies that do not necessarily use SQL as query language.

- The do not fully guarantee ACID properties.
- They are optimized for LOAD and STORE/INSERT operations, but not UPDATE.
- The have limited JOIN capabilities.
- They scale extremely well.
- They are usually distributed solutions.



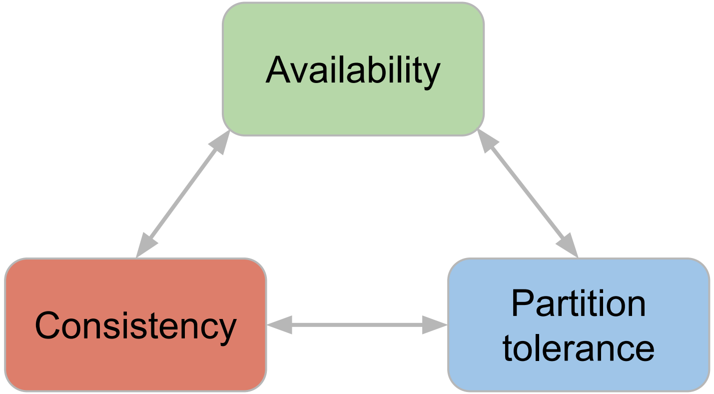
NoSQL databases

- Schemaless (*schema-on-read*)
- Easily replicable
- Simple & custom APIs (No SQL)
- Relaxed consistency requirements (*eventual consistency* vs. *strong consistency* in RDBMS)
- Usually open-source

NoSQL databases

Brewer's Theorem (a.k.a. **CAP** principle):
It is impossible for a **distributed storage system** to present the three following characteristics **simultaneously**:

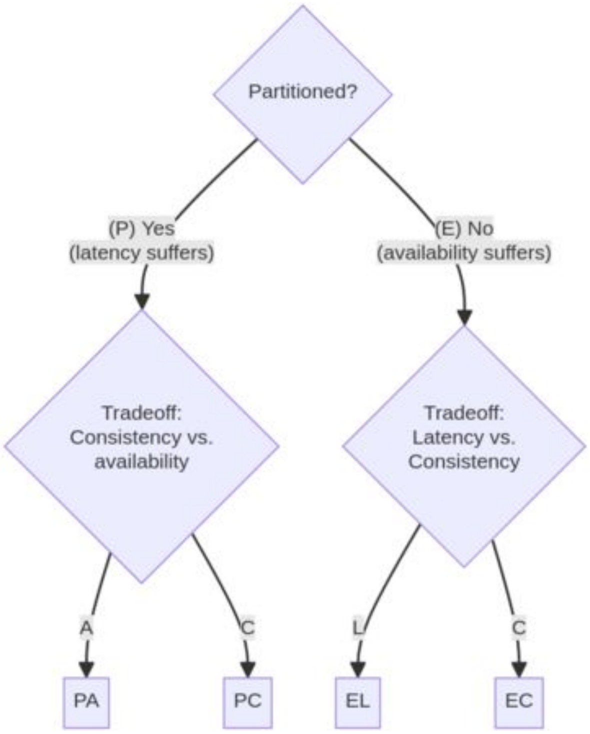
- **C**onsistency
- **A**vailability
- **P**artition tolerance



NoSQL databases

PACELC Theorem:

- In case of network **P**artitioning in a distributed computer system, one has to choose between **A**vailability and **C**onsistency (CAP theorem)
- But **E**lse, in the absence of partitions, one has to choose between **L**atency and loss of **C**onsistency



NoSQL databases

Instead of ACID, NoSQL databases present the BASE properties:

- Basically Available
- Soft state
- Eventual consistency

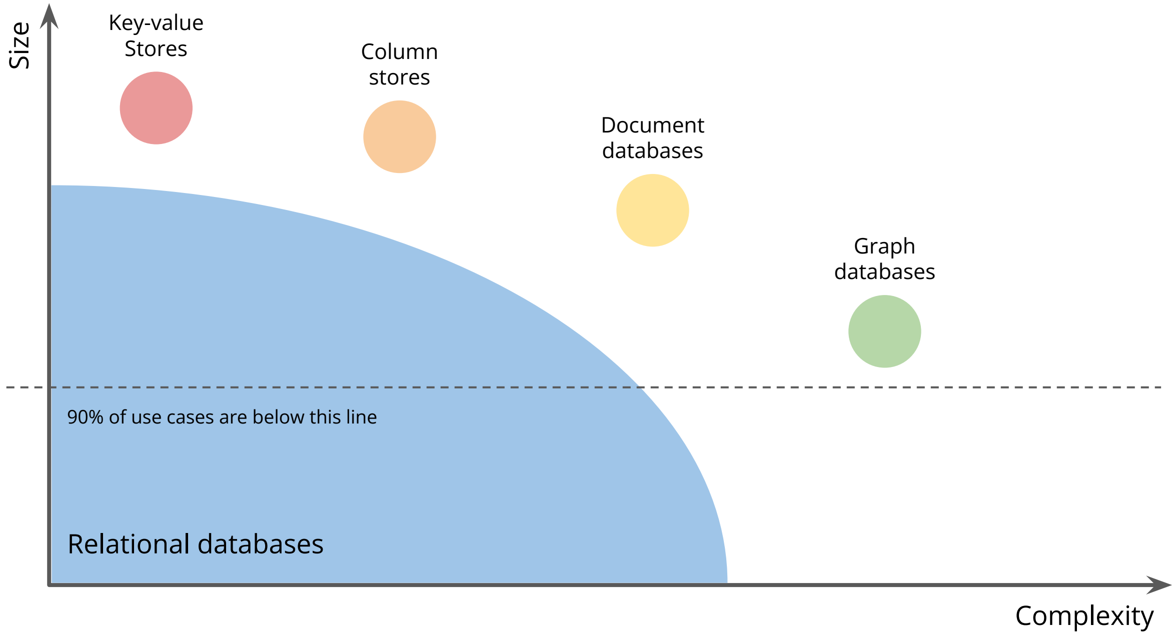
NoSQL databases

NoSQL databases are modern alternatives to traditional RDBMs. They are usually grouped in the following categories:

- Key-value stores
- Wide-column stores/databases
- Document-oriented databases
- Graph databases

CRUD operations (Creation, Retrieval, Update, Deletion)

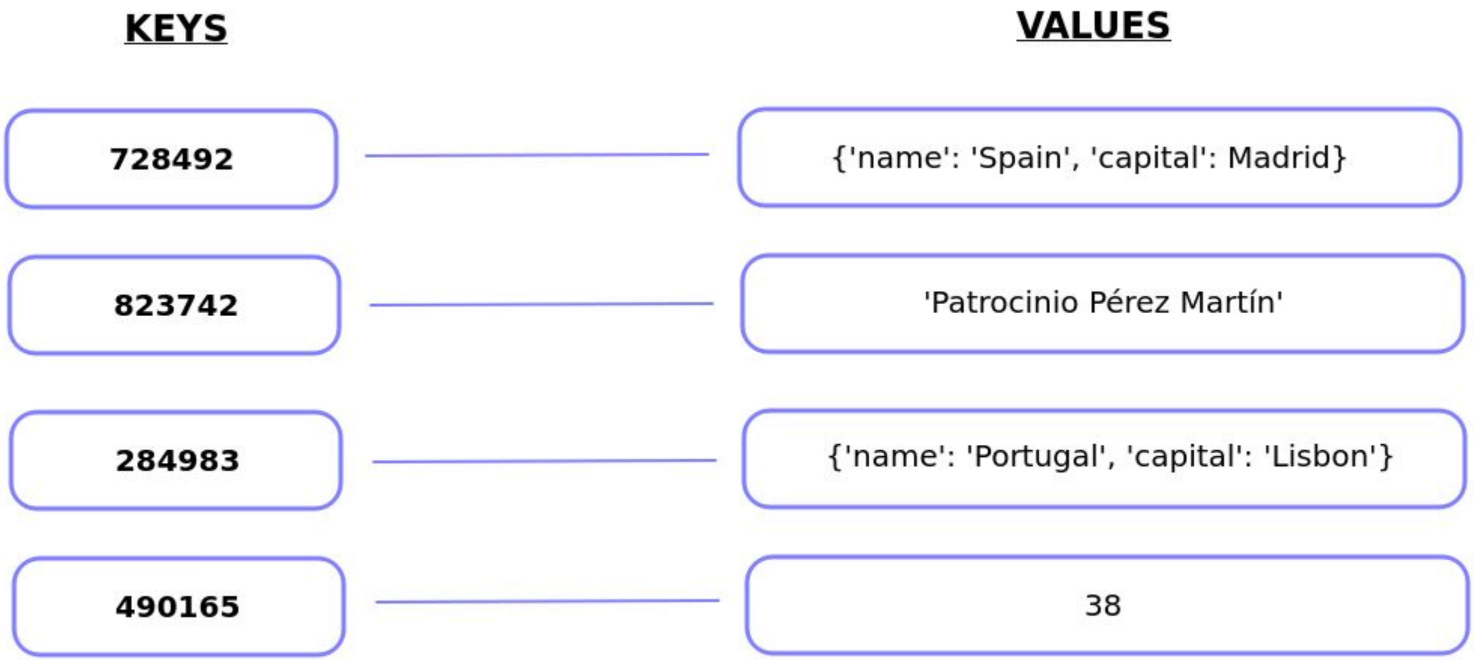
NoSQL databases



Key-value stores

- Very simple data model: just *(key : value)* pairs.
- Designed to store extremely large amounts of data.
- Easy to implement and use.
- Very efficient for locating+reading data.
- Inefficient when we need to access/update only a part of a value:
  - We have to read the entire value.
  - We have to update the entire value.
- Usually implemented in a fully distributed fashion, without a master node that could become a single point of failure (SPF).

Key-value stores



Key-value stores

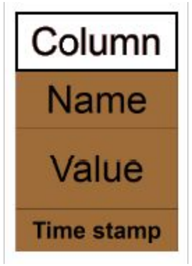
Amazon DynamoDB

redis

riak

RocksDB

Wide-column stores/databases

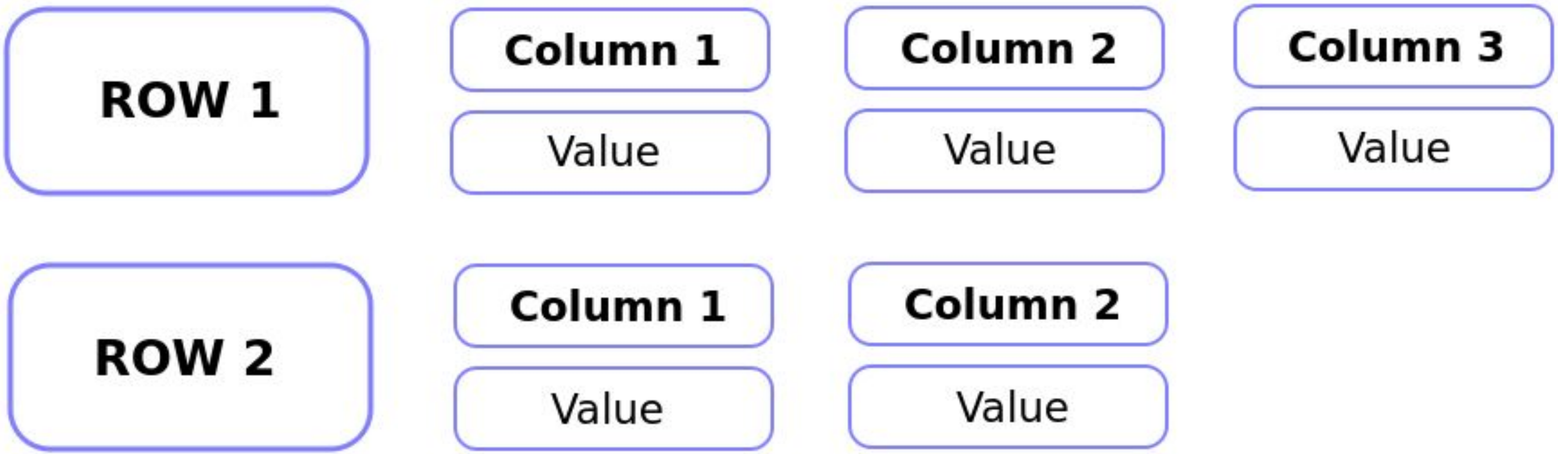


- Data is organized into *columns*, instead of tables with rows.
- **Column**: An object with three fields:
    - Unique name.
    - Value.
    - Timestamp.
- Columns in column-based stores are not the same as matrix columns or attributes in a RDBMs table.
- Columns are grouped into *tuples*, each with a unique key (tuple ID).
- Tuples can be grouped into *column families*, analogous to tables in RDBMSs

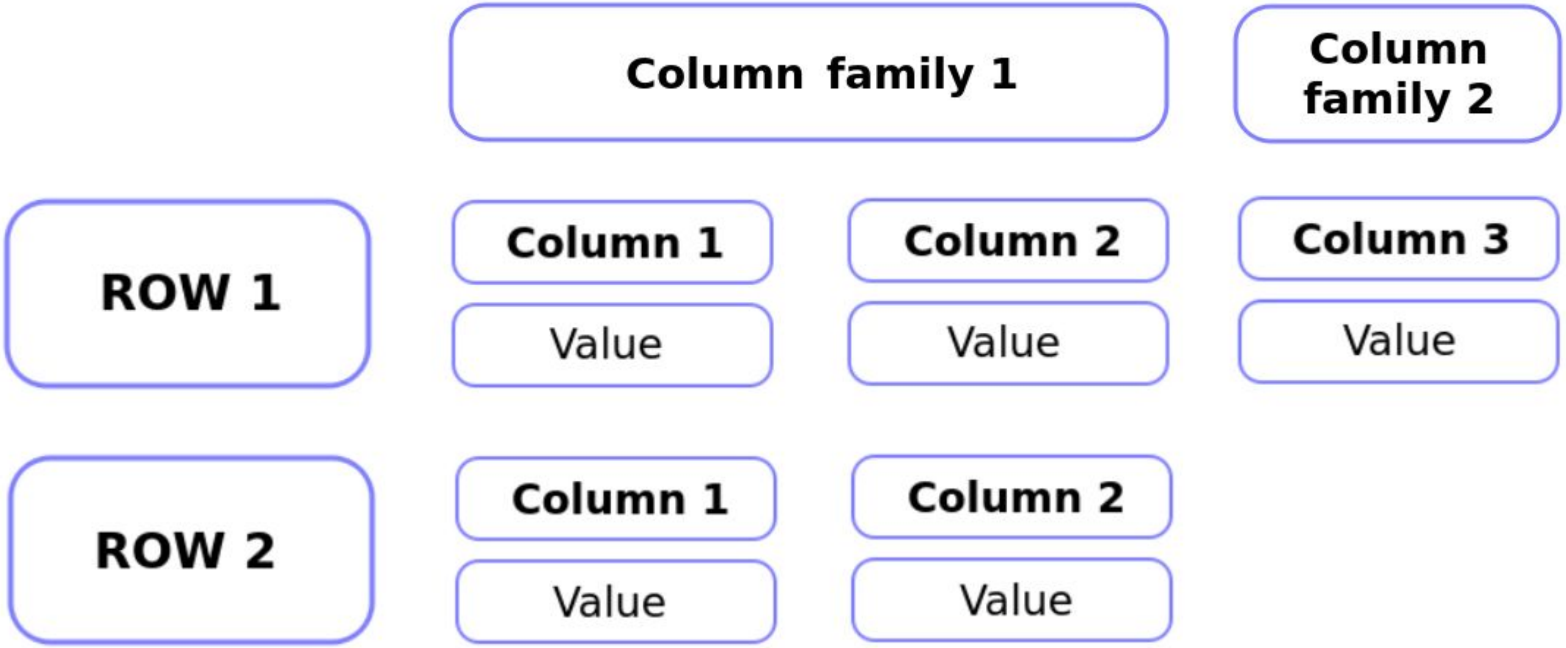
Wide-column stores/databases

- Pros:
  - Aggregation operations (COUNT, SUM, MIN, ...) are very efficient.
  - Batch insertions are very efficient.
  - Optimize the use of storage space.
  - Easy to compress and distribute.
- Cons
  - Read/write an entire tuple takes time (operating over many columns).

Wide-column stores/databases



Wide-column stores/databases



Wide-column stores/databases

Document databases

- Basic principle: Storing documents in a natural way.
- Document:
  - Its specific definition will depend on the database implementation, but they will always present a flexible, rich structure
  - They are typically stored as XML, YAML or JSON/BSON files
  - They can be labeled and organized into collections and/or hierarchies.
- Documents are stored as *(key : value)* pairs. Each document has a unique identifier (the *key*). Its contents are stored in the *value* field.
- The database understands the document format (JSON/BSON, XML, ...) and provides tools to access parts of the documents.

Document databases

- Document example: **JSON**
  - Key-value pairs & arrays

- Organization:
  - Documents are grouped in **Collections** (analogous to tables in relational databases)


```json
{
  "first_name": "John",
  "last_name": "Smith",
  "is_alive": true,
  "age": 27,
  "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}
```

Document databases

> db.persons.count() # count documents in a collection

> db.persons.findOne() # find the first document in a Collection

> db.persons.findOne({_id: ObjectId("a924...104")}) # Find doc. by ID

> db.persons.find().limit(10) # Retrieve a limited number of results

> db.persons.find({"address.city": "New York"}).count() # Find by city

> db.persons.find({year: {$gt: 20}}) # all persons w/ age greater than 20

> db.persons.find().sort({age: 1}) # order by age in ascending order

```
{
  "first_name": "John",
  "last_name": "Smith",
  "is_alive": true,
  "age": 27,
  "address": {
    "street_address": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postal_code": "10021-3100"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}
```

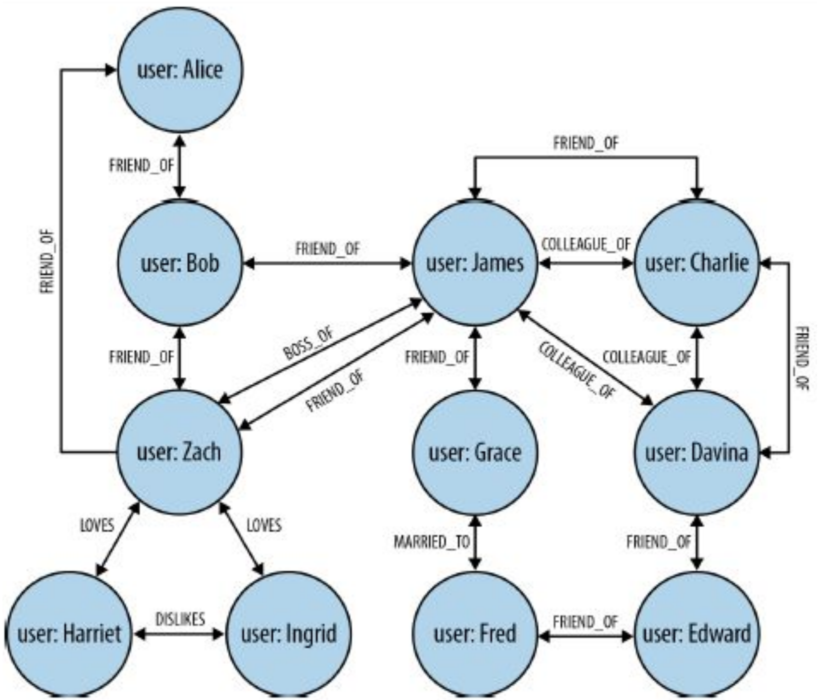
Document databases

mongoDB®

CouchDB relax

Graph databases

- Data is organized into nodes and edges.
- Both nodes and edges have unique IDs, a type ID and a variable set of properties, usually in the form of (key : value) Entries.
- Edges also have a source node ID and a destination node ID.
- Data in graph databases are not restricted to a fixed schema, as in relational DBs.
- The graph model allows for an extremely rich data representation, and to perform data queries that would not be feasible in a relational DB (using graph based algorithms).



Graph databases

Building a graph on top of a relational database

```
CREATE TABLE vertices (
    vertex_id integer PRIMARY KEY,
    label text,
    properties json
);

CREATE TABLE edges (
    edge_id integer PRIMARY KEY,
    tail_vertex integer REFERENCES vertices (vertex_id),
    head_vertex integer REFERENCES vertices (vertex_id),
    label text,
    properties json
);
```

Graph databases

Graph databases: practical exercise

- Creating and querying a graph database
- Use Kùzu in Google Colab
- General Kùzu demo: https://colab.research.google.com/drive/15OLPggnRSBmR_K9yzq6iAGE5MDzNwqoN
- Cypher in Kùzu: https://colab.research.google.com/drive/1NcR-xL4Rb7nprgbvk6N2dIP30oqyUucm
- Kùzu docs: https://kuzudb.com/docusaurus/getting-started/

Big Data processing

- We need an efficient way of processing data.
- When data is too large and/or complex, parallel and distributed approaches are good.
    - Increased performance and throughput.
    - Better use of computational resources (avoiding bottlenecks).
- Parallel programming, however, is often complex and developed *ad hoc* for each problem
- Is there an alternative, more convenient approach?

A little bit of history...

- 2003: Google publishes its papers about Google File System (GFS).
- 2004: Jeffrey Dean and Sanjay Ghemawat (Google) publish their paper about **MapReduce**.
- 2006: Doug Cutting and Mike Cafarella (Yahoo) develop **Hadoop**, based on Google's MapReduce.
  - Open source framework for data processing using the MapReduce model
  - Includes the Hadoop File System (HDFS)
  - Donated to the Apache Foundation and distributed under Apache License 2.0
- 2010: Matei Zaharia develops **Spark** (initially his Ph.D. thesis).
- 2014: Spark becomes a Top-Level Apache Project, with more than 1000 contributors worldwide.

MapReduce

- Distributed data processing framework for large problems that can be parallelized.
- Designed for **large data sets**.
  - A small problem will be much slower with MapReduce.
- Makes use of large clusters (many nodes).
- Uses low-level techniques to improve performance, mainly **data locality**.
- Inspired by functional programming *map/reduce*, but with different objectives.

MapReduce

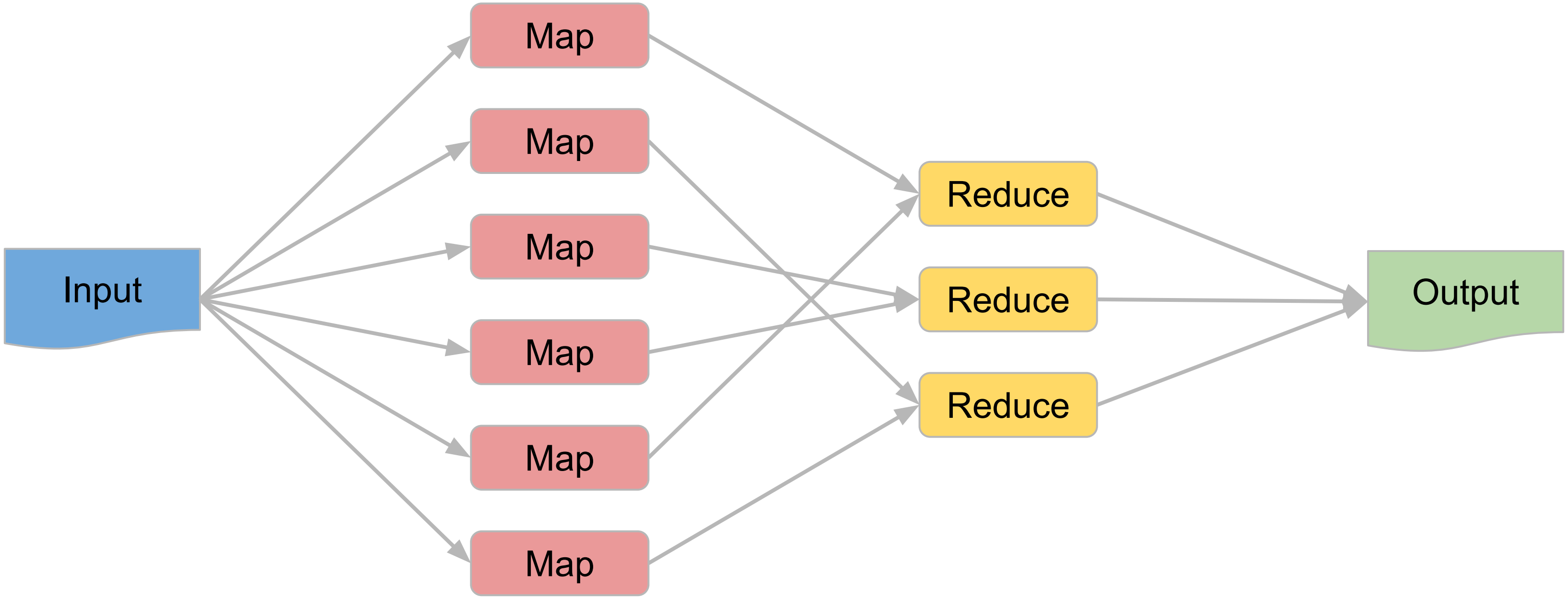
MapReduce is a distributed computing framework.

- Inspired by *map/reduce*
- Designed to make possible the processing of very large datasets

Two phases:

- **Map** phase: Input data are splitted in blocks (sub-problems) and distributed throughout the cluster (worker nodes). Each node processes its sub-problem.
- **Reduce** phase: Sub-problem results are combined into a final output.

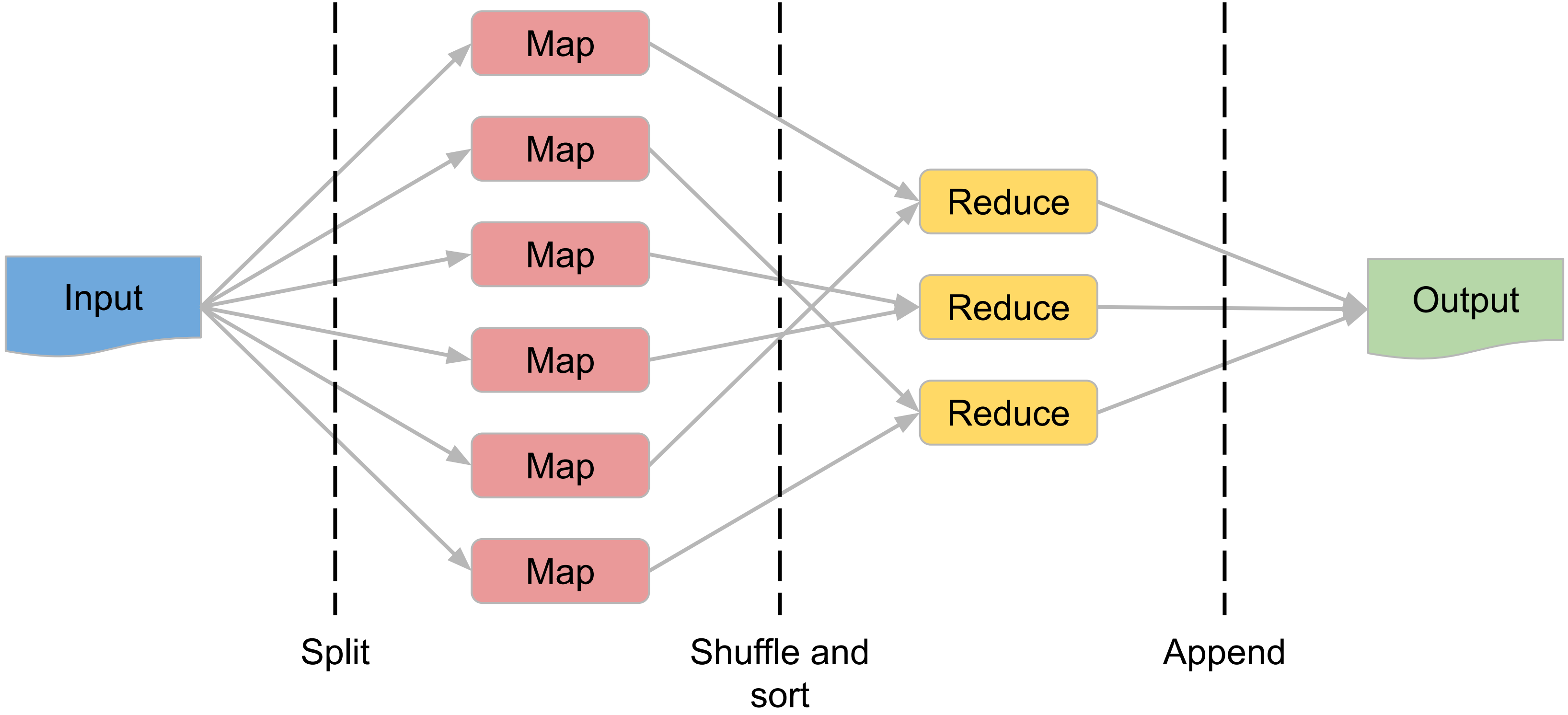
MapReduce



The 5 steps of MapReduce

1. Data is randomly splitted and disseminated throughout the cluster.
2. Map: Each *mapper* (a worker node or processor assigned) executes the user-provided **Map()** function over its assigned data block. The result of each sub-problem is a set of key-value pairs.
3. Shuffle and sort: Map() output is sorted by key and redistributed in the cluster.
4. Reduce: Each *reducer* (again, a worker node or processor assigned) executes the user-provided **Reduce()** over all data associated to a single key. One reducer is executed per key generated.
5. Final output: The output of all reducers put together.

MapReduce

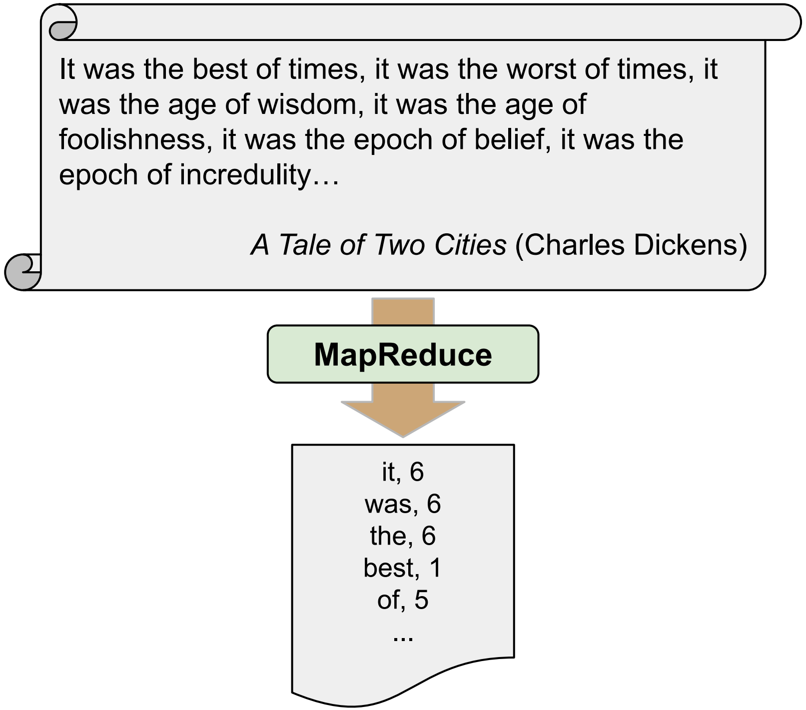

Input
Map
Map
Map
Map
Map
Map
Reduce
Reduce
Reduce
Output
Split
Shuffle and sort
Append

Word Count

One of the most typical MapReduce basic example is the word count problem:

- Input: A (very large) text, usually simply a collection of text lines.
- Desired output: A list of all the word present in the text, and the number of times each appears in the text.

How can we do it with MapReduce?



Word Count: Solution

Map function:

```
Map(key, value) {
     // key: line number
     // value: line contents
     for each word in value {
          emit(word, 1)
     }
}
```

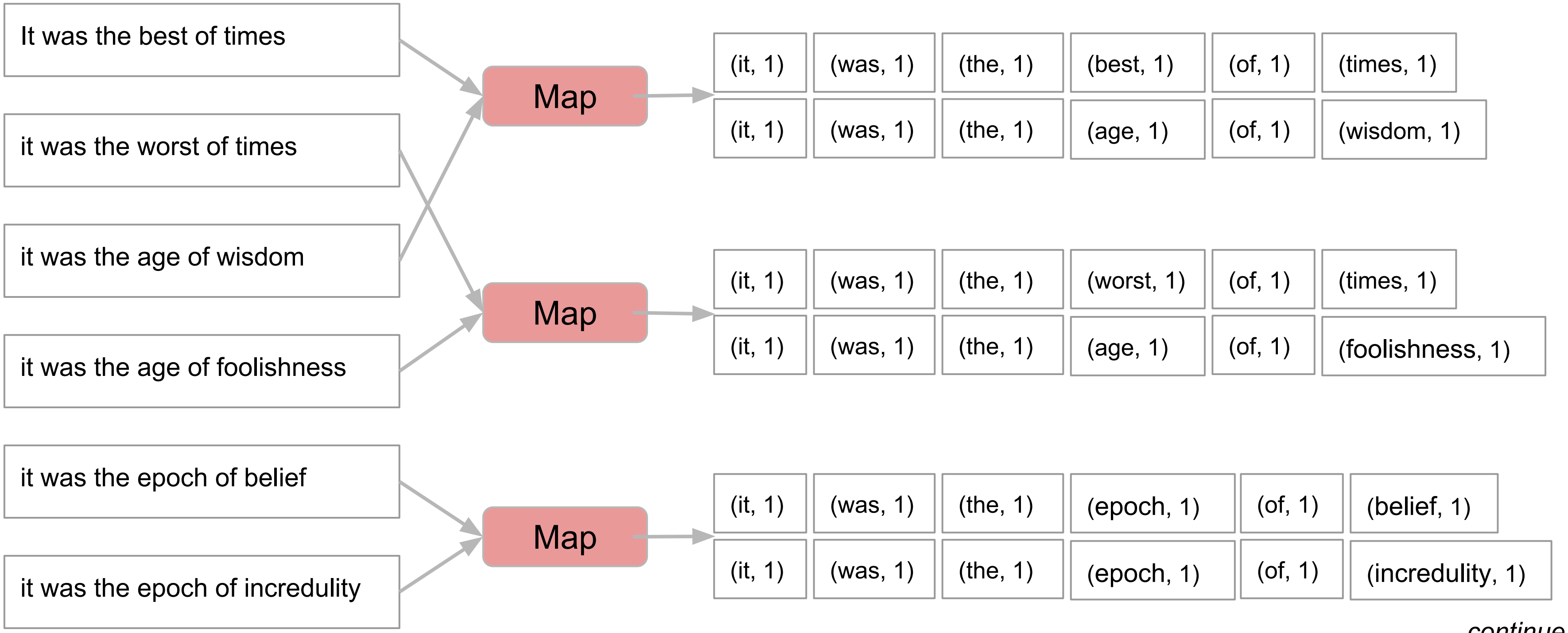
For each word in the line received, the Map function generates a (key, value) pair. The key is the word being processed, and the value is always the number 1.

Reduce function:

```
Reduce(key, values) {
     // key: a word
     // values: a list of counts
     sum = 0
     for each value in values {
          sum += value
     }
     emit(key, sum)
}
```

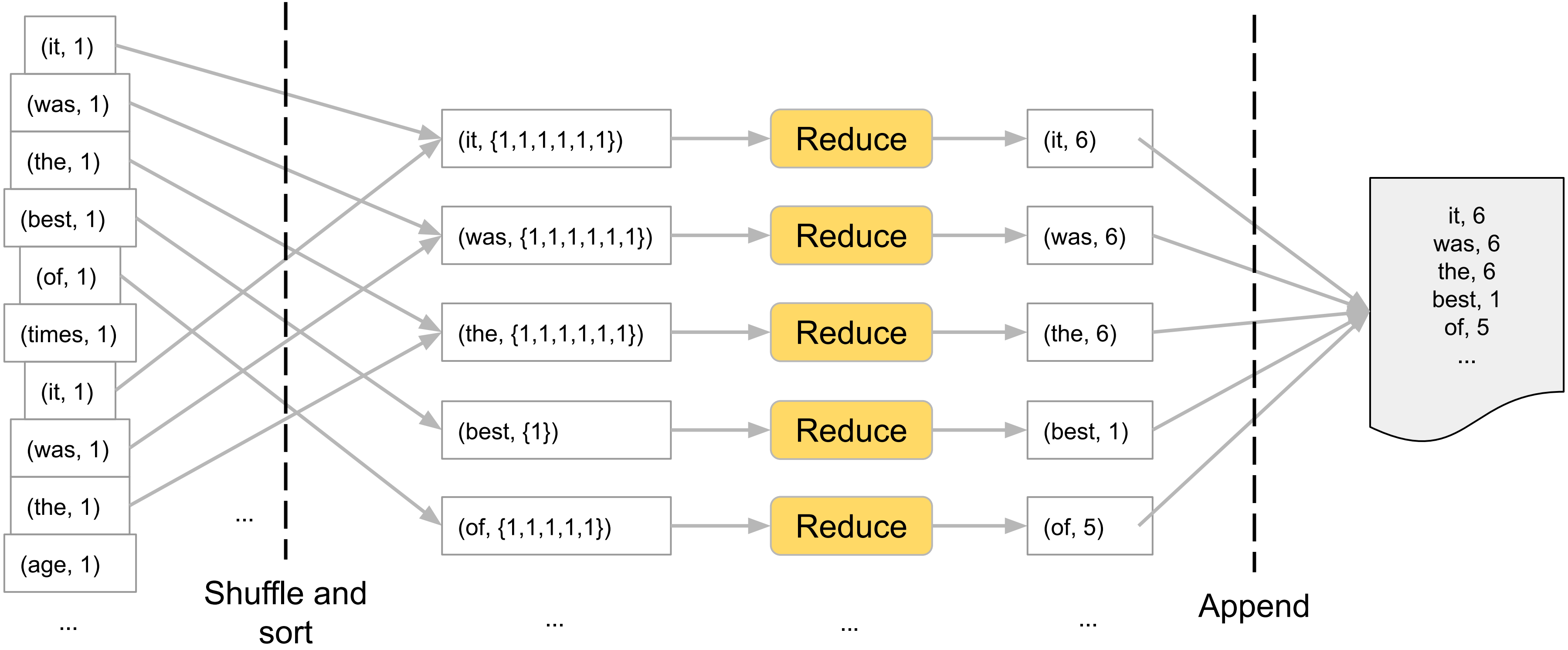
The reduce function receives all values emitted by the mappers for a single key. The function adds these values and produces this sum as a result.

Word Count: How does it work?



continues...

Word Count: How does it work?

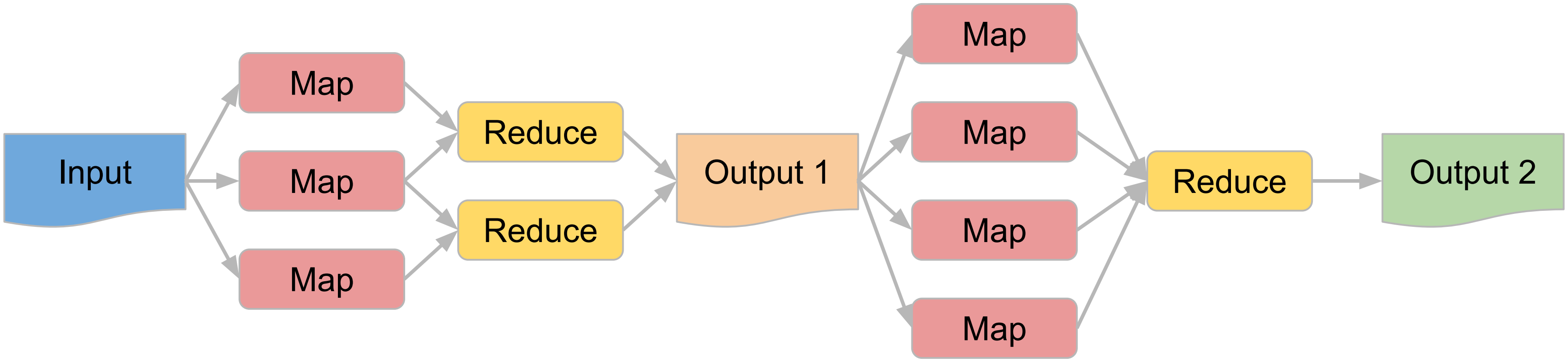


MapReduce

- MapReduce applications require only to provide the implementation of the **Map** and **Reduce** functions.
- MapReduce applications are deployed over a MapReduce framework, usually running in a cluster.
- The framework takes care of all data management operations:
  - Data splitting
  - Shuffle and sort
  - Collection of results
- The parallelization is transparent to the programmer.
- The MapReduce paradigm sacrifices design flexibility in exchange for easy and fast development of parallel applications.

MapReduce

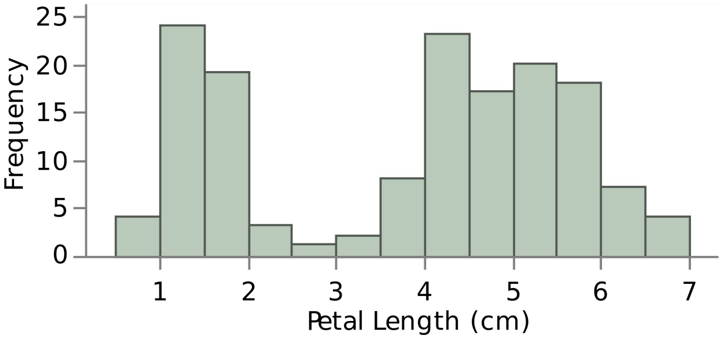
Typical MapReduce applications present more than one MapReduce cycle



In addition to the Map and Reduce functions of each cycle, global/cycle parameters can be defined, but state is never shared between mappers or reducers in the same stage.

The histogram

"A histogram is a graphical representation of the distribution of numerical data. [...] To construct a histogram, the first step is to "bin" the range of values—that is, divide the entire range of values into a series of intervals—and then count how many values fall into each interval." [Wikipedia]



Suppose we want to create a histogram of an extremely large sample of a random variable.

- We know the number of bins we want, but the data file is so big it does not fit in the memory of any single machine we have.
- **Can we do it in a cluster, with MapReduce? If so, how?**

The histogram: Solution

The histogram problem can be solved in two MapReduce cycles/jobs:

- Job 1: Calculate the data range (maximum and minimum).
    - Input: The data file
    - Output: The maximum and minimum of the sample
- Job 2: Knowing the data range and the number of bins, construct the histogram.
    - Input: The data file and the maximum and minimum calculated in job 1 (as global parameters known by all workers).
    - Output: The histogram.

The histogram: Solution

Job 1

```
Map(key, value) {
	// key: line number
	// value: line contents
	numbers = value.split()
	emit(1, (max(numbers), min(numbers)))
}

Reduce(key, values) {
	max_v = -infinity
	min_v = infinity
	for each pair in values {
		max_v = max(max_v, pair[0])
		min_v = min(min_v, pair[1])
	}
	emit("max", max_v)
	emit("min", min_v)
}
```

Job 2

```
Map(key, value) {
	// key: line number
	// value: line contents
	for each number in line {
		bar = floor((number-min)/
		            ((max-min)/n)))
		if (number=max)
			emit(n-1,1)
		else
			emit(bar,1)
	}
}

Reduce(key, values) {
	emit(key, sum(values))
}
```

MapReduce with Document stores

- MapReduce is supported by some NoSQL stores, including MongoDB and CouchDB, as a mechanism for performing read-only queries across many documents.
- Example: you are a marine biologist, and you add an observation record to your database every time you see animals in the ocean. Now you want to generate a report saying how many sharks you have sighted per month.

```
{   “year”:           2022,
    “Month”:           7,
    “family”:          “Sharks”,
    “numAnimals”:     3
    “species”:        {“Cacharias taurus”}
    …   }
```

MapReduce with Document stores

```
db.observations.mapReduce(
    function map() {
        var year = this.observationTimestamp.getFullYear();
        var month = this.observationTimestamp.getMonth() + 1;
        emit(year + "-" + month, this.numAnimals);
    },
    function reduce(key, values) {
        return Array.sum(values);
    },
    {
        query: { family: "Sharks" },
        out: "monthlySharkReport"
    }
);
```

MapReduce: Key Takeaways

- As software, MapReduce is nowadays obsolete
  (it has been replaced by more mature technologies like Spark)
- Its theoretical principles, however, are the basis of most modern data processing frameworks, meeting the developer “halfway” between the infrastructure and the data processing/analysis problem:
  - Development based on **versatile primitive operations** that can be easily developed
  - The framework takes care of data splitting and distribution, load balancing and network management
  - Take advantage of **data locality** to improve performance
  - Procedures can be sequenced/orchestrated to solve more complex tasks