Introduction to NoSQL Databases

ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

The Big Data Context

Applications and Web platforms

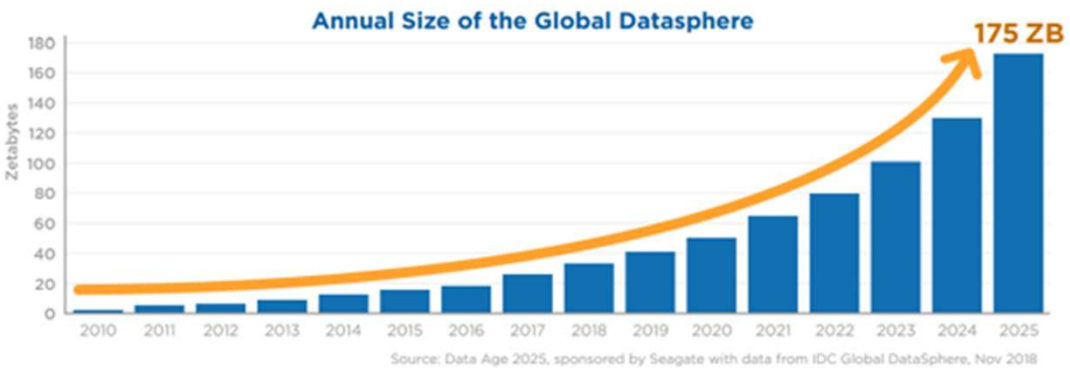
Exponential growth of the amount of Data **x2 / 2 years**

Unprecedent management of this volume

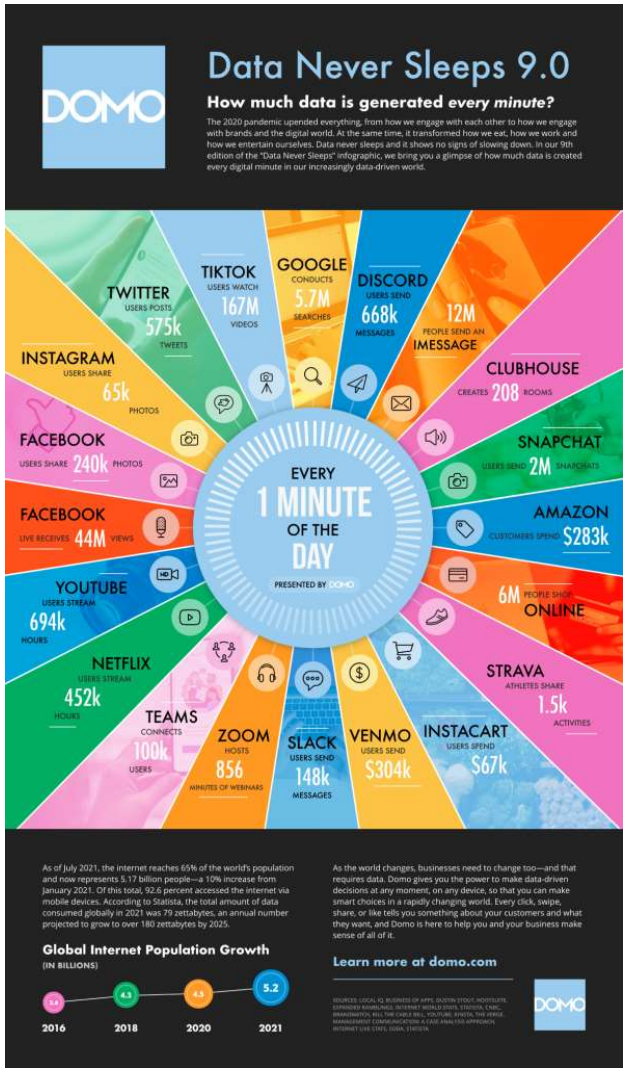
- Need **distribution**: Computation & Data
- **Clusters**: Huge number of servers (Google, Amazon, Facebook, etc.)

  DataCenter : ~5000 servers/data center

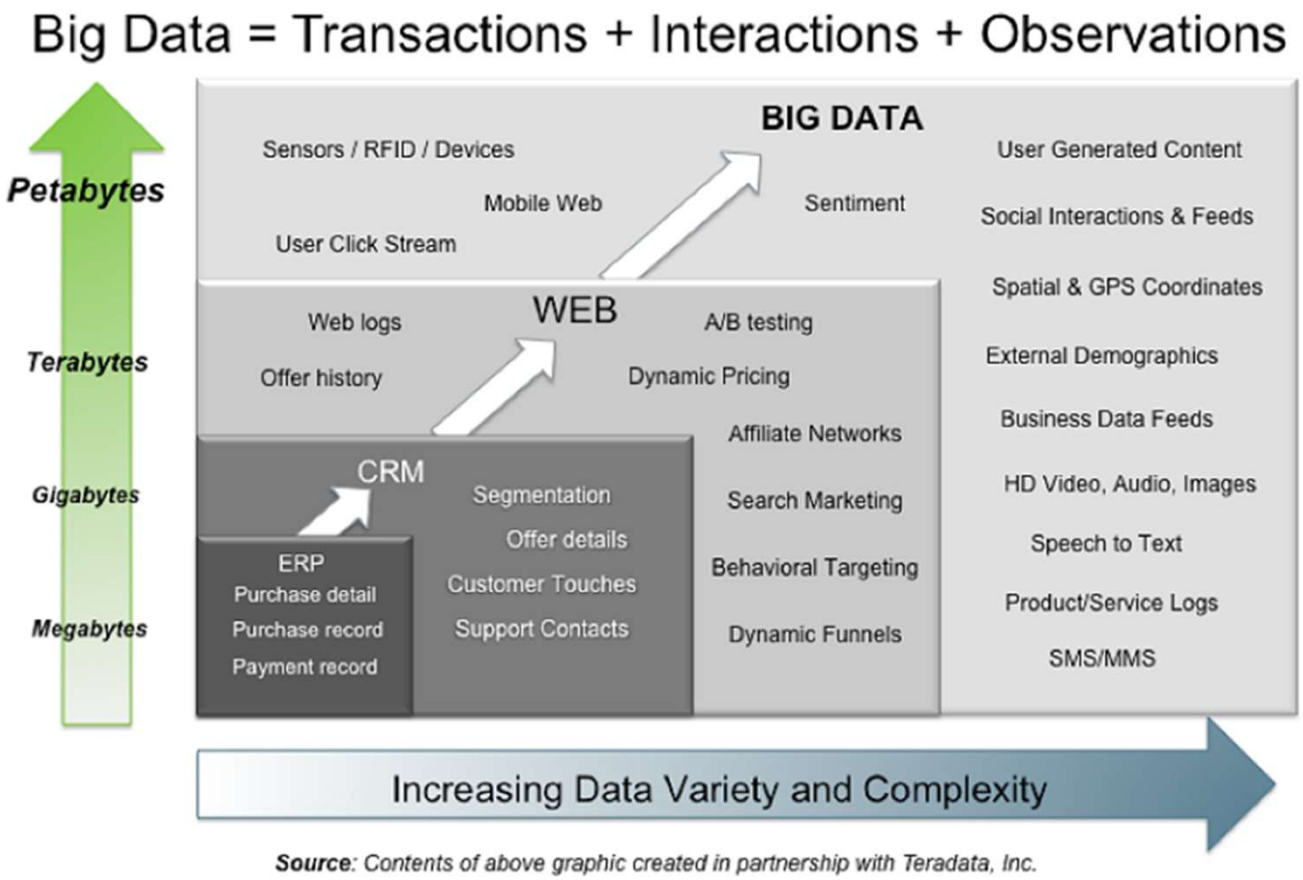
  Google : ~1M de servers



IDC predicts that the Global Datasphere will grow from **33 Zettabytes** in 2018 to **175 Zettabytes** by 2025



RDBMS vs the Problem of 3V



***Source**: Contents of above graphic created in partnership with Teradata, Inc.*

The Problem of 3V

- Volume
  - Designed to store GB/TB
  - But needs PB (maybe EB)
- Variety
  - Heterogeneous data
  - Variable types
  - Text, semi-structured
- Velocity
  - Fast data production

Eventually

- Value
- Veracity

RDBMS vs Distribution

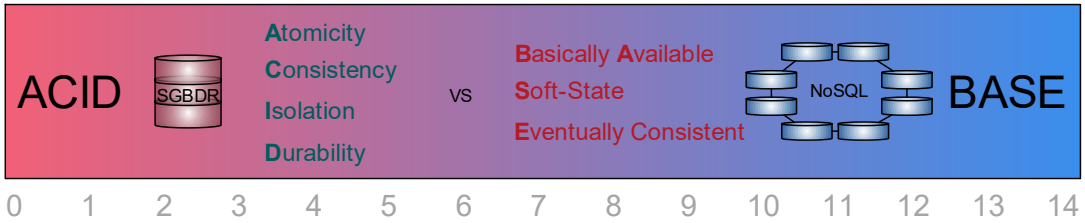
- Pros RDBMS
  - Joins between tables
  - SQL: Rich query language
  - Constraints: integrity, data types, normal forms

- Cons for Distribution
  - How to partition/place data?
  - How to make joins?
  - How to manage fault tolerance?

ACID vs BASE

Local RDBMS: **ACID** transactions

- ***A**tomicity*: integral completion or none
- ***C**onsistency*: consistent at start and end
- ***I**solation*: no communication between them
- ***D**urability*: an operation cannot be reversed



Distributed systems: BASE

- ***B**asically **A**vailable* :

  Any query => An answer
  Even in a changing state

- ***S**oft-state* :

  Opposite to Durability.
  System's state (servers or data) could change over time (without any update)

- ***E**ventually consistent* :

  With time, data can be consistent
  Updates have to be propagated

What is NoSQL?

NoSQL : Not Only SQL

- New data storage/management approach
- Scales up the system (through distribution)
- Complex metadata management (schemaless)

Does not substitute RDBMS!!

NoSQL's target:

- Very huge volume of data (PetaBytes)
- Very short response time
- Consistency is not mandatory

NoSQL characteristics

- No more relations
  - No schema (hardly data types)
  - Dynamic schema
  - ➢ **Collections**
- No more tuples (*3rd Normal Form)*
  - Nesting
  - Arrays
  - ➢ JSON documents
- Distribution
  - Parallelism (abstraction of Map/Reduce, not cloud computing)
- Data replication
  - Availability vs Consistency
  - No more transactions
  - ➢ Few writes, many reads

Sharding: Data Distribution Strategies

- Files are segmented in chunks (64MB, 256MB…)
  - Chunks are distributed in the cluster
  - Data are placed according to a **sharding** strategy

- 3 types of technics of sharding:
  1. **HDFS:** Resource allocation based (racks, switches, datacenter)
     ++ Fault tolerance and massive computations
  2. ***Clustered index:*** Tree-based structure (sort)
     ++ Physical data clustering and dynamicity
  3. **DHT:** Hash-based structure
     ++ elasticity and self-management

Sharding with HDFS

Hadoop Distributed File Sytem : Resources allocation strategy

- Distributed file system
- Rely on servers load balancing
- Dedicated to fault tolerance
- Dynamic allocation and optimization

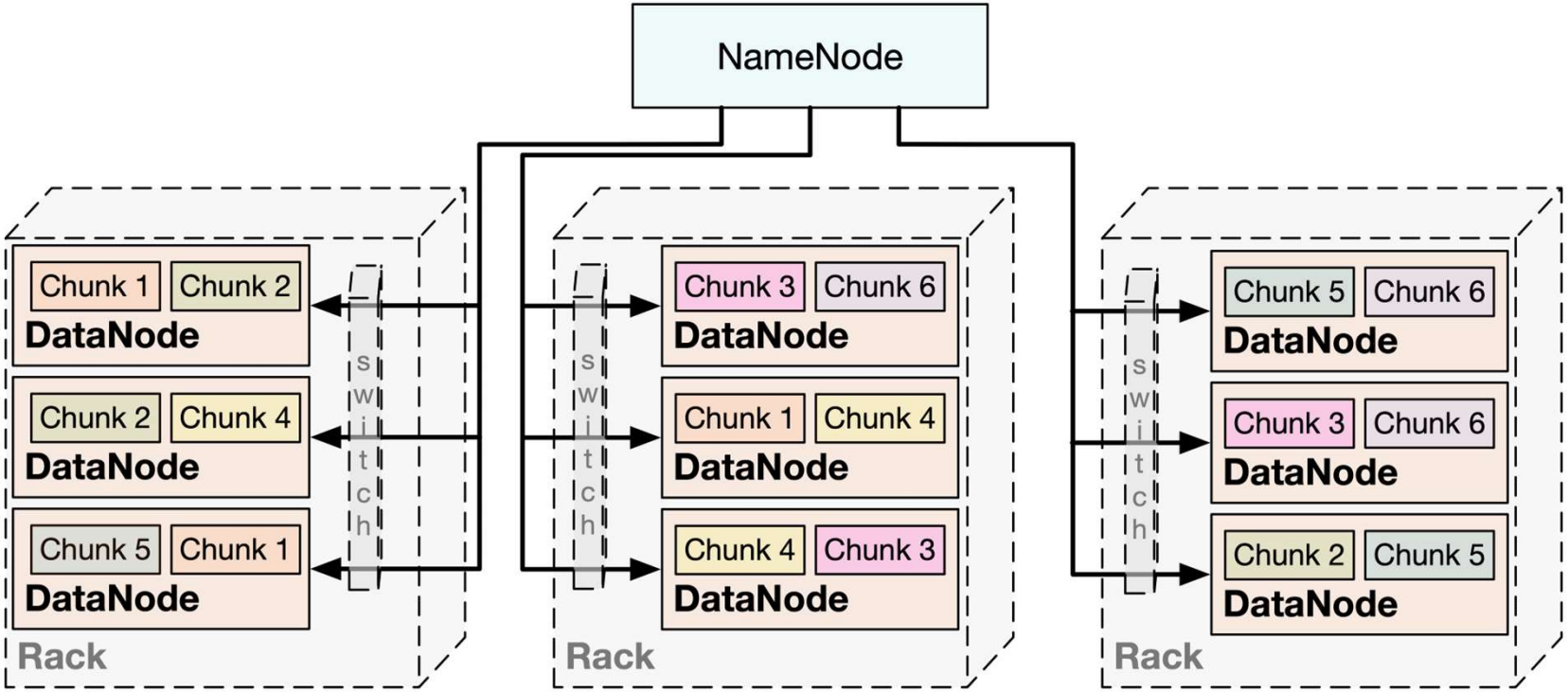
(1) See *Hadoop Distributed File System* http://chewbii.com/transparents-hdfs-hadoop/

Sharding with HDFS

Example

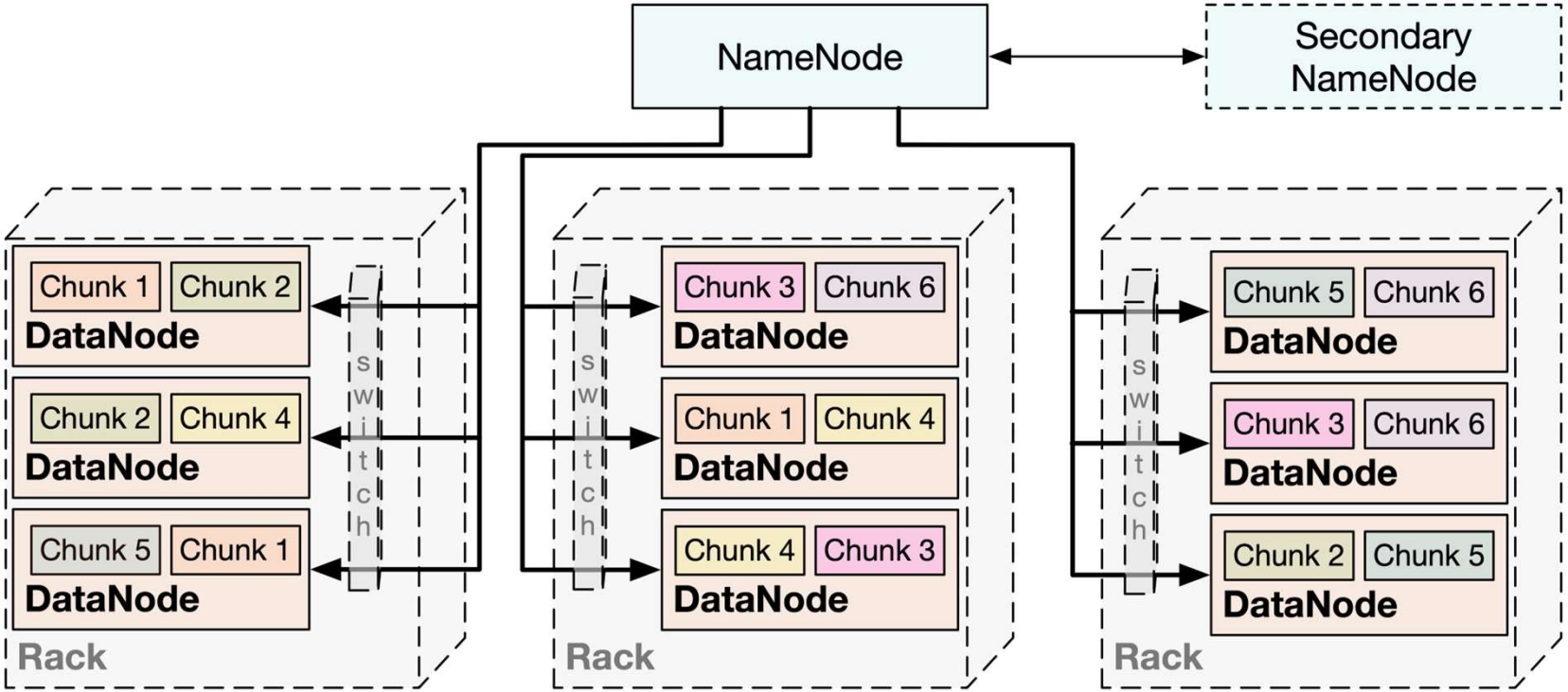
Sharding with HDFS

Example



Sharding with HDFS

Example



Sharding with HDFS

NoSQL solutions

Highly distributed computation

Fault Tolerance

Sharding with Clustered index

Distributed clustered index

- Sorted data according to a key
  - Primary key
  - Build chunks (default 256Mo)
  - Distribute chunks
  - Replicate chunks

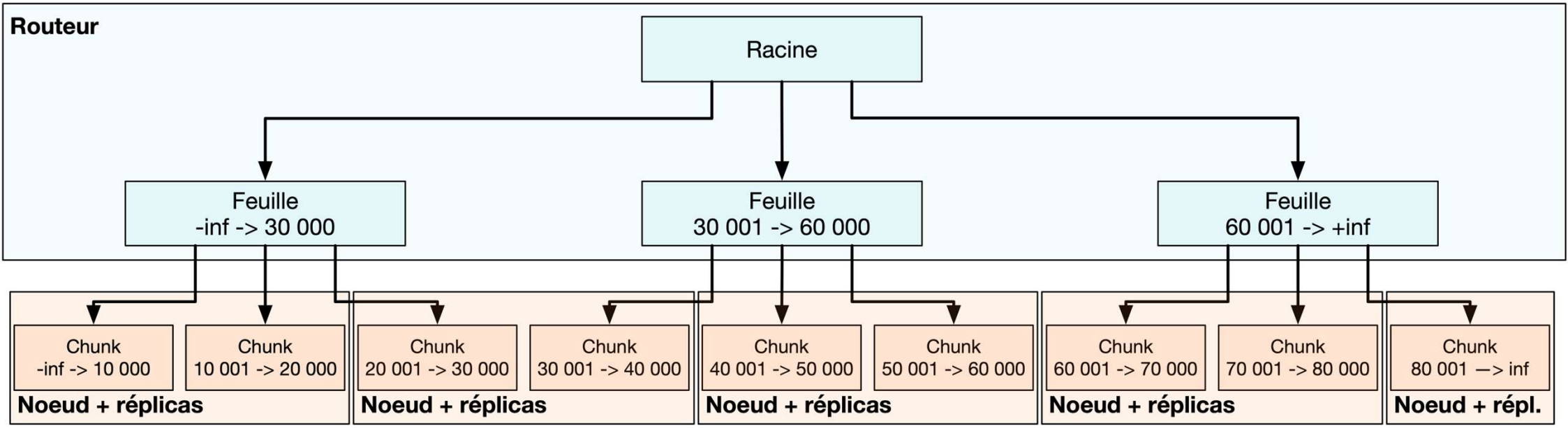
➞ *Range* queries and *group by* queries

⚠ Only one key can be chosen

(2) See « index non-dense » http://chewbii.com/videos-optimisation-bases-de-donnees/ (Vidéo 4)

Sharding with Clustered Index

Example



Sharding with Clustered Index

Solutions

Data organization

Complex queries

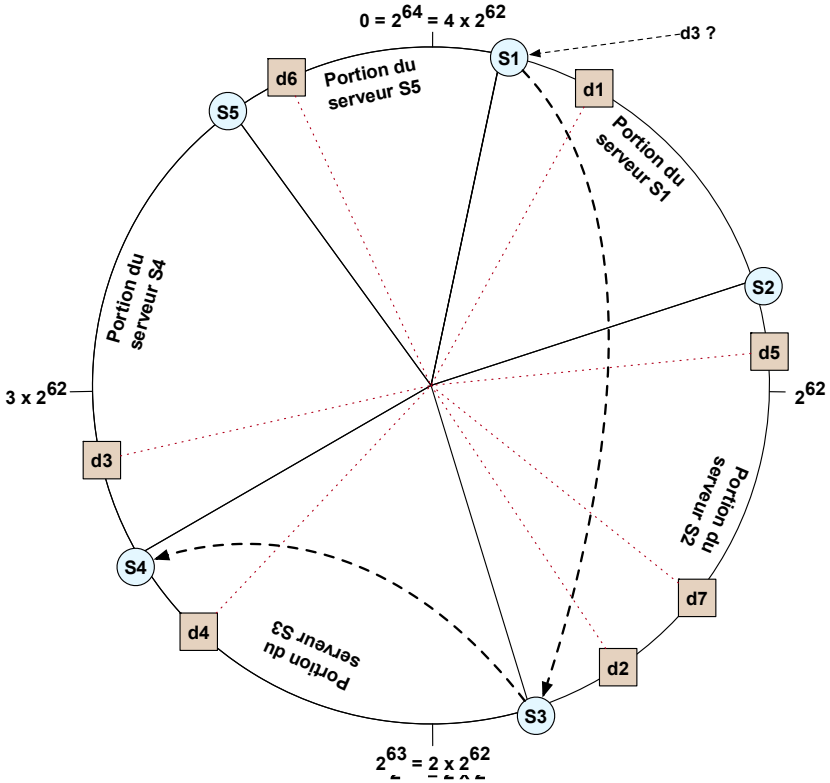
Sharding with DHT

Distributed Hash Table (DHT[3])

- Ring of virtual servers
  - Max 2^{64}
- Unique hash table: splitted & distributed
  - Routing
  - Efficiency
  - Self-Management (no main server)

(3) See DHT http://chewbii.com/hachagedynamique/ (Vidéo 4 & 5)

Sharding with DHT Example



Chunk S1
resp d1
réplicas d5 d2 d4 d7

Chunk S2
resp d5 d2 d7
réplicas d4 d3

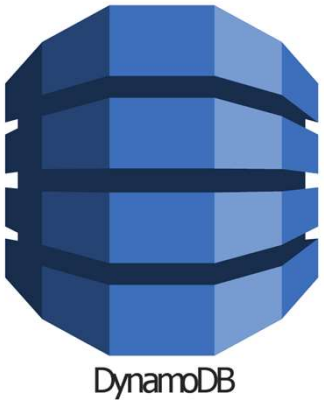
Chunk S3
resp d4
réplicas d3 d6

Chunk S4
resp d3
réplicas d6 d1

Chunk S5
resp d6
réplicas d1 d5 d2 d7

Sharding with DHT

Solutions

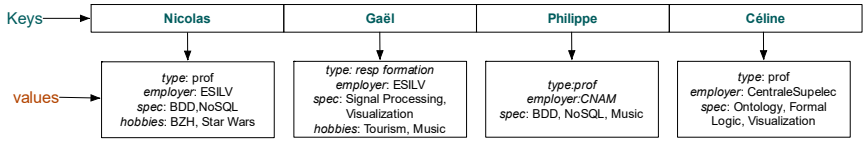


Elasticity

Self-Management

NoSQL Data Type Families

Key-Value oriented



Wide-Column oriented

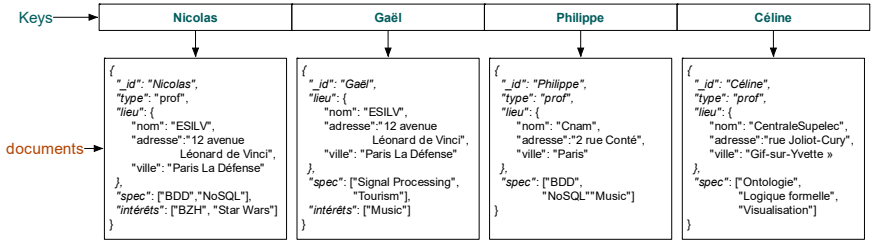
id	status
Nicolas	prof
Gaël	resp formation
Philippe	prof
Céline	prof

id	employer
Nicolas	ESILV
Gaël	ESILV
Philippe	CNAM
Céline	CentraleSupelec

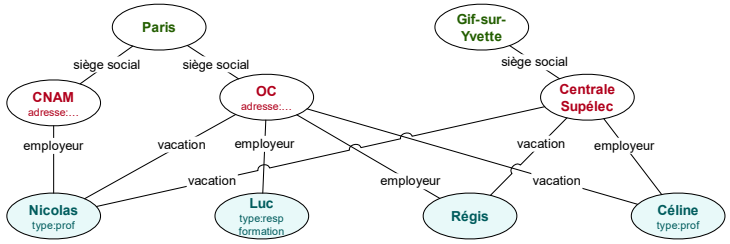
id	spec
Nicolas	BDD
Nicolas	NoSQL
Gaël	Signal Processing
Gaël	Visualization
Philippe	BDD
Philippe	NoSQL
Philippe	Music
Céline	Ontology
Céline	Formal Logic
Céline	Visualization

id	hobbies
Nicolas	BZH
Nicolas	Star Wars
Gaël	Tourism
Gaël	Music

Document oriented



Graph oriented



Key-Value Oriented

Similar to a distributed "*HashMap*"

Key + Value:

No fixed schema on values (strings, object, integer, binaries…)

Drawbacks:

No structures nor typing

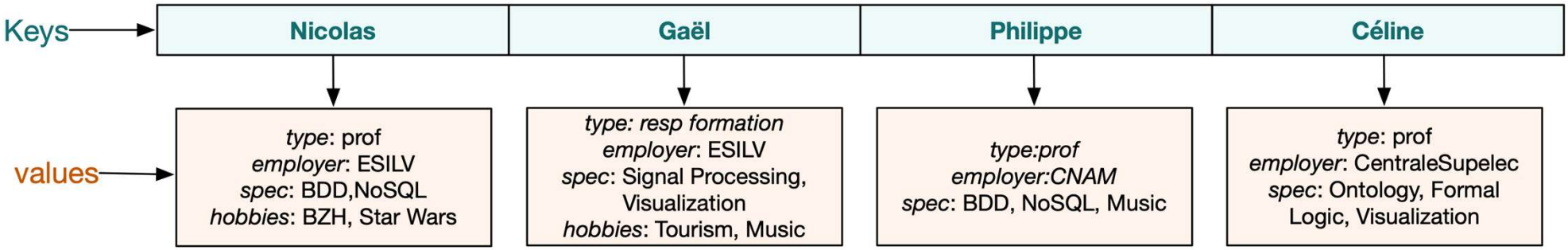
No structured-based queries

Key-Value Oriented

Example

Relational database

id	status	employer	spec	hobbies
Nicolas	prof	ESILV	BDD, NoSQL	BZH, Star Wars
Gaël	resp formation	ESILV	Signal Processing, Visualization	Tourism, Music
Philippe	prof	CNAM	BDD, NoSQL, Music	
Céline	prof	CentraleSupelec	Ontology, Formal Logic, Visualization	



Key-Value Oriented

CRUD

Queries

CREATE (**key**, **value**)

CREATE (**"Nicolas"**, **"type:'prof',employer:'ESILV',spec:'BDD,NoSQL',hobbies:'BZH,Star Wars' "**) → **OK**

READ(**key**)

READ(**"Nicolas")** → **"type:'prof',employer:'ESILV',spec:'BDD,NoSQL',hobbies:'BZH,Star Wars' «** → **OK**

UPDATE(**key**, **value**)

UPDATE(**"Nicolas"**, **"type:'prof',employer:ESILV',spec:'BDD,NoSQL' "**) → **OK**

DELETE(**key**)

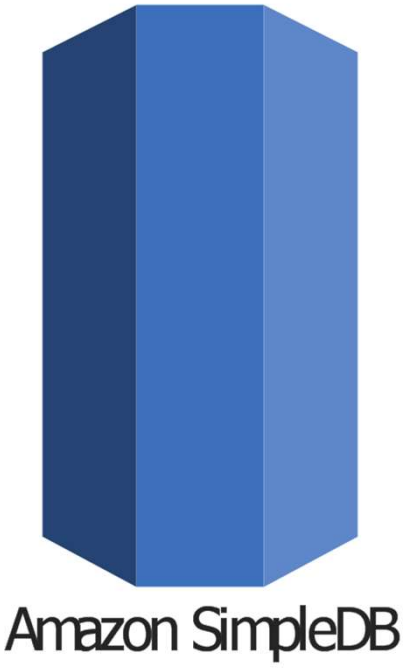
DELETE(**"Nicolas")** → **OK**

Key-Value Oriented

Use cases

- Web site logs store
- Web site/DB cache
- User profil management
- Sensors status
- E-commerce bucket storage

Key-value Oriented
Solutions



Efficiency

Easy to set up

Wide-Column Oriented

Wide Column ≠ Columnar ≠ Column

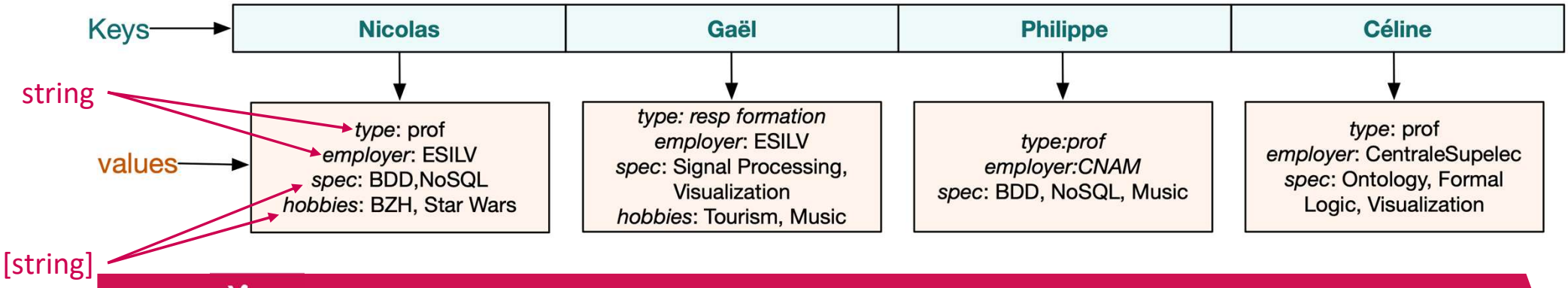
- Wide-Column -> tuples with enhanced typing (arrays, dictionaries, dynamic schema)
  - Remain RDBMS oriented with strong typing (query validation)
  - Also called wide-column oriented stores
- Columnar -> 1 attribute = 1 storage file
  - Column-oriented: attributes are separated in column-families and **stored independently**
  - Efficient queries on columns (**aggregates**)
  - Not NoSQL -> In Memory databases
- Column -> highly structured tuples = relational (SQL)

Wide-Column Oriented

Example

Relational database

id	status	employer	spec	hobbies
Nicolas	prof	ESILV	BDD, NoSQL	BZH, Star Wars
Gaël	resp formation	ESILV	Signal Processing, Visualization	Tourism, Music
Philippe	prof	CNAM	BDD, NoSQL, Music	
Céline	prof	CentraleSupelec	Ontology, Formal Logic, Visualization	



Wide-Column Oriented ≠ Columnar oriented

Example

id	status	employer	spec	hobbies
Nicolas	prof	ESILV	BDD, NoSQL	BZH, Star Wars
Gaël	resp formation	ESILV	Signal Processing, Visualization	Tourism, Music
Philippe	prof	CNAM	BDD, NoSQL, Music	
Céline	prof	CentraleSupelec	Ontology, Formal Logic, Visualization	

id	status
Nicolas	prof
Gaël	resp formation
Philippe	prof
Céline	prof

id	employer
Nicolas	ESILV
Gaël	ESILV
Philippe	CNAM
Céline	CentraleSupelec

id	spec
Nicolas	BDD
Nicolas	NoSQL
Gaël	Signal Processing
Gaël	Visualization
Philippe	BDD
Philippe	NoSQL
Philippe	Music
Céline	Ontology
Céline	Formal Logic
Céline	Visualization

id	hobbies
Nicolas	BZH
Nicolas	Star Wars
Gaël	Tourism
Gaël	Music

Wide-Column Oriented Queries

Column-oriented queries

How many **professors (status)** are at **ESILV (employer)**

id	status
Nicolas	prof
Gaël	resp formation
Philippe	prof
Céline	prof

id	employer
Nicolas	ESILV
Gaël	ESILV
Philippe	CNAM
Céline	CentraleSupelec

id	spec
Nicolas	BDD
Nicolas	NoSQL
Gaël	Signal Processing
Gaël	Visualization
Philippe	BDD
Philippe	NoSQL
Philippe	Music
Céline	Ontology
Céline	Formal Logic
Céline	Visualization

id	hobbies
Nicolas	BZH
Nicolas	Star Wars
Gaël	Tourism
Gaël	Music

Wide-Column Oriented

Use cases

- Statistics on online polls
- Logging
- Category searchs on products (ie. Ebay)
- Reporting

Wide-Column Oriented

Solutions

Simple queries

Strong typing

Document Oriented

Based on the key-value store

- Add semi-structured data (JSON/XML)
- Keys indexing
- Lists of values, nesting
- Allow fusion of concepts

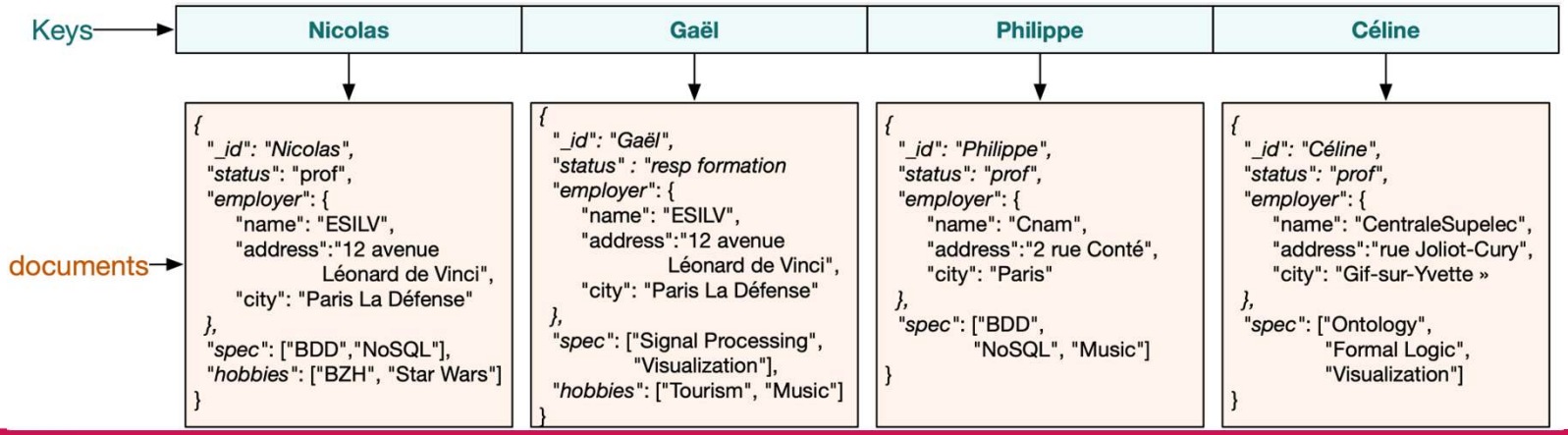
HTTP API

- More complex than CRUD
- Rich queries (database)

Document-Oriented Stores

Example

id	status	employer	spec	hobbies
Nicolas	prof	ESILV	BDD, NoSQL	BZH, Star Wars
Gaël	resp formation	ESILV	Signal Processing, Visualization	Tourism, Music
Philippe	prof	CNAM	BDD, NoSQL, Music	
Céline	prof	CentraleSupelec	Ontology, Formal Logic, Visualization	

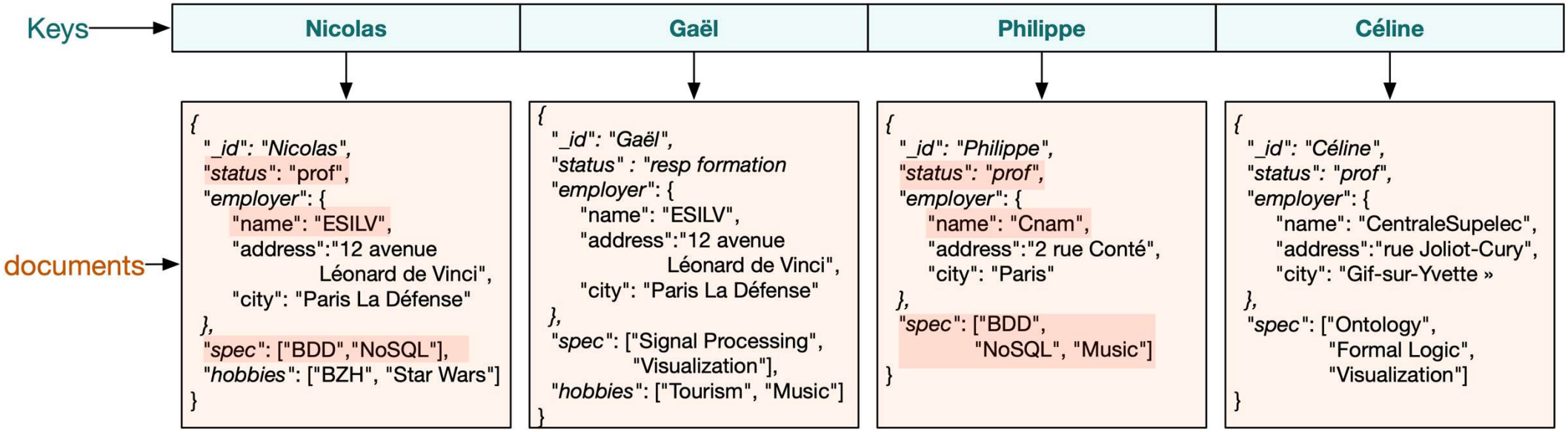


Document Oriented

Queries

Manipulations on documents content

Establishment (employer.name) of **professors (status)** specialized in **BDD (*in* spec)**



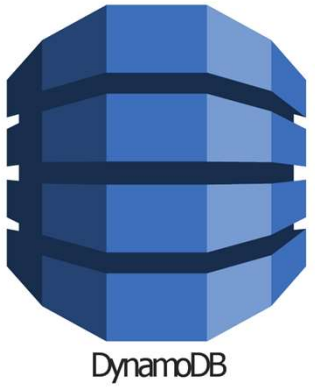
Document Oriented

Use cases

- CMS: Content Management Systems
  - Online libraries
  - Products management
  - Software logging
  - Metadata on multimedia stores
- Collection of complex events
- Emails storage
- Social networks logging

Document Oriented

Solutions



Richness of queries

Manage objects

Graph Oriented

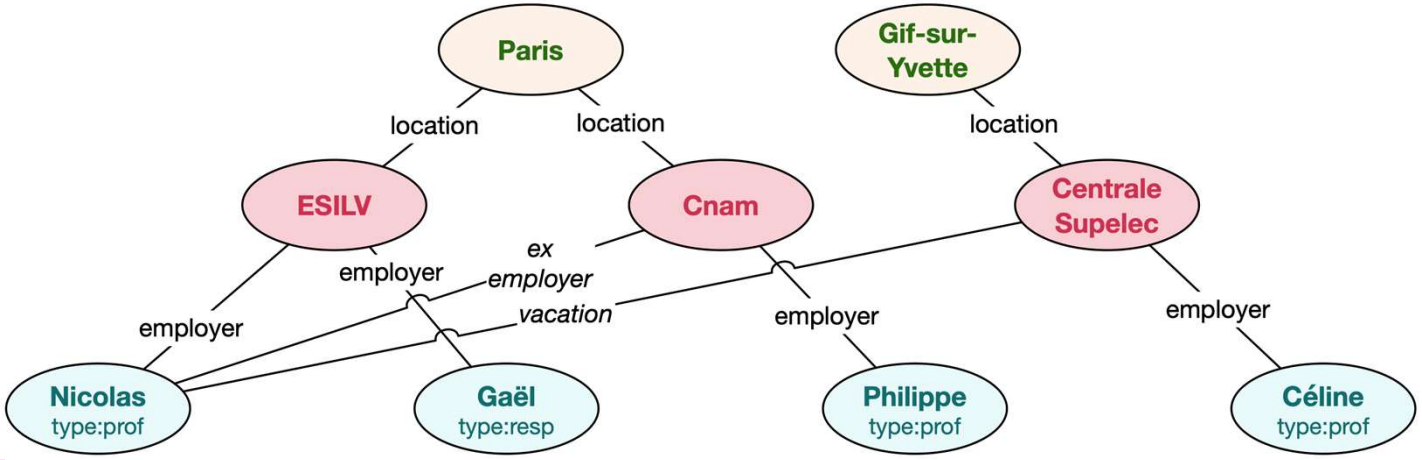
Storage: nodes, relations and properties

- Graph Theory
- **Pattern queries** on the graph
- Data are loaded on demand
- Difficulties for **data modeling**

Graph Oriented

Example

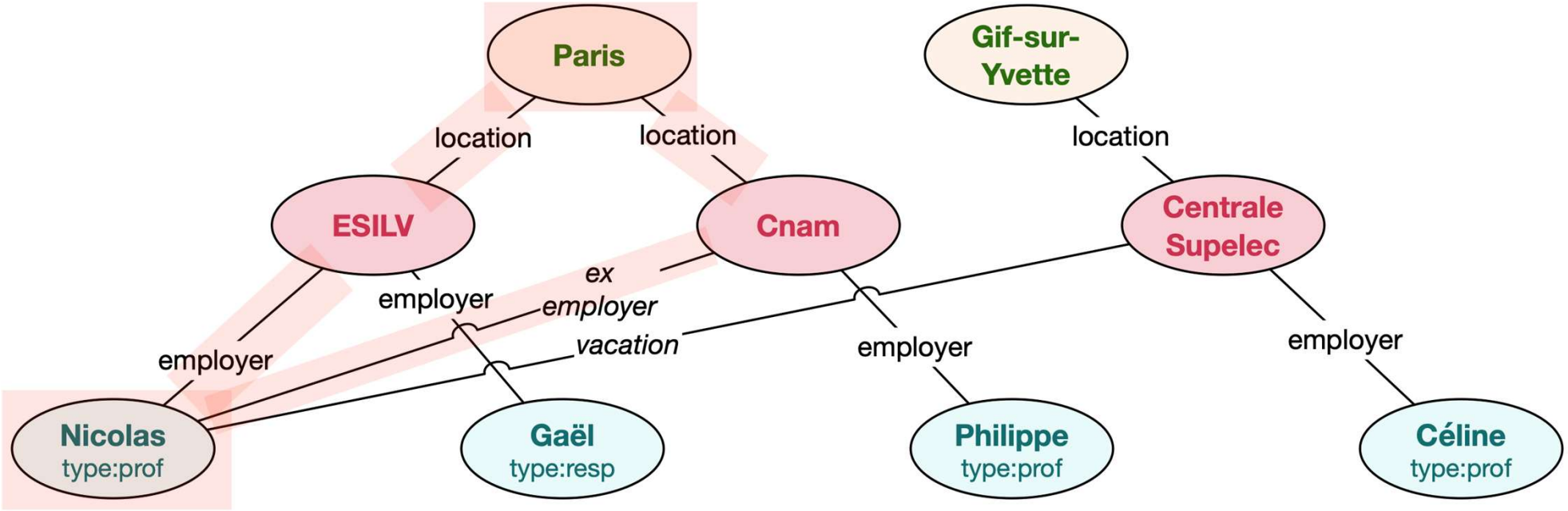
id	status	employer	spec	hobbies
Nicolas	prof	ESILV	BDD, NoSQL	BZH, Star Wars
Gaël	resp formation	ESILV	Signal Processing, Visualization	Tourism, Music
Philippe	prof	CNAM	BDD, NoSQL, Music	
Céline	prof	CentraleSupelec	Ontology, Formal Logic, Visualization	



Graph Oriented
Queries

Pattern queries

Persons who **had 2 employers** at **Paris**



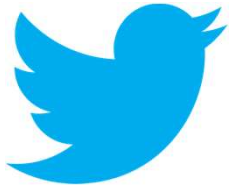
Graph Oriented

Use cases

- Pattern recognition
  - Recommandation
  - Fraud detection
  - SIG (road network, electricty, reachability)
- Graph computations
  - Shorted path
  - PageRank
  - Connexity
  - Communities detection
- Linked data

Graph Oriented
Solutions

Azure Cosmos DB:



FlockDB

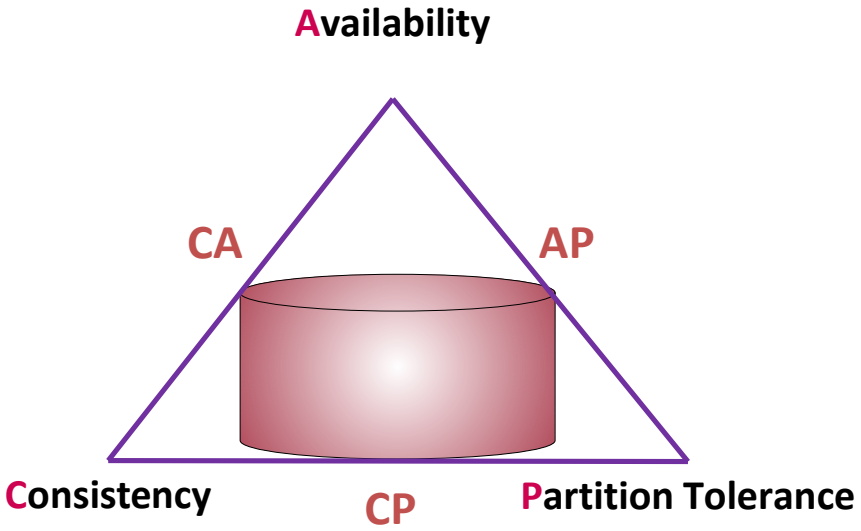
Network

Recommendation

The CAP Theorem
[Brewer 2000]

3 main properties for distributed management

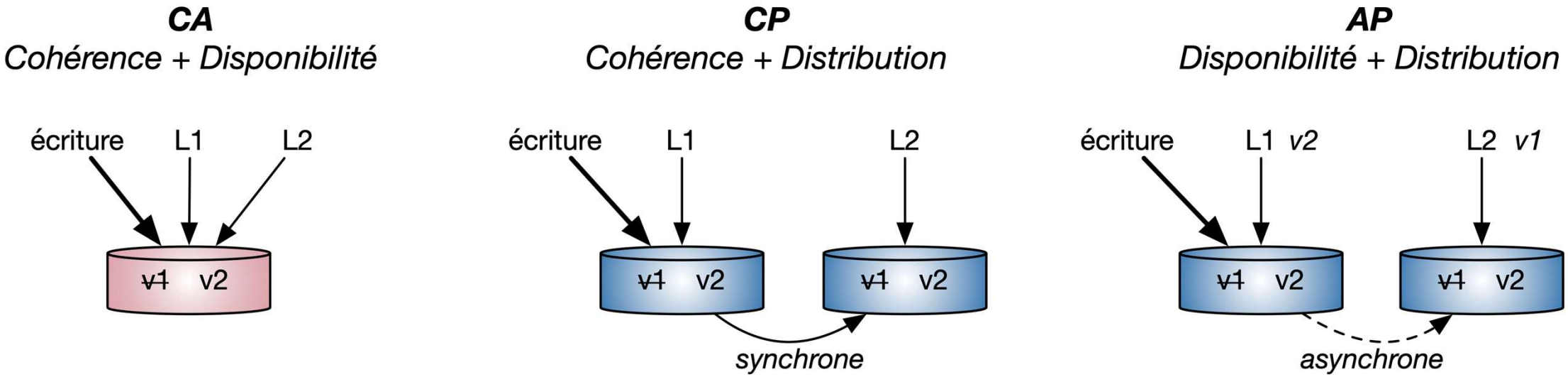
1. ***Consistency***:
   A data returns the proper value at any time (coherency)
2. ***Availability***:
   Even if a server is down, data is available
3. ***Partition Tolerance***:
   Even if the system is partitioned, a query must have an answer (unless for global failures)



Theorem: *A distributed, networked system can have only <u>two</u> of these three properties.*

The CAP Theorem

Illustration



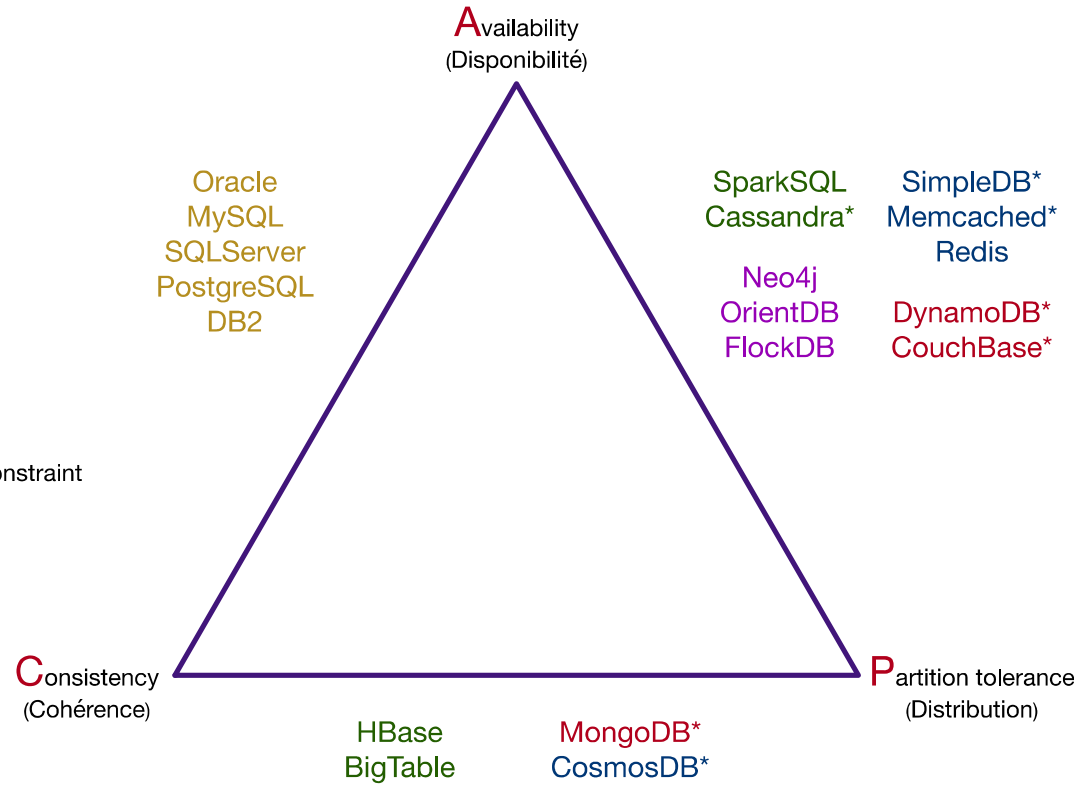
The CAP Theorem

CAP triangle

Models

Relational/Column
Key-Value
Wide Column
Document
Graph

* can change consistency constraint



Map/Reduce (shortly)

Distributed computation framework

2 main functions

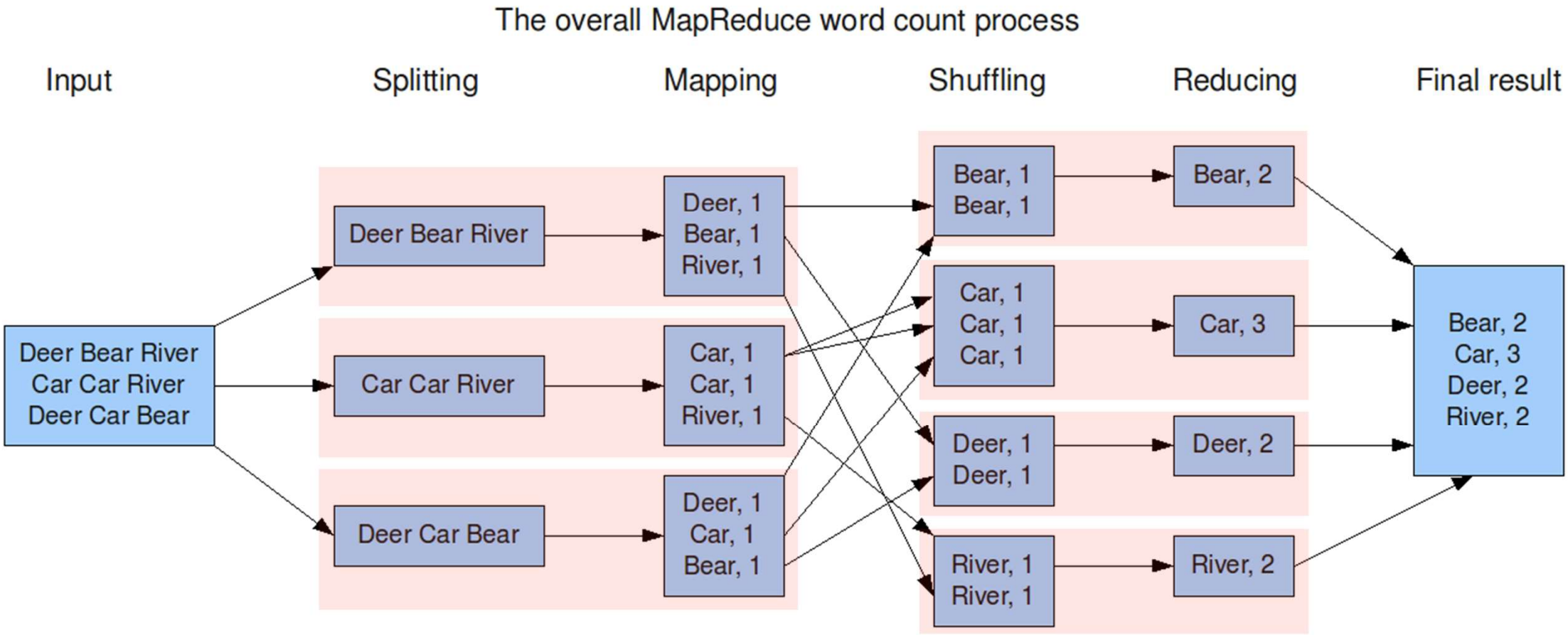
- ***Map***: data transformation
  - input: 1 data
  - output: several pairs (**key/value**)
- ***Reduce***: aggregate values for each key
  - *input : values for a given key :* ***key + list(values)***
  - *output : 1 value* - **key + value**

Scalability, Fault Tolerance

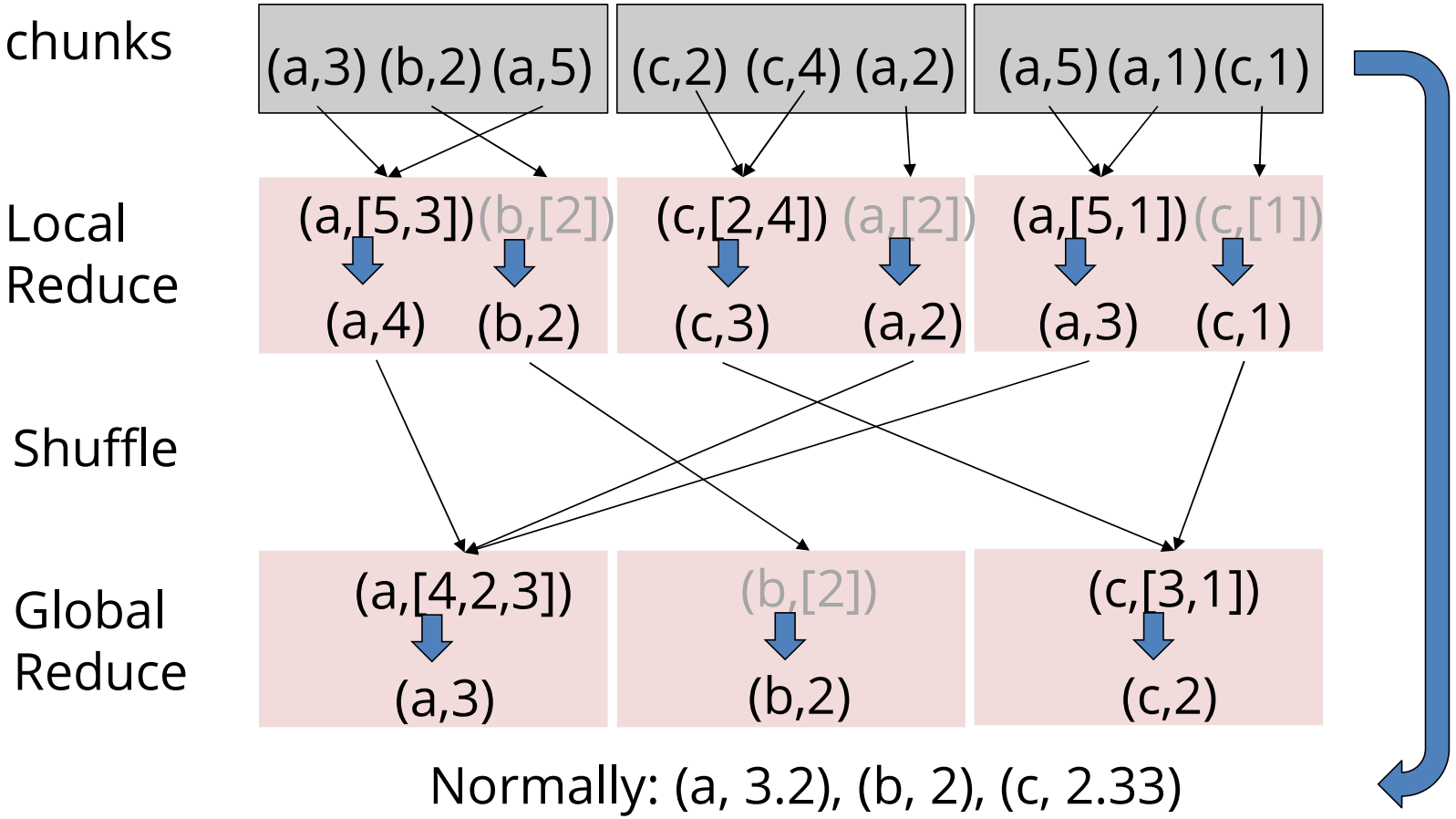
- *Send map&reduce to the cluster (jobs)*
- *In case of failure of 1 job,*

  restart the part of the job on another server

MapReduce: Word Count Execution



Reduce: Average function



MapReduce: Properties

Programming language

- Highly distributable
- Fault tolerant
- Simple to express

NoSQL -> on-top of MapReduce (operations are developped in MR)

Drawbacks

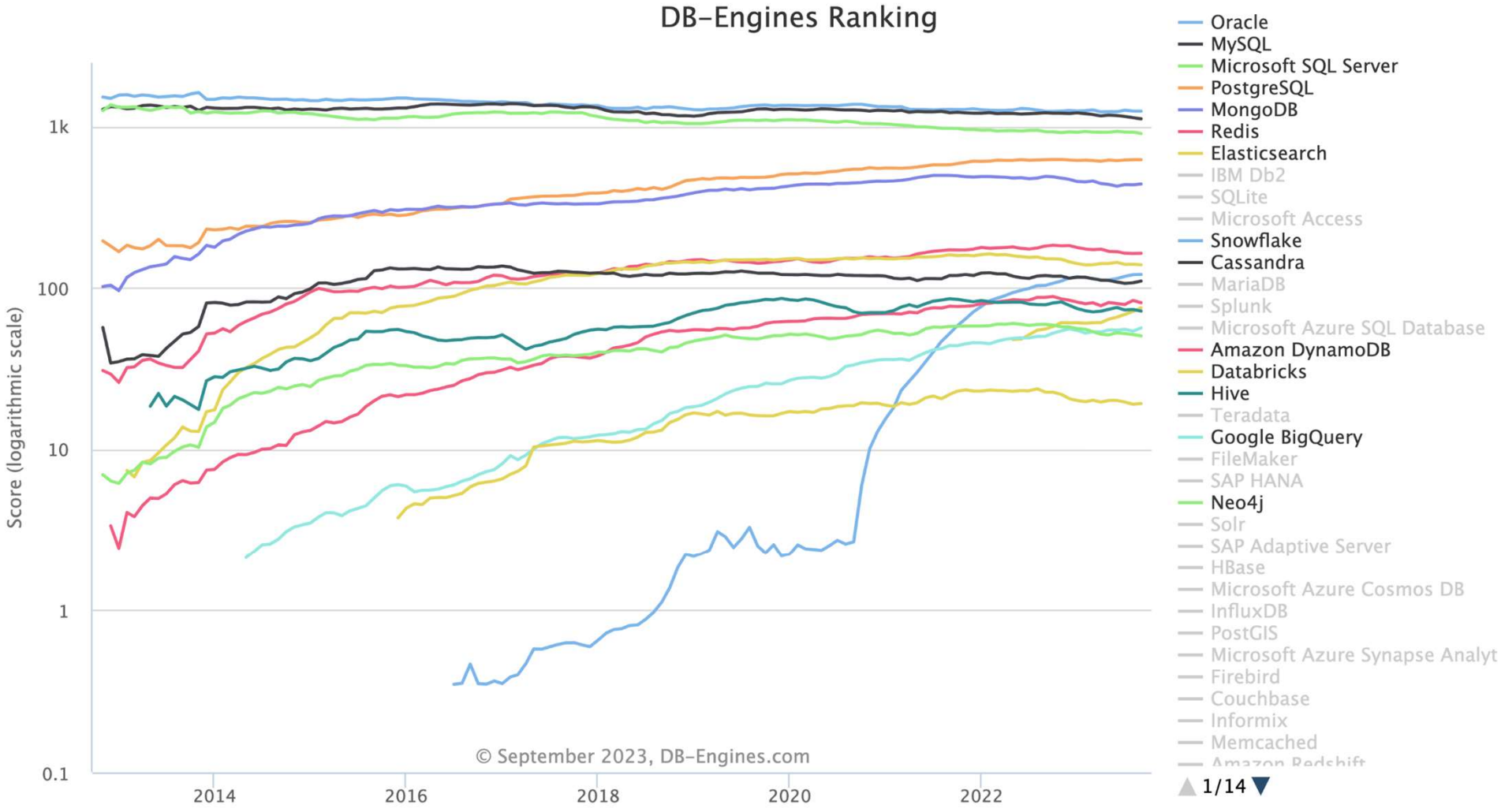
- No join queries
- Many networks communications (shuffle)

End

ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

https://db-engines.com/en/ranking



CAP vs PACELC
[Abadi 2012]

Data synchronization

- PAC
  Partitioning/Data movements
  Between Availability and Consistency
- ELC
  Normal state of the network
  Between Latency and Consistency

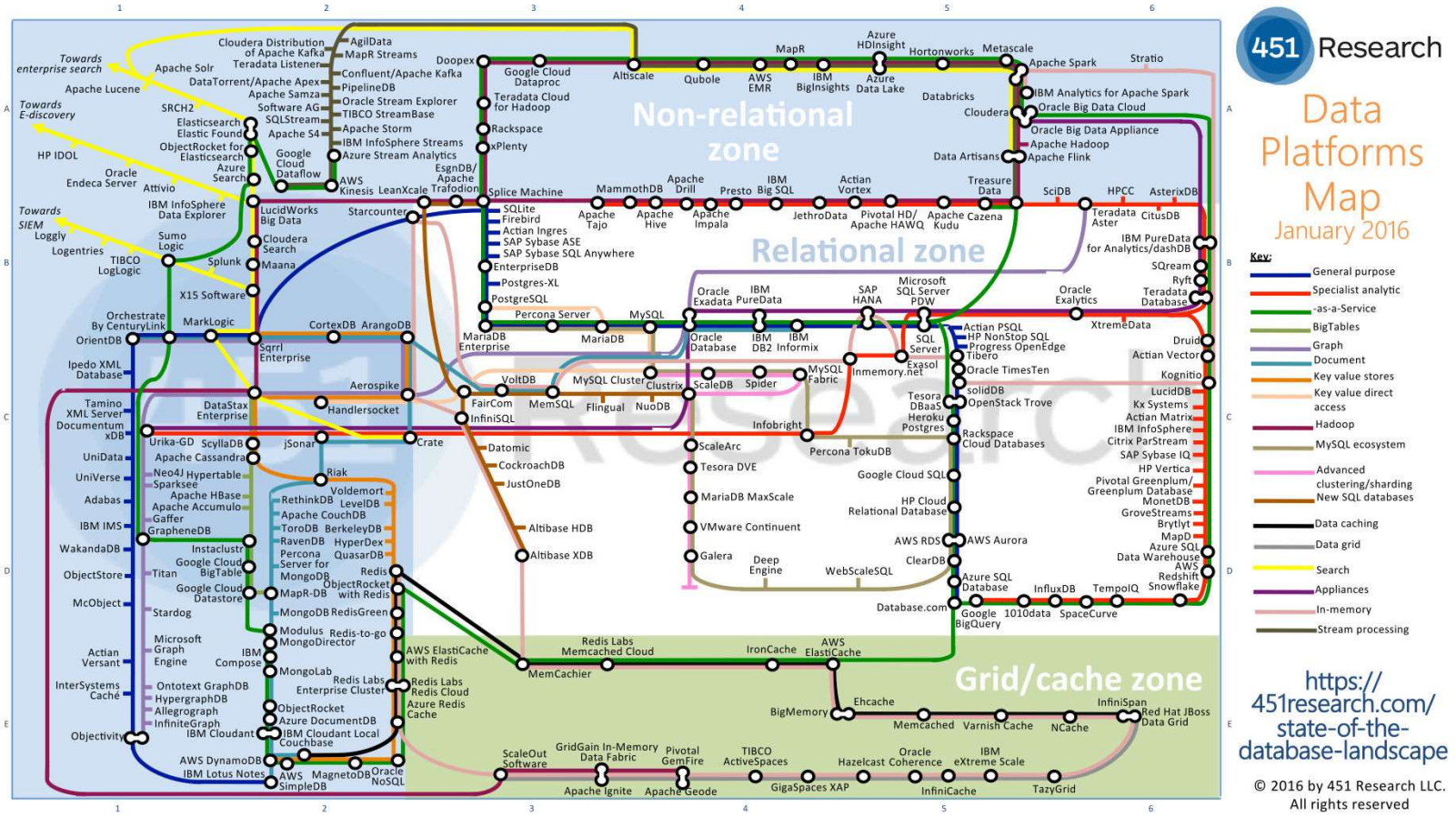
- **PA/EL**
  - *Dynamo, Cassandra, Riak*
  - Always **available** with a minimum of latency
- **PC/EC**
  - *VoltDB/H-Store, MegaStore*
  - **Consistency** whatever happens
- **PA/EC**
  - *MongoDB*
  - While partitioning: **availability**
  - In normal state: **consistency**

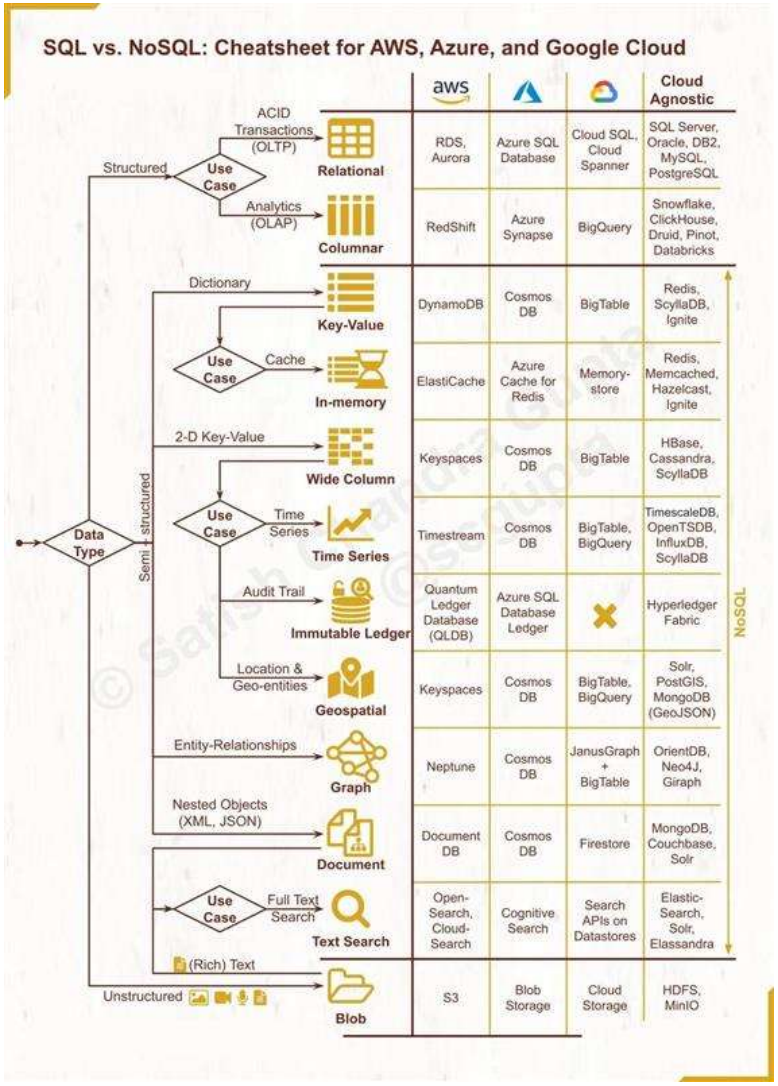
Database choice?

- Data Model
- Sharding strategy
- Consistency vs Efficiency
- Community

  +

- Change management
- Transformation cost





Credits - Satish Chandra Gupta

Javascript Object Notation

ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Semi-Structured Data - JSON
Javascript Object Notation

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

JSON: Javascript Object Notation

- XML initially designed for standardizing computer communications
  - Too verbose & space greedy
  - Dedicated to *Web Services*

vs

- JSON (JavaScript Object Notation)
  - Designed for web server to web browser communications
  - Lightweight, text oriented, language independent
  - Used for APIs (Google API, Twitter API) or dynamic web (Ajax)

JSON: Key-values & Typing

Key + Value

- "lastname" : "Mali"
- Keys with **quotations**

Identifiers

- **" _id "** commonly used
- Overwrite already stored ids
- Can be automaticaly generated
  - Ex MongoDB: "_id" : ObjectId(1234567890)

Objects/Documents

- Collection of **key/values – unique keys**
- { "_id" : 1234,
  "lastname" : "Mali",
  "firstname" : "Jihane", "kind" : 1}

Scalar

- String, Integer, float, boolean, null…

Documents

- Objects **{…}**

List

- Arrays **[** … **]**
- No typing
  "lessons" : **[** "SQL", 1, 4.2, null, "NoSQL" **]**
- Can nest documents
  "doc" : **[**{"test" : 1}**,**
  {"test" : {"nesting" : 1.0}}**,**
  {"key" : "text", "value" : null} **]**

JSON: Complete example

```
{
        "_id" : 1234,
        "lastname" : “Mali", "firstname" : “Jihane",
        "employers" : [
                { "company" : "ESILV", "starting_date" : "2023-12",
                  "location" : {"street" : "12 avenue Léonard de Vinci","city" : "Paris La Défense", "zip" : 92916}},
                { "company" : “ISEP", "starting_date" : "2022-06", "end_date" : "2023-12",
                  "location" : {"street" : "28 Rue Notre Dame des Champs","city" : "Paris", "zip" : 75006}},
                 ],
        "fields" : [ "DB" , "DB optimization", "XML", "NoSQL", "IR" ],
        "hobbies" : [“Cooking", “Traveling"]
}
```

JSON Schema

ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

JSON Schema

- Needs a Schema to validate documents' structure
- Automatic check (exists, inclusion, syntax, queries)

- Requires a typed grammar
  - Objects, arrays,
  - Mandatory/optional keys

http://json-schema.org/

JSON Schema : Objects

```
{
        "type" : "object",
         "properties" : {
                "key1" : {"type": … },
                "key2" : {"type": … }
        }
}
```

Object's keys

Types of values
"number" (option – "min", "max")
"string" (option - "pattern")
"anyOf"
"not"

Other options : "definitions" + "$ref"

JSON Schema : Keys

```
{
        "type" : "object",
         "properties" : {
                "key1" : {"type": … },
                "key2" : {"type": … }
        },
        "required" : ["key1"],
        "additionalProperties" : false
}
```

Mandatory keys
Missing = optional

Forbids additional keys

Other options : "minProperties", "maxProperties", "patternProperties"

Warning! Negation

```
{
        "type": "object",
        "properties": {
                "key1": { "type": "integer" }
        },
        "not": { "required": [ "key1" ] }
}
```

JSON Schema : Lists

```
"arrayKey" : {
        "type" : "array",
        "items" : [ {"type" : … } ],
        "minItems" : 1,
        "maxItems" : 10
}
```

Other options
"uniqueItems"
"additionalItems"

JSON Schema : Example

```
{ "type": "object",
  "properties": {
    "_id": { "type": "integer" },
    "lastname": { "type": "string" },
    "firstname": { "type": "string" },
    "employers": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "company": { "type": "string" },
          "starting_date": { "type": "string" },
          "end_date": { "type": "string" } },
...
          "location": {
            "type": "object",
            "properties": {
              "street": { "type": "string" },
              "city": { "type": "string" },
              "zip": { "type": "integer" } },
            "required": [ "street", "city", "zip" ] },
          "required": [ "company", "starting_date", "location" ] } },
    "fields": { "type": "array", "items": { "type": "string" } },
    "hobbies": {
      "type": "array",
      "items": { "type": "string" } } },
    "required": [ "_id", "lastname", "firstname", "employers",
                  "fields", "hobbies" ]
}
```

JSON Schema : Miscellaneous

- Programming language-oriented specification : **JOI**
  - https://github.com/hapijs/joi
- Schema Extraction
  - https://www.liquid-technologies.com/online-json-to-schema-converter
  - https://app.quicktype.io/#l=schema
- Schema validation
  - https://www.jsonschemavalidator.net/
- JSON examples generator from a JSON Schema
  - https://jsonschematool.ew.r.appspot.com/

Cassandra

ESILV

jihane.mali (at) devinci.fr
nicolas.travers (at) devinci.fr

1 Introduction

2 Data Model & Interrogation

3 Scalability & Fault Tolerance

4 Conclusion

What is Cassandra ?

Cassandra[1], originally designed by *Facebook* in 2008, is now an *Apache* project since 2010. The company *Datastax* distributes it and provides several other services.

- Initially based on Google's *BigTable* system
  $\Rightarrow$ **Column-oriented** storage;
- Now based on the *DynamoDB* system (hash-based techninc DHT)
  $\Rightarrow$ **"Key/Value" oriented** storage;
- Has evolved significantly towards an *extended relational model* (N1NF = *Non First Normal Form*);
- One of the few NoSQL systems offering **strong typing**;
- Schema and query definition language, CQL (no joins).

[1] `https://github.com/apache/cassandra/tree/trunk/lib`

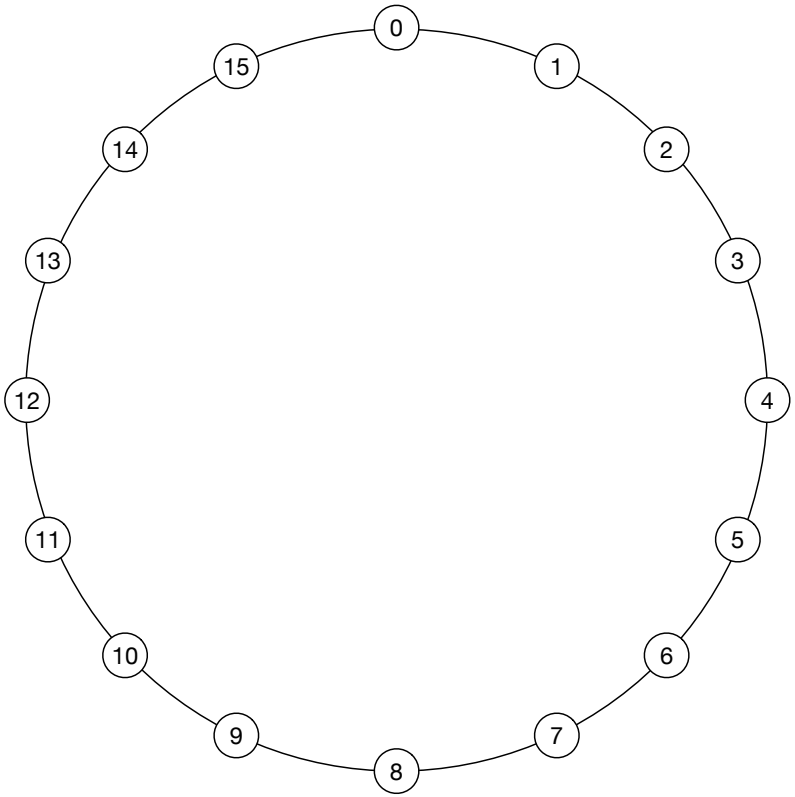
Scalable and distributed

Built as a scalable and distributed system.

- Offers scaling methods inspired by the Dynamo system (Amazon);
- Distribution through consistent hashing;
- Fault tolerance through multi-node replication.

Widely used and renowned for its high performance.

Distributed Hash Table



2^{64} nodes, skip by jumps (up to half the ring), replication, fault tolerance..

1 Introduction

2 Data Model & Interrogation
- Key concepts
- Create a database : Keyspace
- Denormalization choice
- Simple Queries
- Map/Reduce

3 Scalability & Fault Tolerance

4 Conclusion

Data Model = relational + complex types

- *Data bases* : Keyspace
- *Tables* : Table or Column Family
- *Rows* : Row (either simple or complex values)

Perspective A: it is extended relational by breaking the first normalization rule (atomic type).

Perspective B: Each row represents a structured document (nesting).

⚠ Caution ⚠

Confusing and sometimes misleading vocabulary: *Keyspace*, *columns*, *column families*...

Key-value Pairs and Documents

The basic structure is the pair (key, value). A piece of data is composed of:

- **Key**: an identifier
- **Value**: atomic (integer, string) or complex (dictionary, set, list)

A row (document) is identified by a key (row key).

Rows are typed by a schema, including nested data.

Every insert must validate the schema.

Create you own Cassandra Database

Create a **keyspace** :

```
CREATE KEYSPACE IF NOT EXISTS Compagnie
WITH REPLICATION={'class':'SimpleStrategy','replication_factor':3};
```

Create a **Column Family** / table :

```
CREATE TABLE Vols (
    idVol INT, ligneID INT, dateDepart DATE, distance INT,
    duree FLOAT, pilote INT, copilote INT, officier INT,
    ChefCabine1 INT, ChefCabine2 INT,
    primary key (idVol)
);

CREATE INDEX vol_pilote ON Vols (pilote);
```

Insert

Like in SQL

```
INSERT INTO Vols (idVol, ligneID, dateDepart, distance, duree,
    pilote, copilote, officier, ChefCabine1, ChefCabine2)
    VALUES (1, 1, '2016-10-15', 344, 1.3, 1, 2, 3, 4, 5);
```

Insert a JSON document

```
INSERT INTO Vols JSON '{
    "idVol" : 1, "ligneID" : 1, "dateDepart" : "2016-10-15",
    "distance":344,  "duree" : 1.3, "pilote" : 1, "copilote" : 2,
    "officier" : 3, "ChefCabine1" : 4, "ChefCabine2" : 5}';
```

Import a CSV file

```
COPY Compagnie.Vols(idVol, ligneID, dateDepart, distance, duree,
   copilote, officier, chefCabine1, chefCabine2)
FROM 'vols.csv' WITH HEADER=TRUE AND DELIMITER = ',';
```

Querying language

The **CQL** language: *Cassandra Query Language*
Very close to SQL (simplified version for NoSQL)."

```
SELECT * FROM Vols WHERE idVol = 1;
```

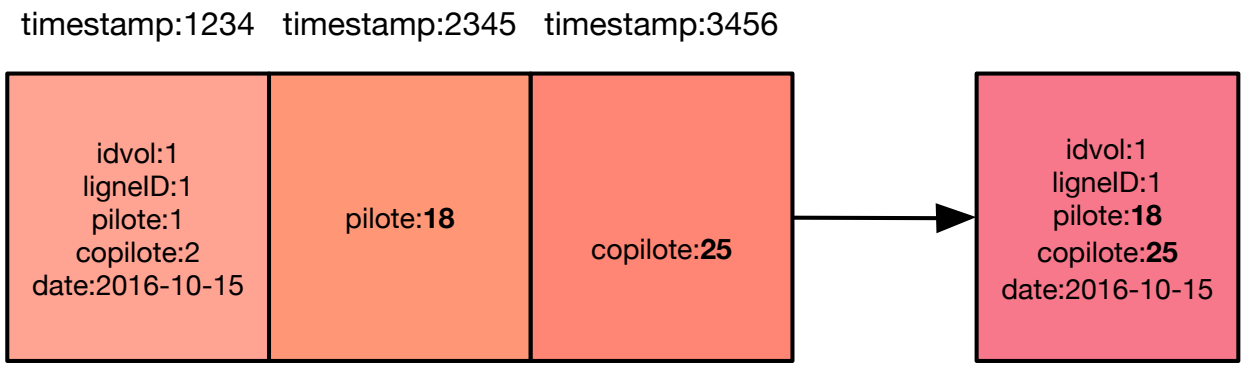
Result :

```
idVol | ligneID | dateDepart  | distance |  duree | pilote | copilote | officier | ChefCabine1 | ChefCabi
-------------------------------------------------------------------------------------------------------
    1 |       1 |  2016-10-15|      344 |    1.3 |      1 |        2 |        3 |           4 |
```

Temporal Data

Every value is associated with a TIMESTAMP. For every update a timestamp (ms) is generated. Can be specified with the query:

```
UPDATE Vols USING TIMESTAMP 2345
    SET pilote=18 WHERE idVol = 1;
```



Temporal Data

Retrieve update time:

```
SELECT writetime(depart), writetime(pilote), writetime(copilote)
FROM Vols WHERE idVol=1;
```

```
writetime(depart) | writetime(pilote) | writetime(copilote)
------------------------------------------------------------
             1234 |              2345 |                3456
```

We can delete a defined value at a given moment T in time:

```
DELETE pilote USING TIMESTAMP 1234 FROM Vols WHERE idVol=1;
```

This value cannot be queried in the WHERE clause.

Temporary Data

Possible to manage volatile data: TTL[2]
Used similarly to TIMESTAMP, it indicates the number of seconds during which the value is visible.

```
UPDATE Vols USING TTL 3600
    SET pilote=18 WHERE idVol = 1;
```

[2] Time To Live

NoSQL : No joins

Due to distributed data, a join is not feasible.
It is therefore preferable to aggregate the data as much as possible in rows.

- Query-oriented design based on the most frequent application queries.
- Often favor the largest dimension for better distribution.
- ⚠The new schema introduces redundancy and potential inconsistency.

At worst, we represent the same data in multiple forms (materialized views).

- All flights with their flight attendants (a feasible solution).
- For each flight attendant, the list of her flights.

Nesting : solutions

An attribute can be typed for nesting (XXX):

1. **SET** : set of unique values (unordered)
2. **LIST** : list of values (ordered)
3. **MAP** : set of key/value pairs (nested document)
4. **TYPE** : typed nested row (whole row)
5. **TUPLE** : a nested typed row without keys

```
CREATE TABLE Vols (
    idVol INT, ligneID INT, dateDepart DATE, distance INT,
    duree FLOAT, pilote INT, copilote INT, officier INT,
    ChefCabine1 INT, ChefCabine2 INT,
    hotesses XXX <YYY>, aeroports XXX <YYY>,
    primary key (idVol)
);
```

Hotesses **SET**<int>

- *Insert* :

  INSERT INTO Vols (..., hotesses) VALUES (..., {6, 7, 8});

- *Update* :

  UPDATE Vols SET hotesses = hotesses + {9} WHERE idVol = 1;
  UPDATE Vols SET hotesses = hotesses - {8} WHERE idVol = 1;
  UPDATE Vols SET hotesses = {10} WHERE idVol = 1;

  DELETE hotesses FROM Vols WHERE idvol = 1;

- *Project the set* :

  SELECT idVol, hotesses FROM Vols WHERE idVol = 1;

```
idVol | hotesses
------------------
    1 | {6, 7, 8}
```

Hotesses **LIST**<int>

- *Insert* :

  INSERT INTO Vols (..., hotesses) VALUES (..., [6, 7, 8]);

- *Update* :

  UPDATE Vols SET hotesses = hotesses + [9] WHERE idVol = 1;
  UPDATE Vols SET hotesses[1] = 8 WHERE idVol = 1;
  UPDATE Vols SET hotesses = [10] WHERE idVol = 1;

  DELETE hotesses[0] FROM Vols WHERE idvol = 1;

- *Project the list* :

  SELECT idVol, hotesses FROM Vols WHERE idVol = 1;

```
idVol | hotesses
------------------
    1 | [6, 7, 8]
```

Aeroports **MAP**<text, text>

- *Insert* :
  INSERT INTO Vols (..., aeroports) VALUES (...,
  {'depart' : 'CDG', 'arrivee' : 'LCY'}]);
- *Update* :
  UPDATE Vols SET aeroports = aeroports + {'escale' : 'TXL'}
  WHERE idVol = 1;
  UPDATE Vols SET aeroports['escale'] = 'NTE'
  WHERE idVol = 1;
  DELETE aeroports['escale'] FROM Vols WHERE idvol = 1;
- *Project the map* :
  SELECT idVol, aeroports FROM Vols WHERE idVol = 1;

```
idVol |                              aeroports
-------------------------------------------
    1 | {"depart":"CDG", "arrivee":"LCY"}
```

Depart frozen<aeroportType>

- First, create a *data type* for nesting

  CREATE TYPE aeroportType (IATA varchar, nom varchar, lat float, long float);

  It is necessary to "*freeze*" the type : (frozen<aeroportType>).

- *Insert* :

  INSERT INTO Vols (..., depart, arrivee) VALUES (...,
  {IATA:"CDG", nom:"Charles de Gaulle", lat:49.0097, long:2.5479},
  {IATA:"LCY", nom:"London City", lat:51.5035, long:0.0487});

- *Update* :

  UPDATE Vols SET depart["nom"] = "Aeroport Charles de Gaulle"
  WHERE idVol = 1;

  DELETE arrivee FROM Vols WHERE idvol = 1;

- *Project IATA codes* :

  SELECT idVol, depart.IATA, arrivee.IATA FROM Vols WHERE idVol=1;

```
idVol | depart.IATA | arrivee.IATA
-----------------------------------
    1 |         CDG |          LCY
```

Depart tuple<text, text, float, float>

- Create a *typed* row to nest, but without keys.
- *Insert* :
  INSERT INTO Vols (..., depart, arrivee) VALUES (...,
  ("CDG", "Charles de Gaulle", 49.0097, 2.5479),
  ("LCY", "London City", 51.5035, 0.0487));
- *Update* :
  UPDATE Vols SET depart = ("TXL", "Berlin", 52.5580, 13.2847)
  WHERE idVol = 1;
  DELETE depart FROM Vols WHERE idvol = 1;
- *Project the tuple* :
  SELECT idVol, depart FROM Vols WHERE idVol=1;

```
idVol | depart
-------------------------------------
    1 |    ("CDG", "Charles de Gaulle", 49.0097, 2.5479)
```

Nesting choice

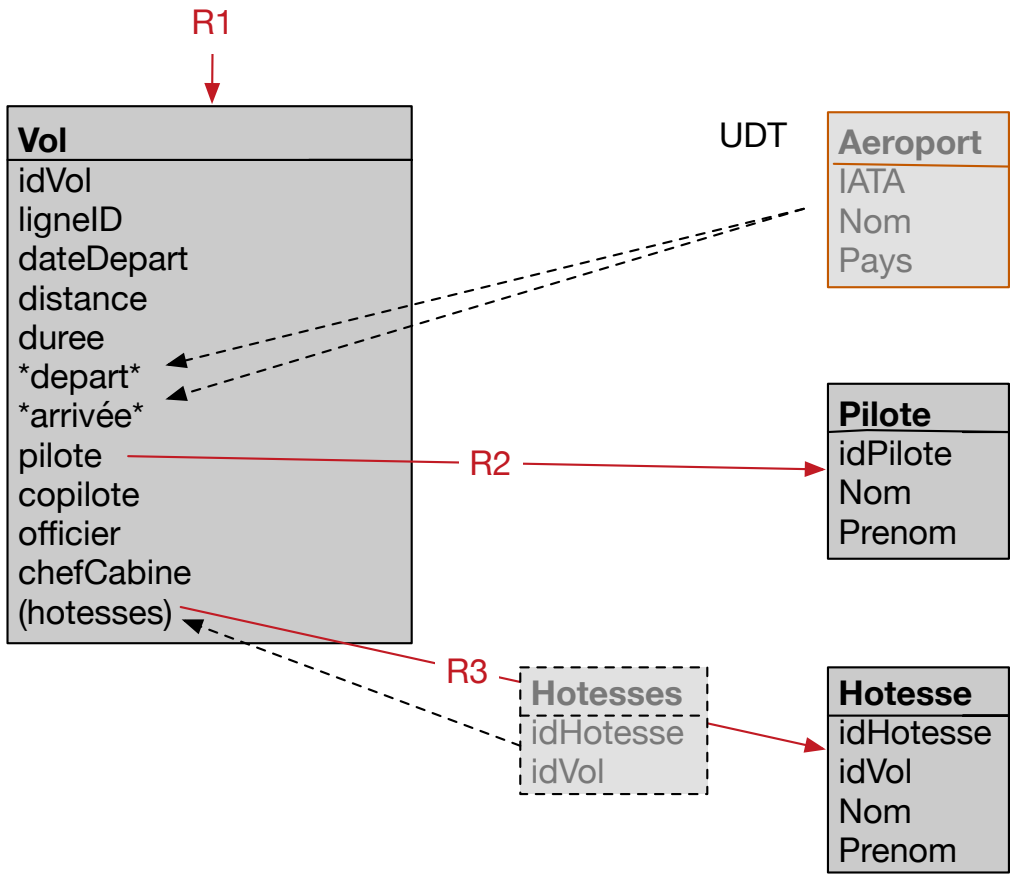
How to choose the nesting?

1. Chebotko Diagram[3]: *Query-Driven methodology*
   - Defining the access pattern to the tables
   - Nesting in the way that queries are applied
   - Data model adapted for each nesting
   - ⚠Query choice
2. Focus on the largest dimension (number of rows)
   - Fewer updates
   - Better distribution
   - Improved schema compatibility
   - ⚠Complex data model

[3][Chebotko et al. 2015] https://pdfs.semanticscholar.org/22c6/740341ef13d3c5ee52044a4fbaad911f7322.pdf

Query-Driven Modelling

- List []
- Set ()
- Map {}
- UDT **



CQL 3.3

- `SELECT` ...
  Attributes
  DISTINCT, COUNT(*) for primary key and indexed attributes
- `FROM`
  A single table
- (`WHERE` ...)?
  Detailed in next slide
- (`ORDER BY` ...)?
  Only on *partitioning key/primary key*
- (`LIMIT` ...)?
  Only on *partitioning key/primary key*
- (`ALLOW FILTERING`)?
  Detailed in next slide

Clause *WHERE*

- **Primary key = value** : Most efficient
- **token(Primary key) = valeur** : Hashing function
- **Attribute = value + INDEX** : Less efficient
- **Attribute = value + ALLOW FILTERING** : Totally inefficient
  broadcast on all servers
- **Queries on nested data types** :
  - SET : *CONTAINS*
  - LIST : *CONTAINS*
  - MAP : *CONTAINS / CONTAINS KEY*
  - TYPE/TUPLE : attribute = Whole nested value

  $\Rightarrow$ *SET* and *MAP* are better to use

Aggregates

How to perform complex calculations?

- Use Map/Reduce to aggregate data
  - Two-part program: filtering + grouping
  - *Map*: takes a row and produces a **key/value** as output
  - *Reduce*: takes a key (from the map) with the list of values and produces an output value (for the key).
- In *Cassandra*:
  - Java program: **User-Defined Aggregate Function**[4]

[4] UDA: `https://docs.datastax.com/en/cql/3.3/cql/cql_using/useCreateUDA.html`

User-Defined Aggregate Function

Map

```
CREATE OR REPLACE FUNCTION avgState ( state tuple<int,bigint>, val int )
    CALLED ON NULL INPUT RETURNS tuple<int,bigint> LANGUAGE java
    AS 'if (val !=null) { state.setInt(0, state.getInt(0)+1);
                          state.setLong(1, state.getLong(1)+val.intValue()); }
        return state;';
```

Reduce

```
CREATE OR REPLACE FUNCTION avgFinal ( state tuple<int,bigint> )
    CALLED ON NULL INPUT RETURNS double LANGUAGE java
    AS 'if (state.getInt(0) == 0) return null;
        return Double.valueOf( state.getLong(1) / state.getLong(0) );';
```

UDA

```
CREATE AGGREGATE IF NOT EXISTS average ( int )
    SFUNC avgState STYPE tuple<int,bigint>
    FINALFUNC avgFinal INITCOND (0,0);

SELECT average (duree) FROM Vols WHERE ligneID = 1;
```

Sahrding : *Partitioning Key*

- By default : Partitioning is done on the Primary key

```
PRIMARY KEY (idVol);
```

- Primary keys can be composite (must query with both values)

```
PRIMARY KEY (ligneID, idVol);
```

- *Partitioning Key* :

```
PRIMARY KEY ( (ligneID), idVol);
```

  - Flights corresponding to the same *ligneID* are **all** placed on the same server
  - Instances of the same *ligneID* are placed in a file `sstable` and sorted according to *idVol*

Indexing

Secondary indexes can be created:

```
CREATE INDEX <index_name> ON <table_name> (<attribute>);
```

- On MAP nested data types:
  - Find values in the nested document. Query:

```
WHERE attribute CONTAINS <value>
```

  - Find existing keys:

```
CREATE INDEX <index_name> ON <table_name> (keys(<attribute>));
```

Query:

```
WHERE attribute CONTAINS KEY <key_name>
```

- Only one index per attribute

1 Introduction

2 Data Model & Interrogation

3 Scalability & Fault Tolerance
- Distribution & Replication
- Consistency

4 Conclusion

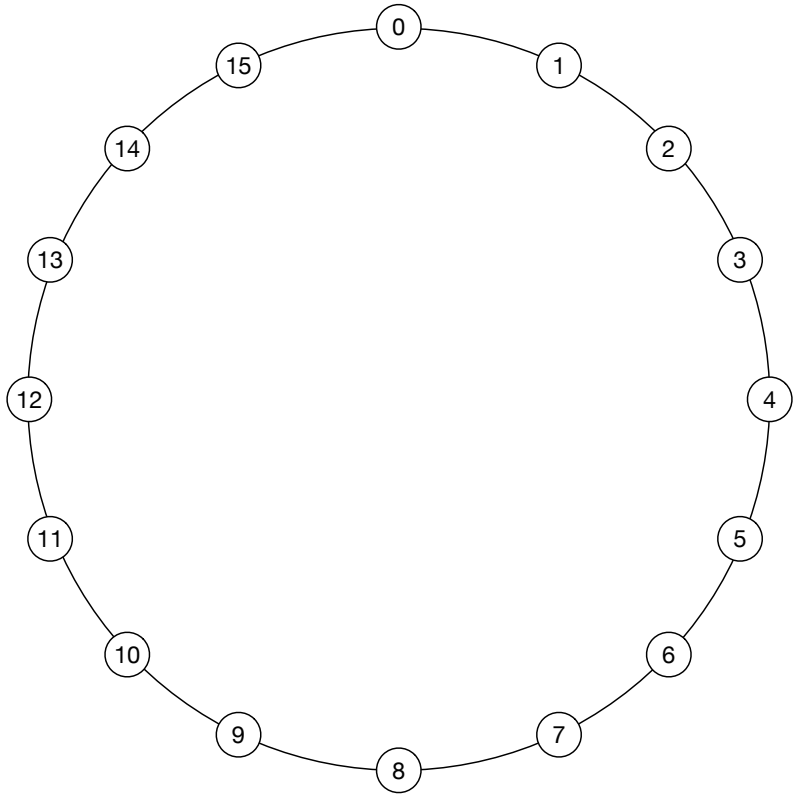
Distribution & Replication

DHT: Distributed Hash Table

- Distribution:
  - A ring of 2^{64} nodes
  - Routing by jumps (hash table)
- Replication:
  - Fault tolerance,
  - Data replicated 3 times (locally and 2 previous nodes)
  - *Replication Factor* (3 by default)
  - Replicates can respond to queries $\Rightarrow$ Consistency ?

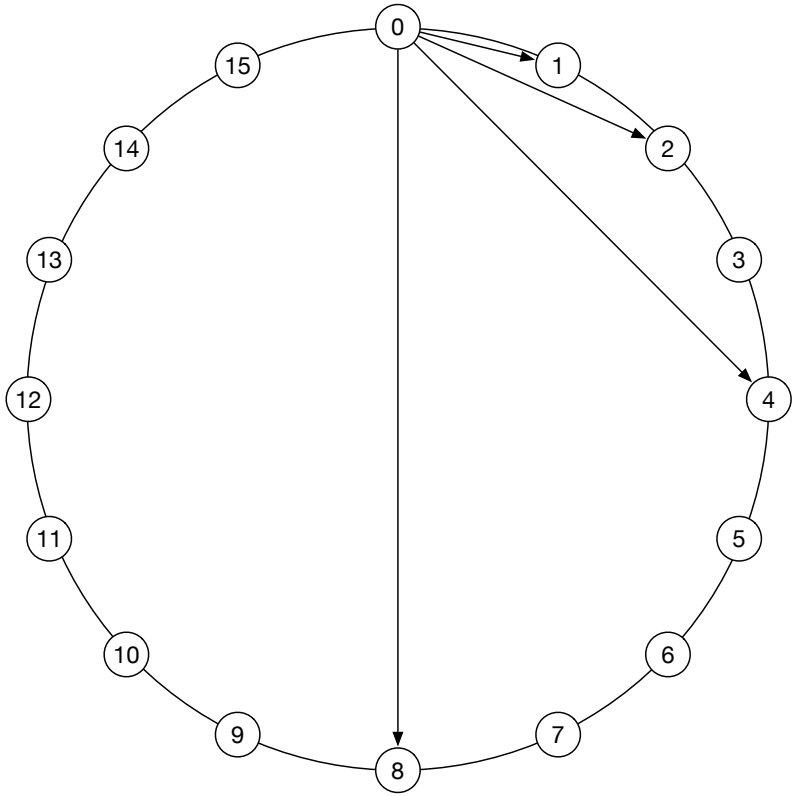
DHT

- Distribution
- Elasticity
- Decentralized
- Rapidity (max $log_2(N)$ jumps)
- Replication



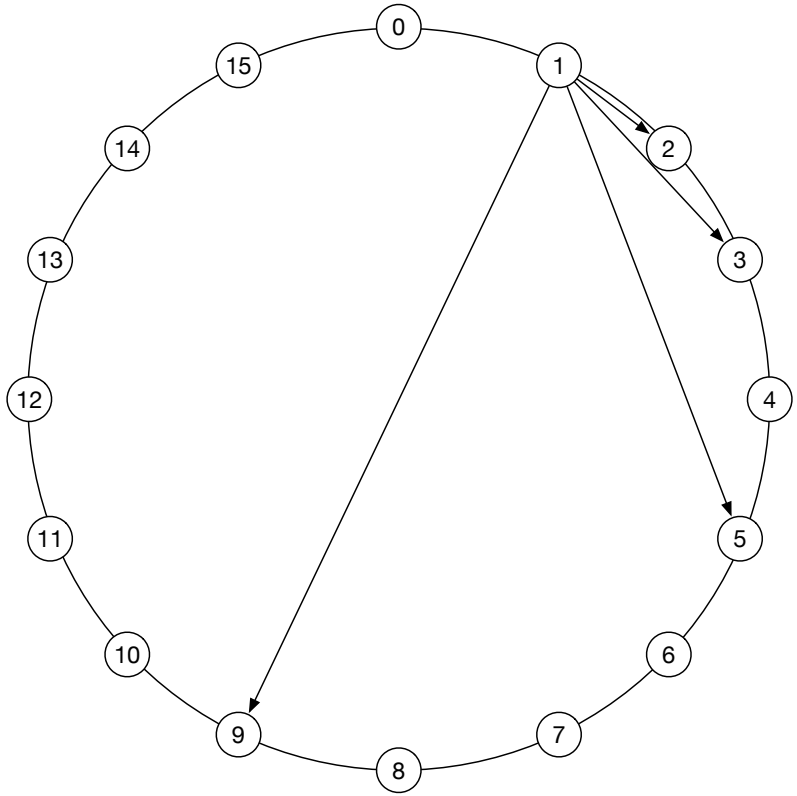
DHT

- Distribution
- Elasticity
- Decentralized
- Rapidity (max $log_2(N)$ jumps)
- Replication



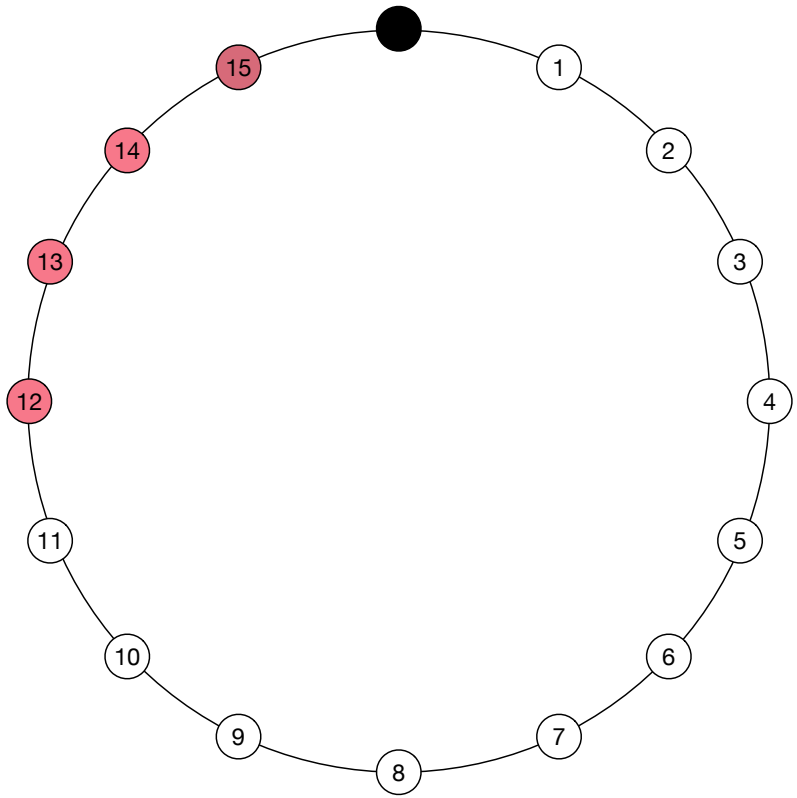
DHT

- Distribution
- Elasticity
- Decentralized
- Rapidity (max $log_2(N)$ jumps)
- Replication



DHT

- Distribution
- Elasticity
- Decentralized
- Rapidity (max $log_2(N)$ jumps)
- Replication



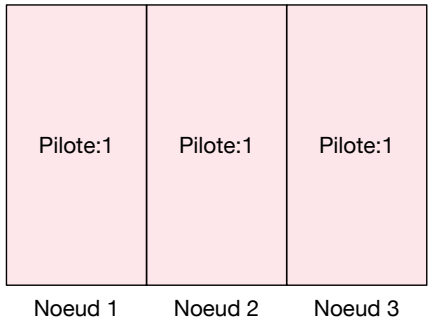
Consistency : Quorum

R + W > N

- **N** : Replication rate
- **W** : Minimum Nb of acknowledged writes
- **R** : Minimum Nb of reads
- Parallel writes and reads
- If W + R <= N ⇒ Eventual Consistency

Example

- Let replication rate be N=3
- Update: pilote=18



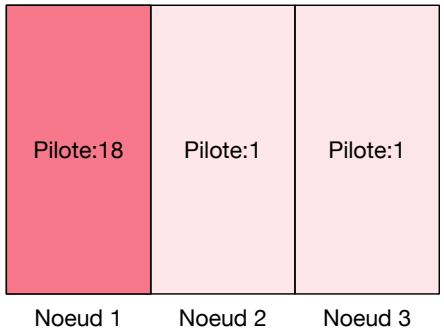
Consistency : Quorum

R + W > N

- **N** : Replication rate
- **W** : Minimum Nb of acknowledged writes
- **R** : Minimum Nb of reads
- Parallel writes and reads
- If W + R <= N ⇒ Eventual Consistency

Example

- Let replication rate be N=3
- Update: pilote=18
  1 acknowledgement (**W=1**)



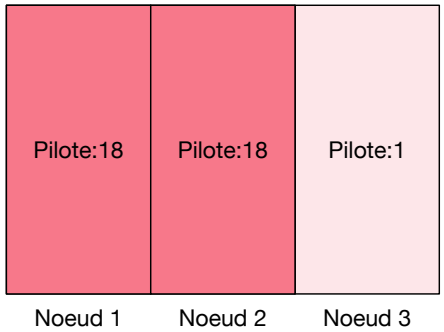
Consistency : Quorum

R + W > N

- **N** : Replication rate
- **W** : Minimum Nb of acknowledged writes
- **R** : Minimum Nb of reads
- Parallel writes and reads
- If W + R <= N ⇒ Eventual Consistency

Example

- Let replication rate be N=3
- Update: pilote=18
  2 acknowledgements (**W=2**)
- 2 parallel reads (**R=2**)



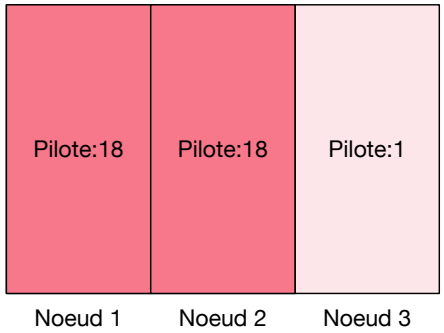
Consistency : Quorum

R + W > N

- **N** : Replication rate
- **W** : Minimum Nb of acknowledged writes
- **R** : Minimum Nb of reads
- Parallel writes and reads
- If W + R <= N ⇒ Eventual Consistency

Example

- Let replication rate be N=3
- Update: pilote=18
- 2 parallel reads (**R=2**)
- 2+2>3 ⇒ returns : 18

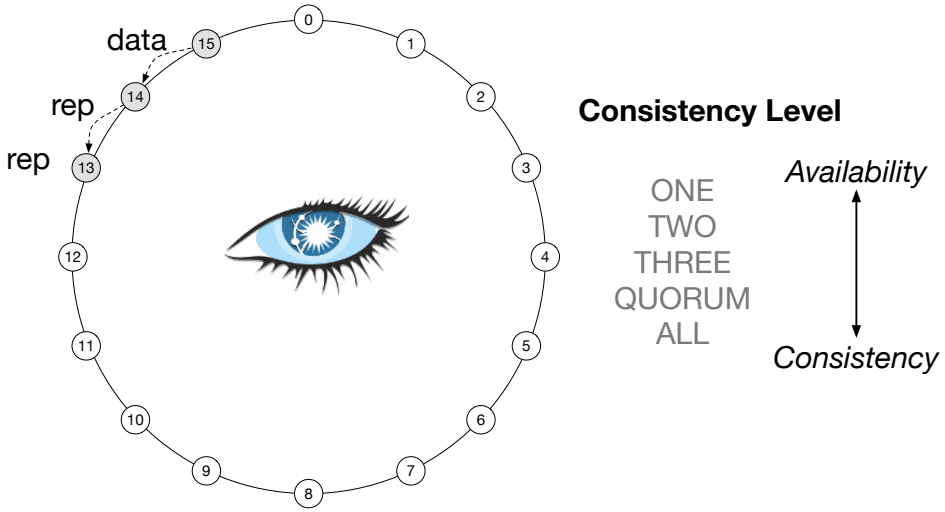


Consistency : Management

```
CONSISTENCY <level> ;
```

level : ANY, ONE, TWO, THREE, QUORUM, ALL

- **ONE/ANY** : by default, at least 1 acknowledgement
- **QUORUM** : more reads but better consistency
- **ALL** : Every replicate must be acknowledged



1 Introduction

2 Data Model & Interrogation

3 Scalability & Fault Tolerance

4 Conclusion

Conclusion

- Cassandra is a solution dedicated to time series clustering $\Rightarrow$ *Partitioning Key*
- Complete self-management of the network
- The simplicity of the language conceals the complexity of use case integration

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

ESILV
ENGINEERING SCHOOL
DE VINCI PARIS

Introduction

Distributed NoSQL database

- Large data management (Hu**mongo**us)
- Document-oriented NoSQL
  - JSON documents
  - (BSON Objects serialization)

Key points:

- Simple to integrate for developers
- Rich query language
  - MQL: Mongo Query Language
- Several optimization features
  - Attributes indexing (Shard/Btree/RTree)
- Smart data allocation
  - Sharding (GridFS) [+ tagging / clustering]
- Several deployement
  - Private/Public Cloud, Local (On Premise)

Applications with MongoDB

Metlife : unified view
Cisco : e-commerce
Bosch : IoT
HSBC : digital transformation
The Weather Channel : mobility
Expedia : trip recommendations
ebay : products
AstraZeneca : medics logging
Comcast : Database-as-a-service
KPMG : Analytics
X.ai : AI
EA : videos games

Interactions: JavaScript

JS objects

- Attributes + functions
- *db*: database
- *sh*: sharding
- *rs*: replica set

MQL

- Pattern = "JSON"

MapReduce

- Functions (untyped)

Studio3t

Robo3t

MongoDB Compass

Starting Commands

Select database (shell)

Command: > use **myBD**

Collections of documents

Create: > db.createCollection('***users***');

Manipulate: > db.**users**. <command> ;

Commands : find(), count(), save(), delete(), update(), aggregate(), distinct(), mapReduce()…

Equivalent in SQL: *FROM*

Documents

JSON:

Insert: > db.users.save ( ); //no quotes!

```
{
    "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
    "name":  "George Lucas", "login": "george", "age": 76,
    "address": {"street":  "1 ranch Skywalker",
                "city":  "San Francisco"},
    "job" : [  "director",  "productor"]
}
```

MQL: find queries[1] (1/2)

- Document oriented queries
- Command: > db.users.**find**(<filter> , <projection>);
- Filter:
  - "JSON" pattern that must be *matched* on the document
  - "Key/Value" format
  - Can embed exact matching, operations, nesting, arrays
    - Operations: **$**op (no quotes)
  - Equivalent in SQL: *WHERE*
- Projection:
  - "Key/Value" pairs that are given in output
  - Equivalent in SQL: *SELECT (no aggregates)*
- Example:

  > db.users.find({"login" : "george"} , {"name" : 1, "age" : 1});

1 - find queries are simple queries for your report

MQL: find queries (2/2)

Exact match:

```
> db.users.find( { “name” : ”George Lucas" } , {“login” : 1, “age” : 1});
```

With nesting "." for the path to the given key

```
> db.users.find( { “address.city” : ”San Francisco" } );
```

Operations

```
> db.users.find( { “age” : { $gt : 50 } } );
//$gt, $gte, $lt, $lte, $ne, $in, $nin, $or, $and, $exists, $type, $size, $cond…
```

Regular expressions[1]

```
> db.users.find( { ”name” : {$regex : “lucas”, $options : “i" }} ); //or : {“name” : /lucas/i}
```

Arrays

```
> db.users.find( { ”job” : “director”} );          //Check in the list
> db.users.find( { ”job.1” : “productor”} );       //2nd position in the list
> db.users.find( { ”job” : [“director”]} );        //List exact match
```

1 - regex : https://docs.mongodb.com/manual/reference/operator/query/regex/

MQL: Distinct - Count

List of distinct values from a key

```
> db.users.distinct( “name” );
> db.users.distinct( “address.city” );
```

Number of documents

```
> db.users.count();
> db.users.find( { “age” : 76 }).count();
```

MQL: pipeline[2] (1/3)

- aggregate() : Ordered sequence of operators *aggregation pipeline*
- Command:

  > db.users.**aggregate([{$op1 : {}}, {$op2 : {}}, …]);**
- Operators:
  - $match : simple queries //equivalent in SQL: WHERE
  - $project : projections // equivalent in SQL: SELECT
  - $sort : sort result set // equivalent in SQL: ORDER BY
  - $unwind : normalization in 1NF
  - $group : group by value + aggregate function // equivalent in SQL: GROUP BY + fn
  - $lookup : left outer join (since v3.2 – work locally)
  - $out : store the output in a collection (since v3.2)
  - $geoNear : sort by nearest points (lat/long) // only as 1st pipeline (need to use an index)
  - …

2 – Pipeline queries (aggregate) are complex queries. For long pipelines, it can be hard queries

MQL: pipeline (2/3)

Pipeline : Each step (operator) gives its output to the following operator (input)

```
> db.users.aggregate([{$match : {"address.city" :   "San Francisco"}},
                      {$project : {"login" : 1, "age" : 1}},
                      {$sort : {"age" : 1, "login" : -1}}
          ]);
```

$unwind

Unnest an array and produce a new document for each item in the list

```
> db.users.aggregate([ {$unwind :  "$job"}} ]);
```

MQL: pipeline (2/3)

Pipeline : Each step (operator) gives its output to the following operator (input)

```
> db.users.aggregate([{$match : {"address.city" :   "San Francisco"}},
                      {$project : {"login" : 1, "age" : 1}},
                      {$sort : {"age" : 1, "login" : -1}}
                ]);
```

$unwind

Unnest an array and produce a new document for each item in the list

```
> db.users.aggregate([ {$unwind :  "$job"}} ]);
```

```
{
    "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
    "name":  "George Lucas", "login": "george", "age": 76,
    "address": {"street":  "1 ranch Skywalker", "city":  "San Francisco"},
    "job" : [  "director",  "productor"]
}
```

```
{
    "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
    "name":  "George Lucas", "login": "george", "age": 76,
    "address": {"street":  "1 ranch Skywalker", "city":  "San Francisco"},
    "job" : "director"
}
```

```
{
    "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
    "name":  "George Lucas", "login": "george", "age": 76,
    "address": {"street":  "1 ranch Skywalker", "city":  "San Francisco"},
    "job" :  "productor"
}
```

MQL: pipeline (3/3)

$group : key (_id) + aggregate ($sum / $avg / …)

No grouping key: *only one output*

```
> db.users.aggregate([ {$group : {"_id" : "age", "res": {$sum : 1}}} ]);
```

Group by value: ***$key***

```
> db.users.aggregate([ {$group:{"_id" : "$age", "res": {$sum : 1}}} ]);
```

Apply an aggregate function: ***$key***

```
> db.users.aggregate([{$group:{"_id":"$address.city", "avg": {$avg: "$age"}}} ]);
```

Sequence example

MQL: pipeline (3/3)

{"_id" : "age", "res": 10000}

$group : key (_id) + aggregate ($sum / $avg / …)

No grouping key: *only one output*

```
> db.users.aggregate([ {$group : {"_id" : "age", "res": {$sum : 1}}} ]);
```

Group by value: ***$key***

```
> db.users.aggregate([ {$group:{"_id" : "$age", "res": {$sum : 1}}} ]);
```

{"_id" : 23, "res": 550}
{"_id" : 22, "res": 600}
{"_id" : 24, "res": 700}

Apply an aggregate function: ***$key***

```
> db.users.aggregate([{$group:{"_id":"$address.city", "avg": {$avg: "$age"}}} ]);
```

Sequence example

MQL: pipeline (3/3)

{"_id" : "age", "res": 10000}

$group : key (_id) + aggregate ($sum / $avg / …)

No grouping key: *only one output*

```
> db.users.aggregate([ {$group : {"_id" : "age", "res": {$sum : 1}}} ]);
```

Group by value: ***$key***

```
> db.users.aggregate([ {$group:{"_id" : "$age", "res": {$sum : 1}}} ]);
```

{"_id" : 23, "res": 550}
{"_id" : 22, "res": 600}
{"_id" : 24, "res": 700}

Apply an aggregate function: ***$key***

```
> db.users.aggregate([{$group:{"_id":"$address.city", "avg": {$avg: "$age"}}} ]);
```

Sequence **example**

```
> db.users.aggregate([
        {$match:    {"address.city" : "San Francisco"} },
        {$unwind : "$job" },
        {$group :    {"_id" : "$job", "val": {$avg: "$age"}} },
        {$match :   {"val" : {$gt : 50}} },
        {$sort :      { "val" : -1} } ]);
```

MQL: updates

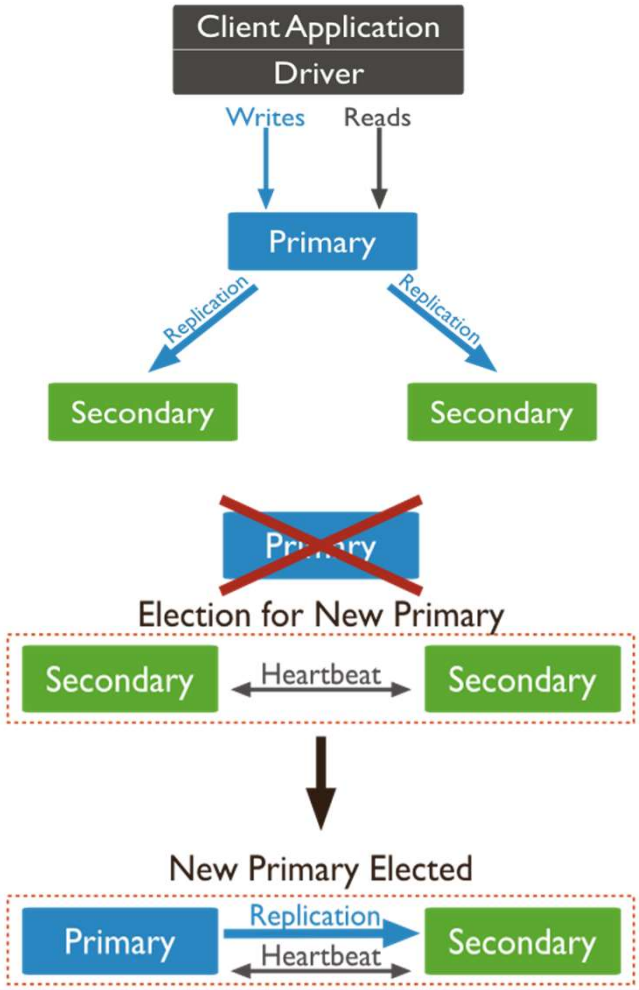
```
> db.users.update (
                    { “_id” : ObjectId("4efa8d2b7d284dad101e4bc7") },
                    { “$inc” : { “age” : 1 } } );
```

- Atomic updates (one document)
  - $set : modify value
  - $unset : remove a key/value
  - $inc : Increment
  - $push : Add inside an array
  - $pushAll : Add several values
  - $pull : Remove a value in an array
  - $pullAll : Remove several values

UpdateAll : all documents are updated

Replica Set

- Data replication
  - Asynchronus
    - Primary server: writes
    - Secondary servers: reads
  - Fault tolerance
    - New primary server election
    - Needs an *arbiter*
- Consistency vs availability
  - Reads applied on the primary (by default)
  - Updates via oPlog (log file)



Sharding

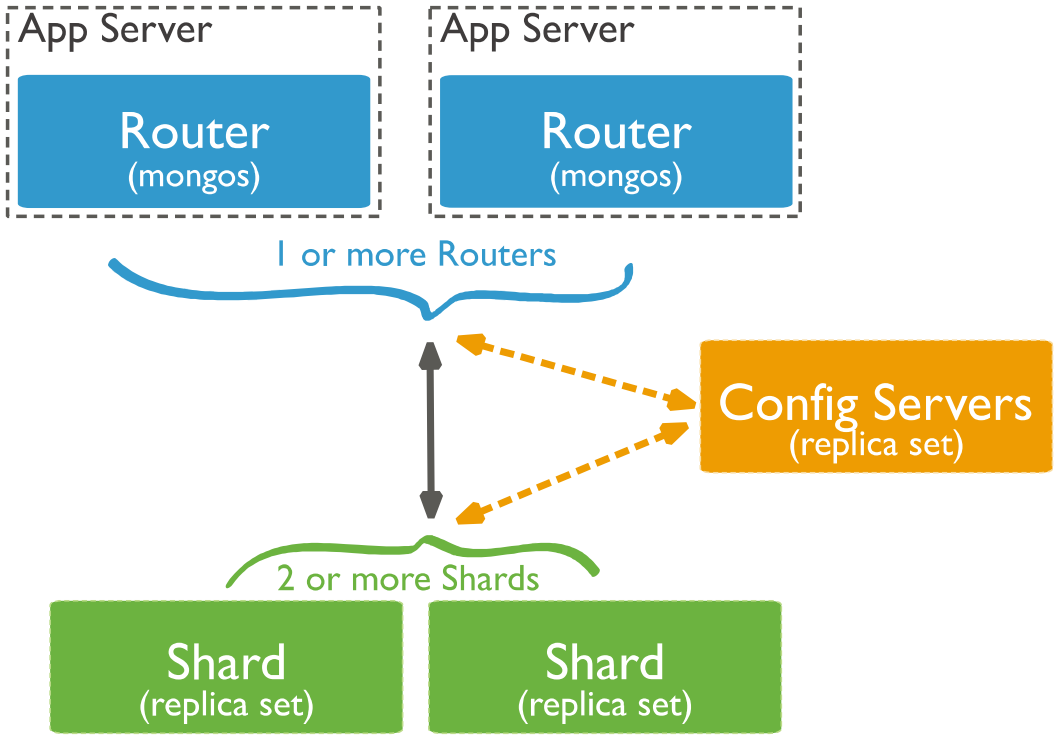
- Data distribution on a cluster
- Data balancing
  - Sharding key choice: beginning
- 1 Shard = 1 Replica Set
- Min config:
  - *Config Servers (x3)*
  - *Mongos* (router x3)
- Partitioning key (sharding)
  - Ranged-based (GridFS : sort)

```
> sh.shardCollection( "myBD.users", "login" );
```

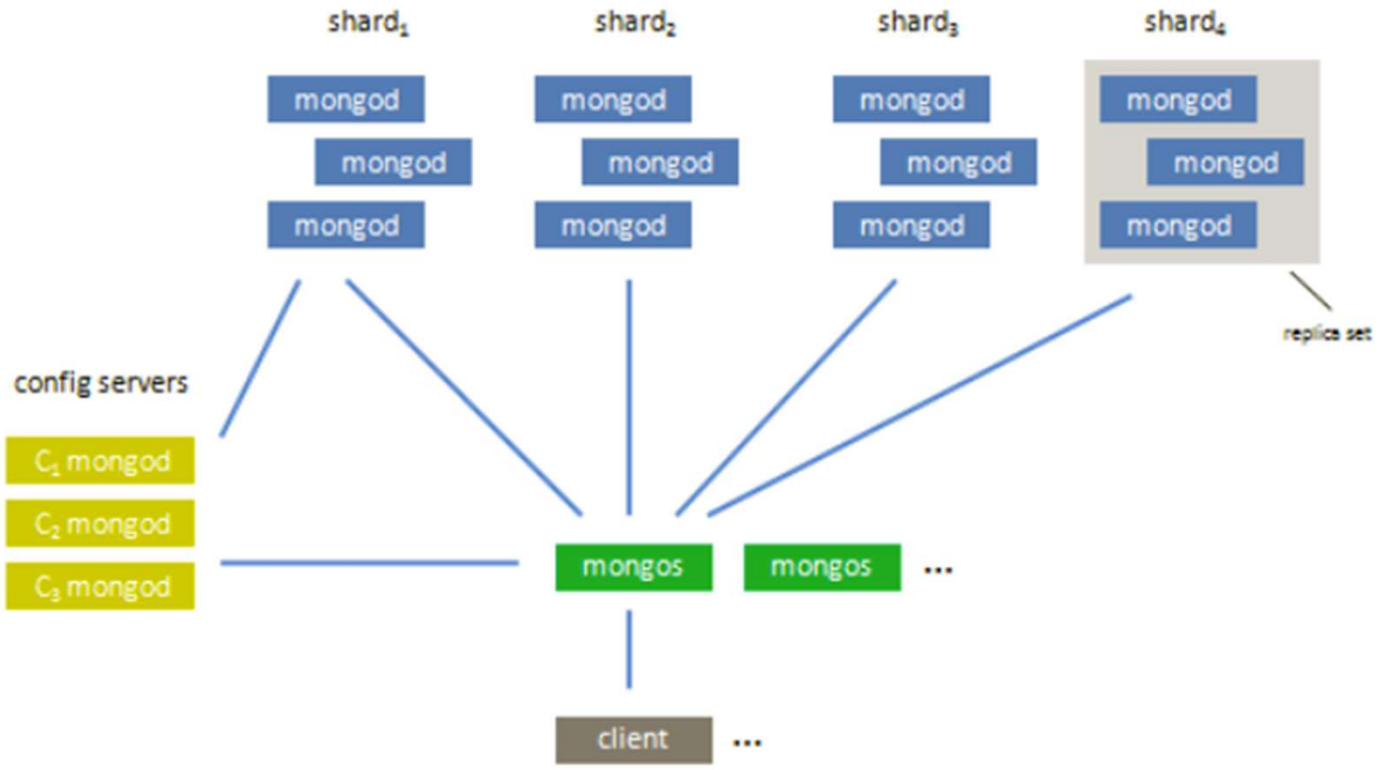
  - Hash-based (md5 sur clé)

```
> sh.shardCollection( "myBD.users", {"_id":"hashed"} )
```

  - By zones/tags (+ sharding)



Cloud Architecture



Indexing

Create a BTree

```
> db.users.createIndex( {"age":1} ) ;
```

- All queries on « age » become more efficient
  - Even for Map/Reduce with « queryParam »
  - Execution plan « .explain() »
- No combination of indexes

Indexing: 2DSphere

Apply geolocalized queries

```
> db.users.ensureIndex( { "address.location" : "2dsphere" } );
```

Localization format

```
"location":{"type": "Point", "coordinates" : [51.489220, -0.162866]}
```

Spherical queries

```
var near = {$near: {$geometry:{"type":"Point","coordinates":[51.489220,-0.162866]},
                                        $maxDistance : 10000}};
db.users.find( {"address.location":near}, {"name":1, "_id":0});
```

$geoNear operator (*aggregate*)

```
var geoNear = {"near":{"type":"Point", "coordinates" : [51.489220,-0.162866]},
            "maxDistance":10000, "distanceField" : "outputDistance", "spherical":true};
db.users.aggregate([ {"$geoNear": geoNear} ]);
```

« polygonal » queries

```
var polygon = {$geoWithin : {$geometry : {"type" : "Polygon", "coordinates" : [ [P1], [P2], [P3] }}}
```

http://docs.mongodb.org/manual/tutorial/query-a-2d-index/

Map/Reduce

Language : Javascript

```
var mapFunction = function () {
        if( this.age > 30 && this.job.contains("MI6") )
            emit (this.address.city, this.age);
} //emit : key, value,      this > current document

var reduceFunction = function (key, values) {
        return Array.sum(values);
} //returns a unique value aggregated from the list of "values" (for a given key)

var queryParam = {query : {}, out : "result_set"}
        //query: filter before applying the map, $match
        //out: output collection

db.users.mapReduce (mapFunction, reduceFunction, queryParam);
db.result_set.find();
```

Map/Reduce

Map

- Several « *emit* » per map (key/value)
- Applied on every documents
  - Unless: index + queryParam

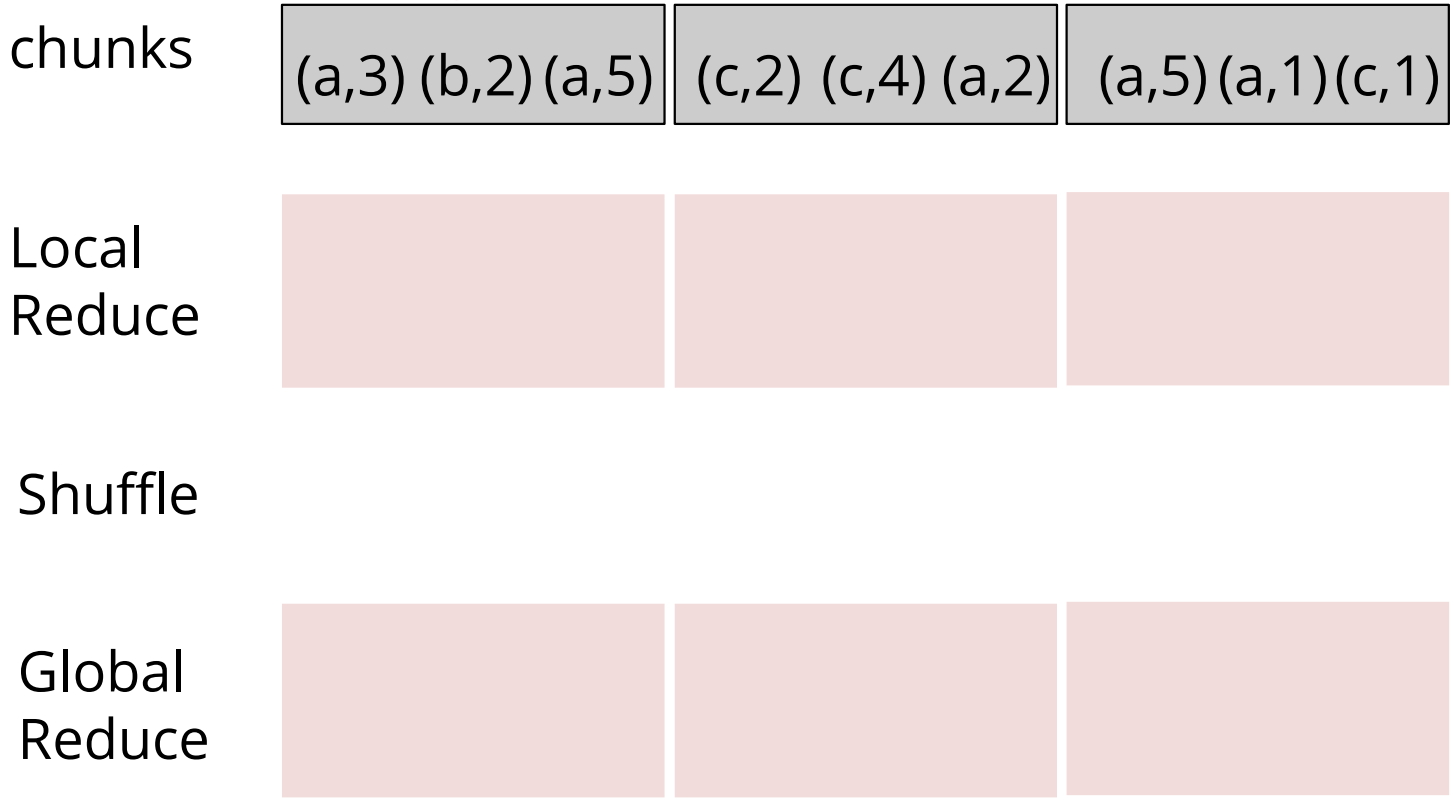
Reduce

- Optimization : local and global
- No reduce if only one « value »
- ➢Heterogeneous results/computations
  - •Beware of non associative functions (averages, lists, etc.)

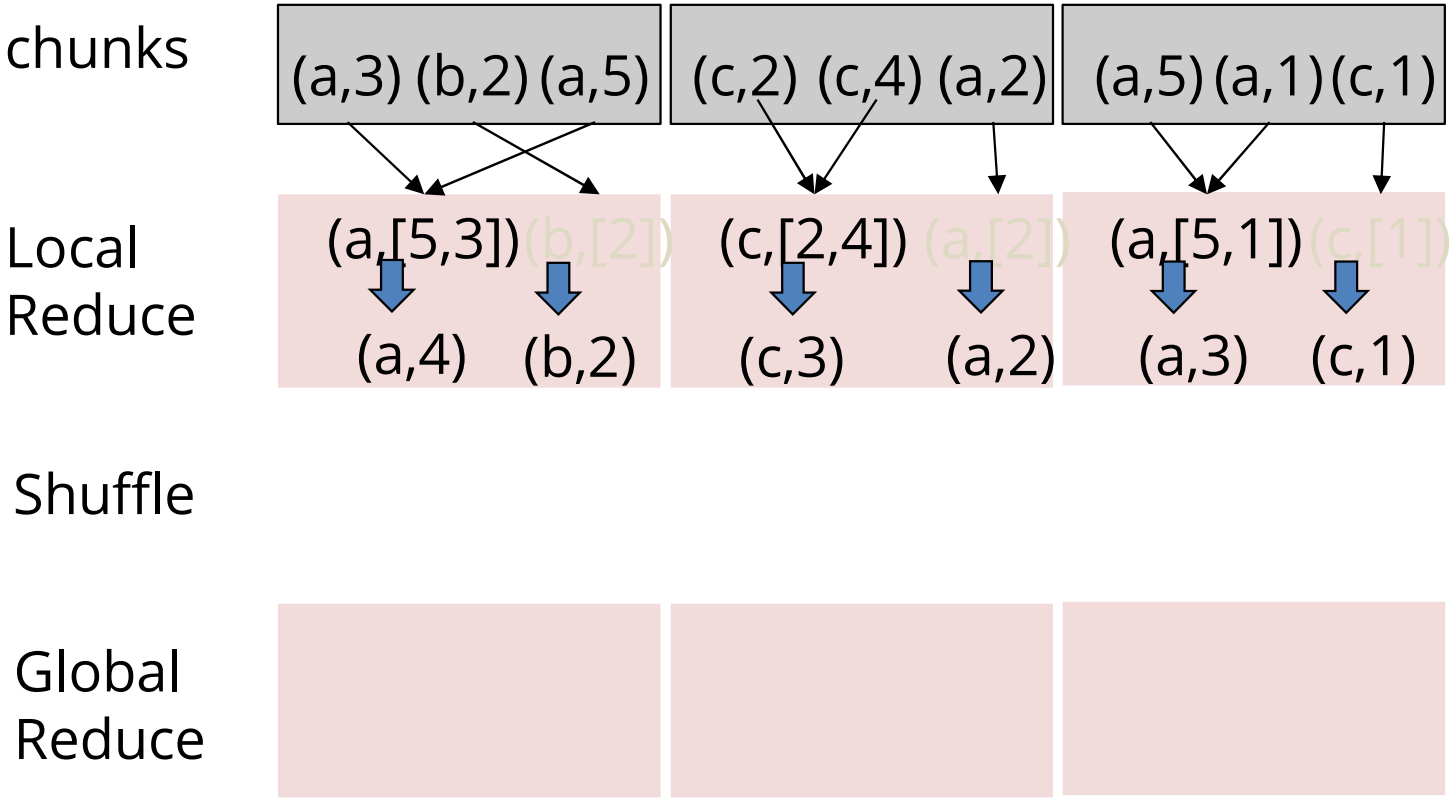
Shuffle

- Group values by keys (from « emit »)
- Optimize on sharding key (by default _id)

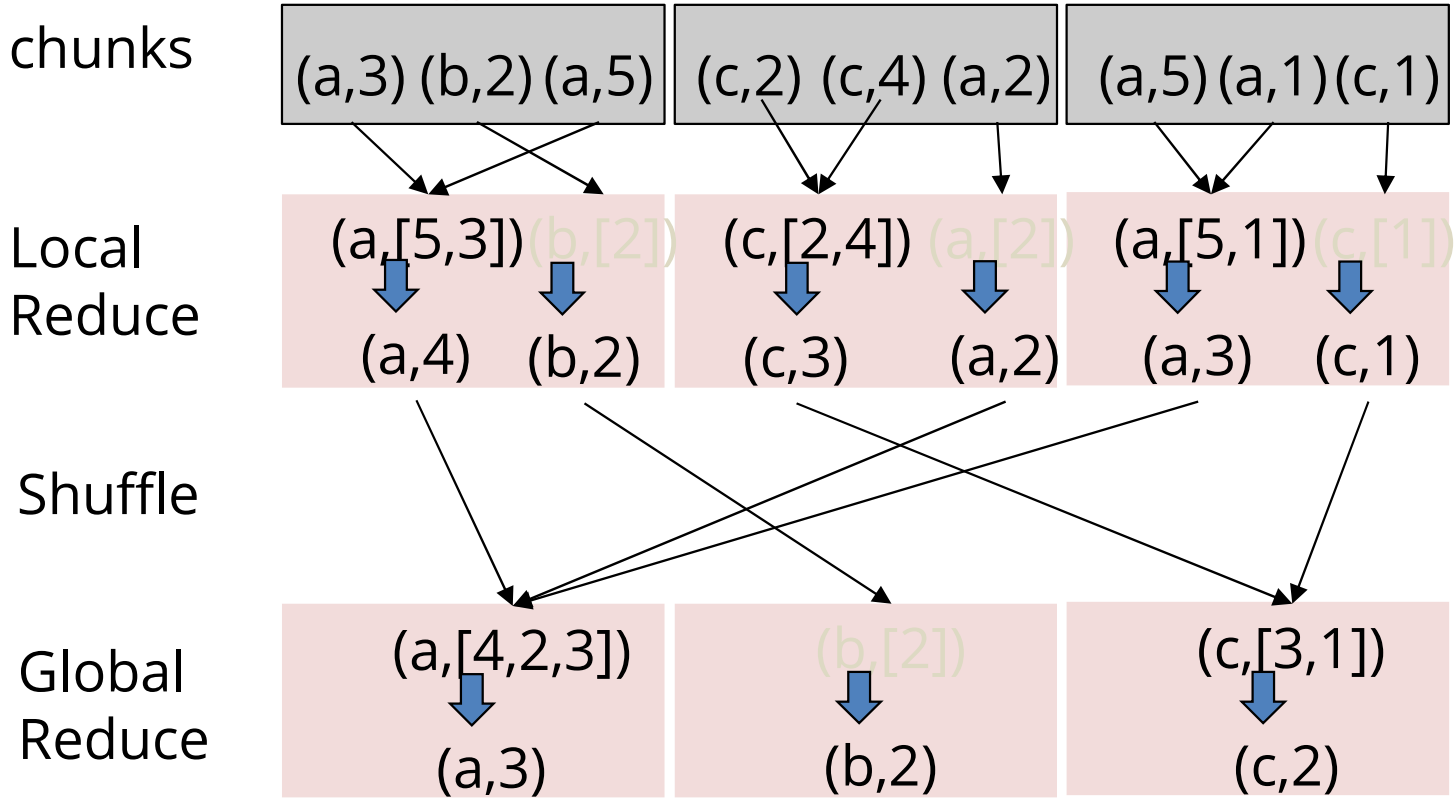
Reduce: Average function



Reduce: Average function



Reduce: Average function



Drivers / API

- In order to connect applications to MongoDB
- Drivers: http://docs.mongodb.org/ecosystem/drivers/
  - Python, Ruby, Java, Javascript (Node.js), C++, C#, PHP, Perl, Scala…
  - Syntax: http://docs.mongodb.org/ecosystem/drivers/syntax-table/

Driver Java

Connection

```
ServerAddress server = new ServerAddress ("localhost", 27017);
Mongo mongo = new Mongo(server);
```

Environment (DB + collection)

```
DB db = mongoClient.getDB( "maBD" );
DBCollection users = db.getCollection("users");
```

Authentification

```
db.authenticate(login, passwd.toCharArray());
```

Document inserting (DBObject)

```
DBObject doc = new BasicDBObject("name", "MongoDB")
.append("type", "database")
.append("count", 1)
.append("info", new BasicDBObject("x",
                203).append("y", 102));
// or doc = (DBObject) JSON.parse(jsonText) ;
users.insert(doc);
```

Querying (pattern query + cursor)

```
BasicDBObject query = new
            BasicDBObject("name", "James Bond");
DBCursor cursor = coll.find(query);
try {
  while(cursor.hasNext()) {
        System.out.println(cursor.next());
    }
} finally {
    cursor.close();
}
```

Map/Reduce :

```
MapReduceCommand cmd =
            new MapReduceCommand("users", map,
                        reduce, "outputColl",
            MapReduceCommand.OutputType.REPLACE, query);
MapReduceOutput out = users.mapReduce(cmd);
for (DBObject o : out.results()) {
            System.out.println(o.toString());
}
//outputType : INLINE, REPLACE, MERGE, REDUCE
```

http://api.mongodb.org/java/current/index.html?_ga=1.177444515.761372013.1398850293

Driver C#

References/Libraries

MongoDB.Bson.dll

MongoDB.Driver.dll

Connection

```
var connectionString = "mongodb://localhost";
var client = new MongoClient(connectionString);
var server = client.GetServer();
```

Database & Collection

```
var db = server.GetDatabase("maBD");
var coll = db.GetCollection<User>("users");
```

Objet « User » to define

```
public class User{
   public ObjectId Id { get; set; }
   public string name { get; set; }
   public string login { get; set; }
   public int age { get; set; }
   public Address address { get; set; }
}
```

```
using MongoDB.Bson;
using MongoDB.Driver;
```

Get a document :

```
var query=Query<User>.EQ(e=>e.login,"james");
var entity = coll.FindOne(query);
```

Package for output results: LINQ

```
using MongoDB.Driver.Linq;
```

Queries:

```
var query = coll.AsQueryable<User>()
                .Where(e => e.age > 40)
                .OrderBy(c => c.name);
```

Answer:

```
foreach (var user in query)
{
   // traitement
}
```

MapReduce :

```
var mr = coll.MapReduce(map, reduce);
foreach (var document in mr.GetResults()) {
   Console.WriteLine(document.ToJson());
}
```

http://www.nuget.org/packages/mongocsharpdriver/

Driver Python

Importation

```
>>> import pymongo
```

Connexion

```
>>> from pymongo import MongoClient
>>> client = MongoClient()
>>> client = MongoClient('localhost', 27017)
```

Database & collection

```
>>> db = client.maBD
>>> coll = db.users
```

GET one document

```
coll.find_one ( { "login" : "james"} )
```

Query

```
>>> for c in coll.find ( {"age": 40} ) :
…              pprint.pprint (c)
```

MapReduce

```
>>> from bson.code import Code
>>> map = Code("function () {"
... " this.tags.forEach(function(z) {"
... " emit(z, 1);"
... " });"
... "}")
>>> reduce = Code("function (key, values) {"
... " var total = 0;"
... " for (var i = 0; i < values.length; i++) {"
... " total += values[i];}"
... " return total; } ")
>>> result = coll.map_reduce(map, reduce,
                  "myresults")
>>> for doc in myresults.find():
...       print doc
```

elasticsearch

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Introduction

- elasticsearch
  - NoSQL Search engine
  - Document-oriented NoSQL
  - JSON documents
  - Implemented in Java
  - Relies on the full-text search library:

    Lucene
  - Full-text indexing
  - Complex search queries on text

Applications with Elasticsearch

Companies:

- Uber
- Instacart
- Stack Overflow
- Shopify
- Udemy
- Expedia
- …

Integrated in:

- Datadog
- Couchbase
- Amazon
- Jaeger
- …

Evolutions

- V1.0 (2014)
  - Query/Get/Update APIs
- V2.0 (2015)
  - Custom config file, packaging, plugins
- V5.0 (2016)
  - Cluster enhancement, core evolutions, optimizations, mapping corrections
- V6.0 (2017)
  - Changes: mapping types, aggregations, cluster, indices, Java API, packaging, REST, Query DSL, scripting…
- V6.6 (2019)
  - Frozen indices, Index Lifecycle, BKD-backed Geoshapes

ELK Stack

- **Elasticsearch**
  - NoSQL search engine
- **Logstash**
  - Data collection pipeline tool
- **Kibana**
  - Data visualization tool

ELK Stack

medium.com

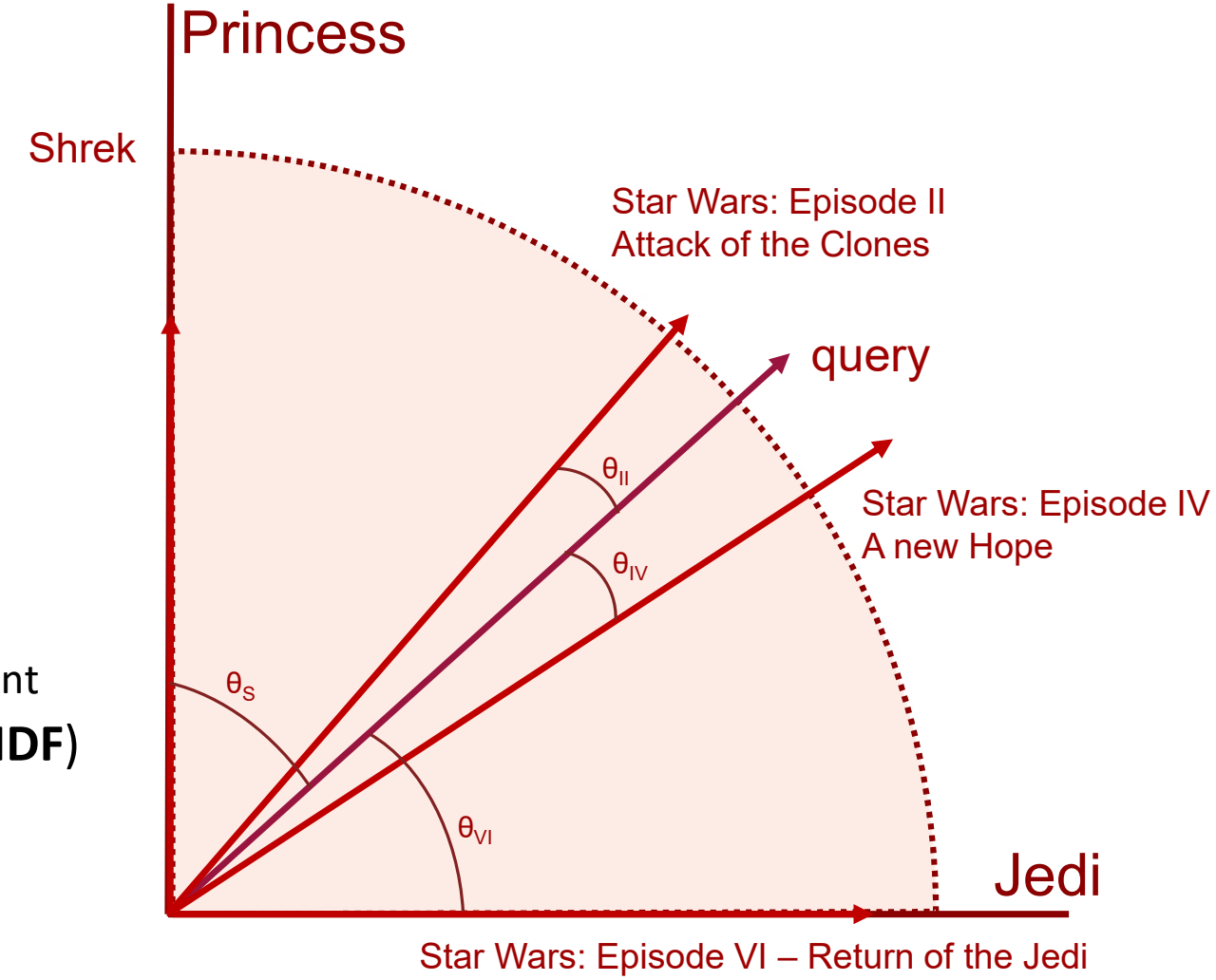
Elasticsearch RESTful API

- **cURL**[1] (executable for HTTP requests)
- Import data
  - **curl -XPUT localhost:9200/_bulk –H"Content-Type: application/json" --data-binary @movies_elastic.json**
  - Dataset:
    - **Each** JSON document must be **prefixed** by a **header**
      - {"index": {"_index" : "INDEXNAME", "_id": X}}
      - Index = collection (table)
    - Documents must not contain an "id" key
- GET
  - Standard query: curl -XGET 'http://localhost:9200/INDEXNAME/_search?q=some+words'
  - Smart query (DSL[2]): curl -H"Content-Type: application/json" -XGET 'http://localhost:9200/INDEXNAME/_search' -d @queryFile
  - RESTful Integrated in Kibana (Dev Tools)
  - Suppose the index is "movies"

1 – https://curl.haxx.se/download.html
2 – DSL: Domain Specific Language

Lucene

- Search Engine
  - Similarity between
    - The query: *q*
    - A textual document: *d*
  - Relevance score[1]: cos (q,d)
- The cosine relies on
  - Term Frequency (**TF**)
    - Normalized per key or per document
  - Inverse Document Frequency (**IDF**)



1 – ranking : http://b3d.bdpedia.fr/ranking.html

DSL – Simple Queries

- Standard queries
  - Whole document: http://localhost:9200/movies/_search?q=**Star+Wars**
  - Within a key: http://localhost:9200/movies/_search?q=**title:**Star+Wars
  - Two keys: http://localhost:9200/movies/_search?q=**title:**Star+Wars AND **actors**:Harrison
- DSL
  - Document query: { "**query**": { "**match**": { "*title*":"Star Wars" }}}
  - Boolean queries:
    - should
    - must/must_not
    - match_phrase
    - Range queries

```
{"query":{ "bool": { "should": [
                { "match": { "title": "Star Wars" }},
                { "match": { "actors": "Harrison" }} ] }}}
```

```
{"query":{ "bool": {
                "should": { "match": { "title": "Star Wars" }},
                "must": { "match": { "actors": "Harrison" }} }}}
```

```
{"query":{"match_phrase": {"title": "Star Wars" }}}
```

```
{"query": { "bool": { "must": { "range": { "rank": {"lt":1000 }}} }}}
{"query": { "bool": { "must": { "range": { "date" : {"from": "2010-01-01", "to": "2015-12-31"}}}}}}
```

DSL – Complex Queries

- Aggregate queries:
  - Simple group

```
{"aggs": { "produced_key": { "terms": { "field": "year"}}}}
{"aggs": { "produced_key": { "terms": { "field": "actors"}}}}
```

  - Group by range

```
{"aggs": { "produced_key": { "range" : {
        "field": "year", "ranges":[ {"from":2000, "to":2021 }] }}}}}
```

  - Number of distinct values

```
{"aggs": { "produced_key": { "cardinality": { "field": "actors.keyword"} }}}
```

  - Averages/Min/Max

```
{"aggs": { "produced_key": { "avg": { "field": "rating"} }}}
```

  - Composition *(maybe "hard" queries)*

```
{"aggs": { "produced_key": { "terms": { "field": "year"}, "aggs":{"avg" : "rating" } }}}}
```

DSL – Hard Queries with Mapping[1]

- In order to group keyword values
  - Query:

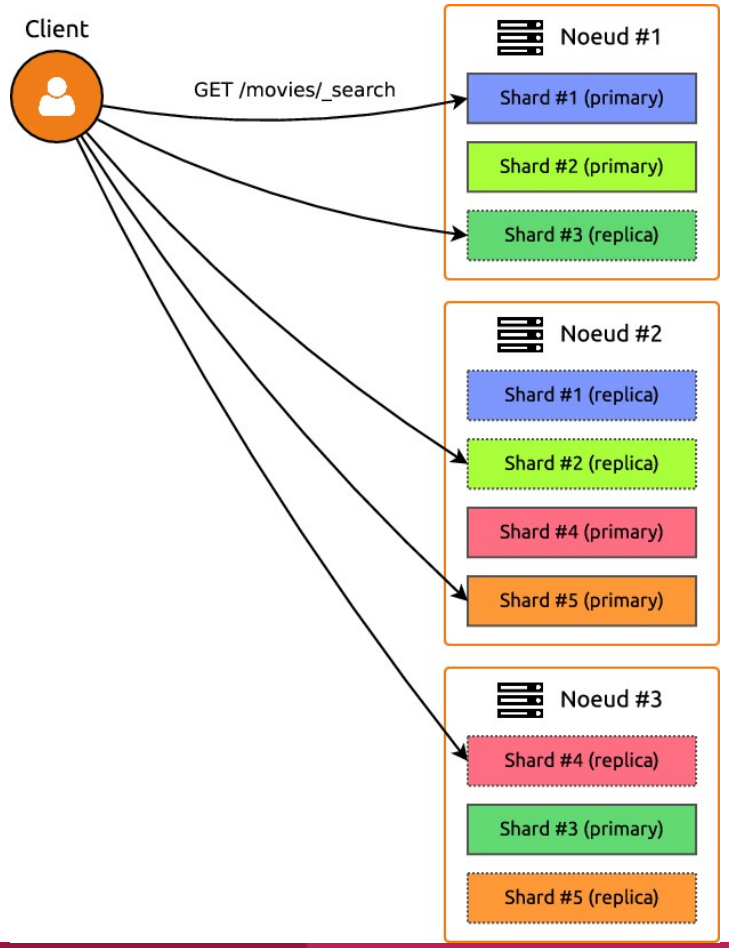
```
{ "query" :{"match" : {"title" : "Star Wars"}},
  "aggs" : { "top_keywords" : { "significant_terms" : {"field" : "plot"}}}}
```

  - Top keywords extraction
  - !!! This can consume a **lot** of memory
  - Need to map keys with type "fielddata"

```
PUT /movies/movie/_mappings
        { "properties": { "plot": { "type": "text", "fielddata": true } } }
```

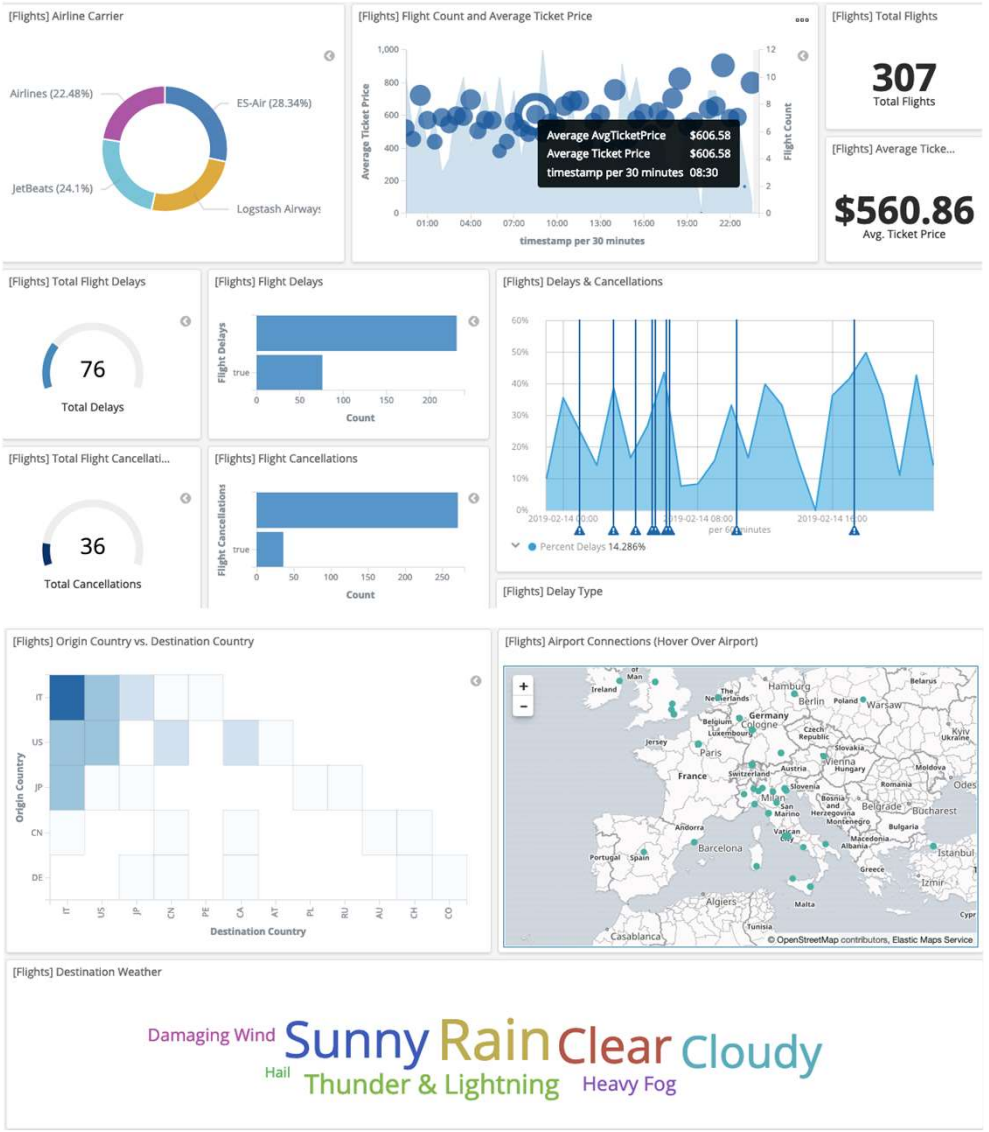
« Sharding » : Distribution & Replication

- Cluster
  - Must be **set** at the beginning of the index
  - **Static** hash function
  - Split the index in X fragments
  - Replicated on 1 or more nodes



Kibana

- Dashboard to visualize
  - Differente chart types
- Intuitive interface
- Can handle:
  - Time-series
  - Geo-shapes (maps)
  - IP/Images/Dates
  - Tag Clouds

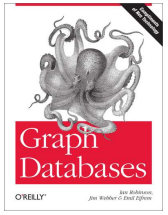


jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Introduction

- NoSQL Graph-oriented database
- Based on **Labeled Property Graph**
- CSV/JSON documents
- Implemented in Java
- DSL: **Cypher**

- Some readings:
  - Graph databases (O'Reilly ed.)

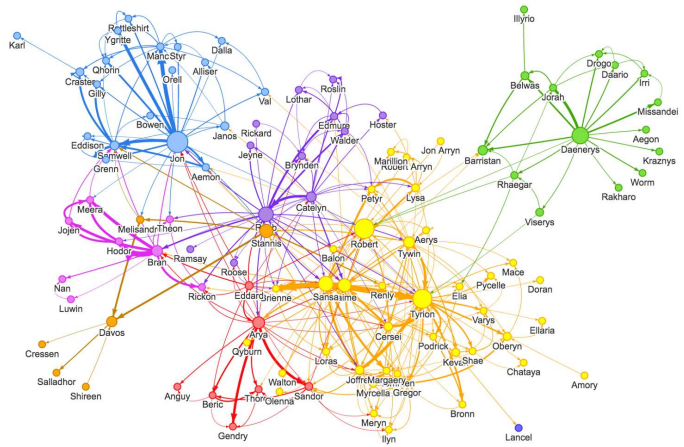


jihane.mali@devinci.fr
nicolas.travers@devinci.fr

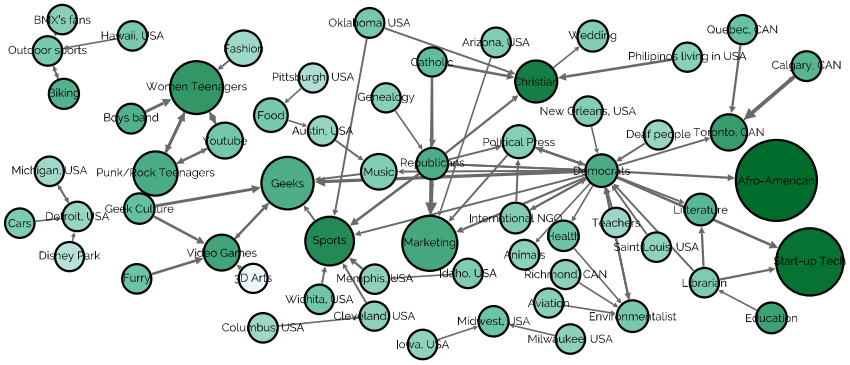
Applications using Neo4j

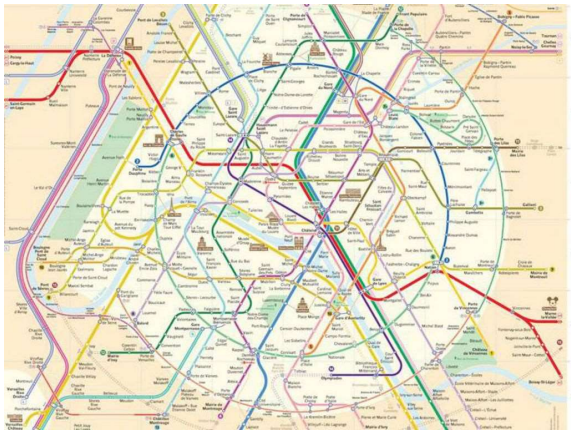
jihane.mali@devinci.fr
nicolas.travers@devinci.fr

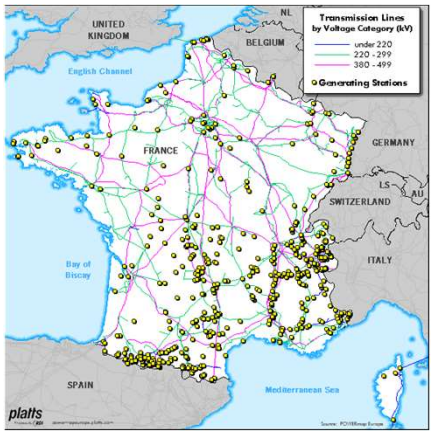
Case Studies



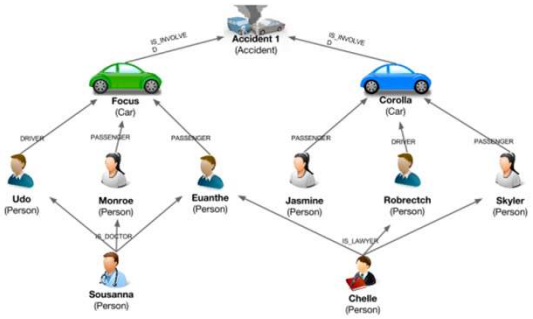
Social networks (analysis/reco)







Geo networks (recommandations)



Typed network (Fraud detection)

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Neo4j – A graph database

- Storage management
  - Typing
  - Structures
  - Cache Management
- Query on graphs
  - Query language
  - Data manipulation
- Query optimization
  - Indexes
  - Execution plans
- Procedures & Functions

neostore.**nodestore**.db
neostore.**relationshipstore**.db
neostore.**propertystore**.db
neostore.propertystore.index
neostore.propertystore.strings
neostore.propertystore.arrays

9 bytes : *max 2^{35} nodes*
1b Flag
+ 4b for 1st ID rel
+ 4b for 1st ID prop

33 bytes : *max 2^{35} relationship*
1b Flag
+ 8b for ID start & end node
+ 4b for type pointer
+ 16b for relationship chain
+ 4b for 1st ID prop

max 2^{38} properties

```
MATCH (a:Actor) –[:acted_in]-> (m:Movie)
WHERE m.title = "Star Wars
RETURN a.name
```

Schema indexing (LOOKUP)
Properties indexing (RANGE)
Full-text indexing (TEXT)
Spatial indexing (POINT)

```
CREATE XXX INDEX FOR (n:Person) ON (n.name);
```

https://neo4j.com/developer/kb/fulltext-search-in-neo4j/

https://neo4j-contrib.github.io/spatial/0.24-neo4j-3.1/index.html

JAVA programming (extensible)
CALL dbms.procedures()
Libraries : apoc, load, security, db, graph algorithms

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Labeled-Property Graph Data Model

FROM RDBMS TO GDBMS

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Relational Data Model

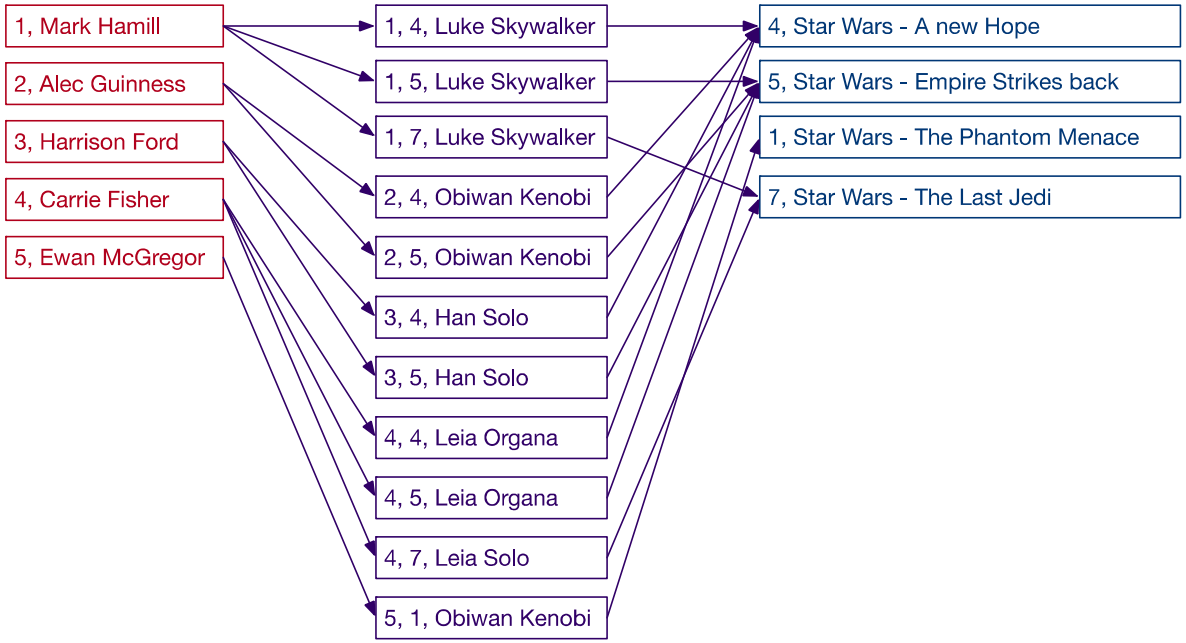
Person
1, Mark Hamill
2, Alec Guinness
3, Harrison Ford
4, Carrie Fisher
5, Ewan McGregor

acted_in
1, 4, Luke Skywalker
1, 5, Luke Skywalker
1, 7, Luke Skywalker
2, 4, Obiwan Kenobi
2, 5, Obiwan Kenobi
3, 4, Han Solo
3, 5, Han Solo
4, 4, Leia Organa
4, 5, Leia Organa
4, 7, Leia Solo
5, 1, Obiwan Kenobi

Movie
4, Star Wars - A new Hope
5, Star Wars - Empire Strikes back
1, Star Wars - The Phantom Menace
7, Star Wars - The Last Jedi

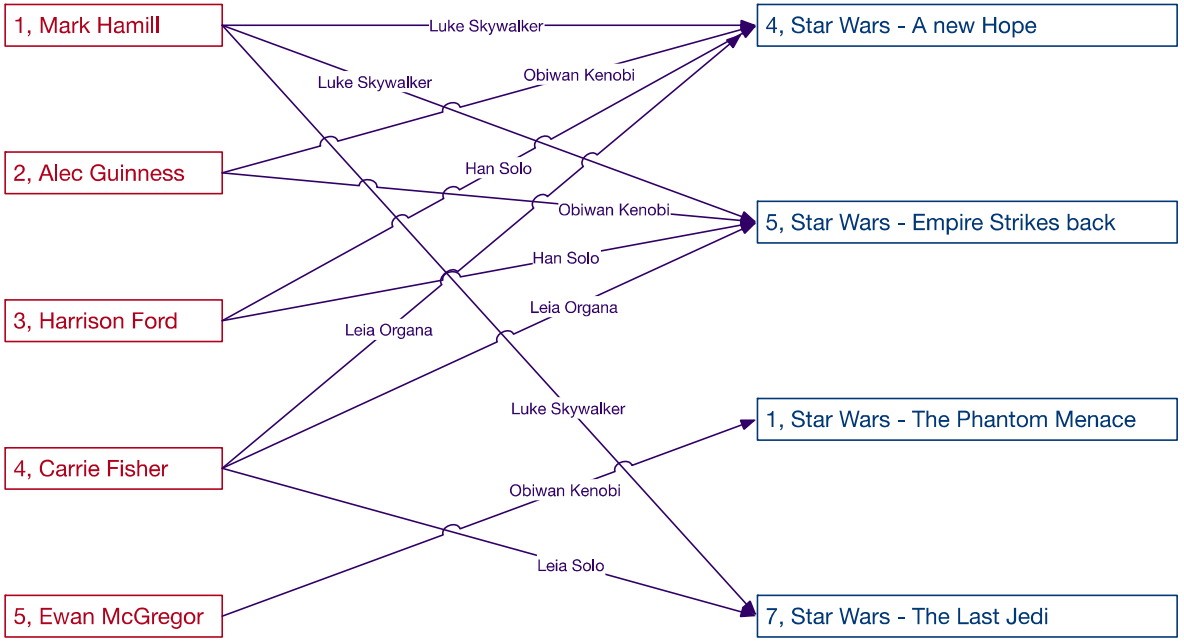
jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Direct Graph Data Model



jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Derived Graph Data Model



jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Graph Data Modeling (work in progress)

- Renzo Angles and Claudio Gutierrez. (2008). ***Survey of graph database models.*** ACM Comput. Survey vol 40(1), 39 pages. https://doi.org/10.1145/1322432.1322433

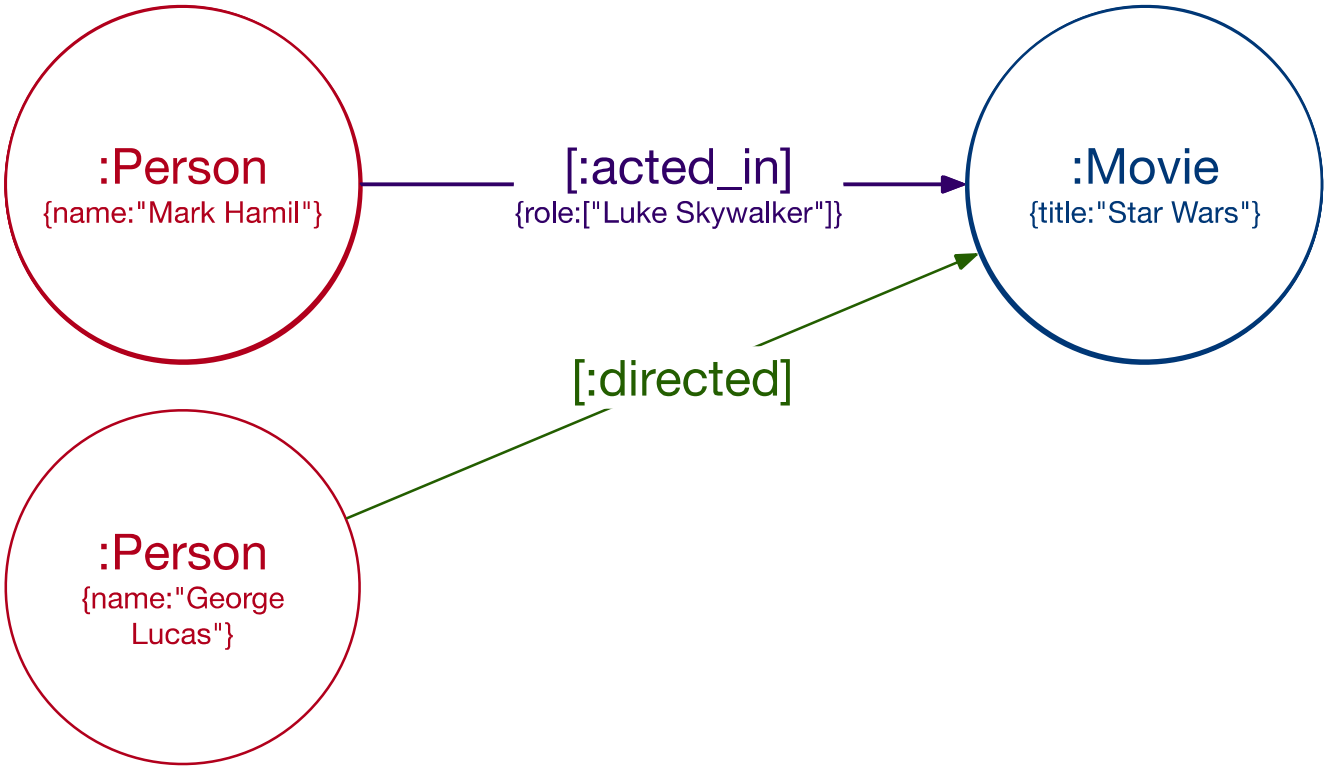
- Daniel, G., Sunyé, G., Cabot, J. (2016). ***UMLtoGraphDB: Mapping conceptual schemas to graph databases.*** ER'16, vol. 9974, pp. 430–444. https://doi.org/10.1007/978-3-319-46397-1_33

- Akoka, J., Comyn-Wattiau, I., du Mouza, C., Prat, N. (2021). ***Mapping Multidimensional Schemas to Property Graph Models.*** In: Reinhartz-Berger, I., Sadiq, S. (eds) Advances in Conceptual Modeling. ER'21, vol 13012. https://doi.org/10.1007/978-3-030-88358-4_1

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Neo4j Graph Data Model

- **Nodes**
  - Labels[1]
  - Properties (typed)
- **Relations**
  - Between 2 nodes
  - Oriented
  - Labels
  - Properties (typed)



1 – case sensitive

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Neo4j Indexes

Before creating the graph!

- Build index to optimize Cypher queries
  - **CREATE** TEXT **INDEX FOR** (a:Person) **ON** (a.name);
  - Can be made on a pattern (FOR <pattern> ON), Node & Relationship
- Range Index : Range values. Single or composite index.
- Text Index : String values
- Point Index : Point values (lat/long)
- Full-Text index : Lucene index. Search engine like
- Lookup Index on labels
  - **CREATE** LOOKUP **INDEX FOR** (n) **ON** *labels*(n);
  - **CREATE** LOOKUP **INDEX FOR** ()-[r]-() **ON** *types*(r);

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Build your own graph – Nodes

- **CREATE** (p:Person{ID:1})
  - Create a new node
- **MERGE** (p:Person{ID:1})
  - Create a new node if it does not exists
  - Or get the existing node
- **MATCH** (p:Person{ID:1}) **SET** p.name="Mark Hamill"
  - Add/Modify a property
- **MERGE** (p:Person{ID:1})
  ON CREATE **SET** p.updates = 1 ***//when created***
  ON MATCH **SET** p.updates = p.updates + 1 ***//when matched***

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Build your own graph – Relationships

- **MATCH** (dr:Person{name:"Daisy Ridley"}), (sw9:Movie{title:"The Rise of Skywalker"})
  MERGE (dr) **-[r:acted_in]->** (sw9)
  ON CREATE r.roles = ["Rey"]
  ON MATCH r.roles = r.roles + "Rey"

- **MATCH** (dr:Person{name:"Daisy Ridley"})
  DELETE dr
  - Applied only if « dr » is not connected to any node

- **MATCH** (dr:Person{name:"Daisy Ridley"})
  OPTIONAL MATCH (dr) -[r]-> ()
  DELETE dr, r

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Import datasets

```
LOAD CSV WITH HEADERS FROM "file:/actors.csv" as l
MERGE (a:Person{id:toInteger(l.IDA)})
MERGE (m:Movie{id:toInteger(l.IDM)})
SET a.name=l.name, m.title=l.title
MERGE (a) -[r:acted_in]-> (m)
ON CREATE SET r.roles = [l.role]
ON MATCH SET r.roles = r.roles + l.role
```

File « actors.csv » must be placed in $NEO4J_FOLDER/**import**

IDA	name	IDM	title	role
1	Mark Hamill	1	SW1	Luke
1	Mark Hamill	2	SW2	Luke
2	Carrie Fisher	1	SW1	Leia
3	Harrison Ford	1	SW1	Han

Typing values for range index

Create index before import!

```
CREATE INDEX FOR (p:Person) ON (p.id);
CREATE INDEX FOR (m:Movie) ON (m.id);
```

Import variations:

- V5 – "*CALL{LOAD CSV…};*"
- V4 – "*:auto LOAD CSV…;*"
- V3 – *"USING PERIODIC COMMIT 200 LOAD CSV…;"*

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

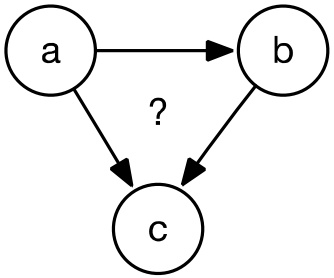
Pattern Matching Queries

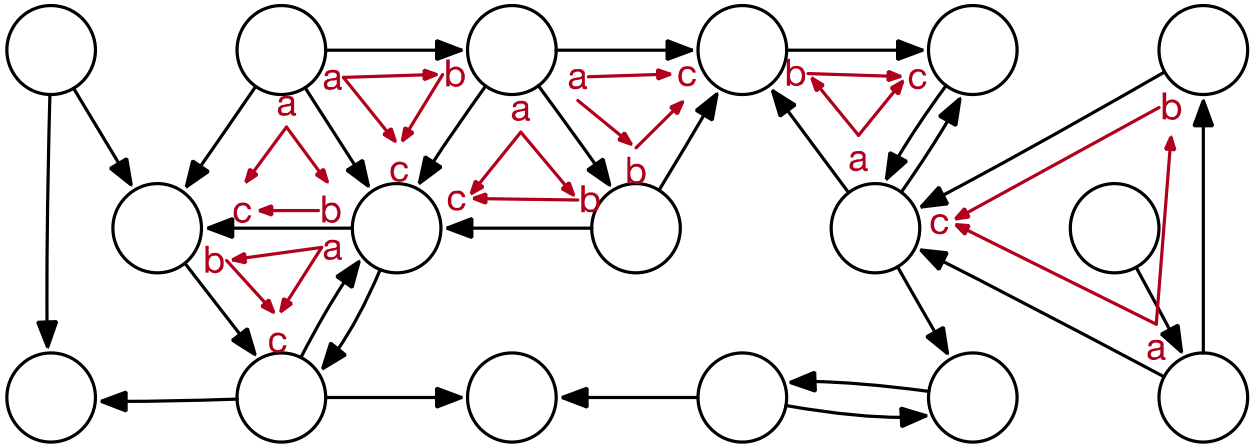
QUERYING GRAPHS WITH CYPHER

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – "SQL for graphs"

- Motivation
  - Pattern query





jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Pattern Matching

- **MATCH**
  - **()** node
  - (**:Person**) typed node
  - (:Person**{name:"Mark Hamill"}**) Direct filter on node properties
  - (**a**:Person) node reference "*a*" in the query (variable)
  - **-- --> <--** relationships
  - () --> () filtering pattern
  - -[**:acted_in**]-> typed relationship
  - -[**r**:acted_in]-> relationship reference "*r*" in the query (variable)
- **WHERE**
  - **a.name =** "Mark Hamill" Filter on properties
- **RETURN**
  - Projection of values
  - a.name properties
  - a nodes, relationships

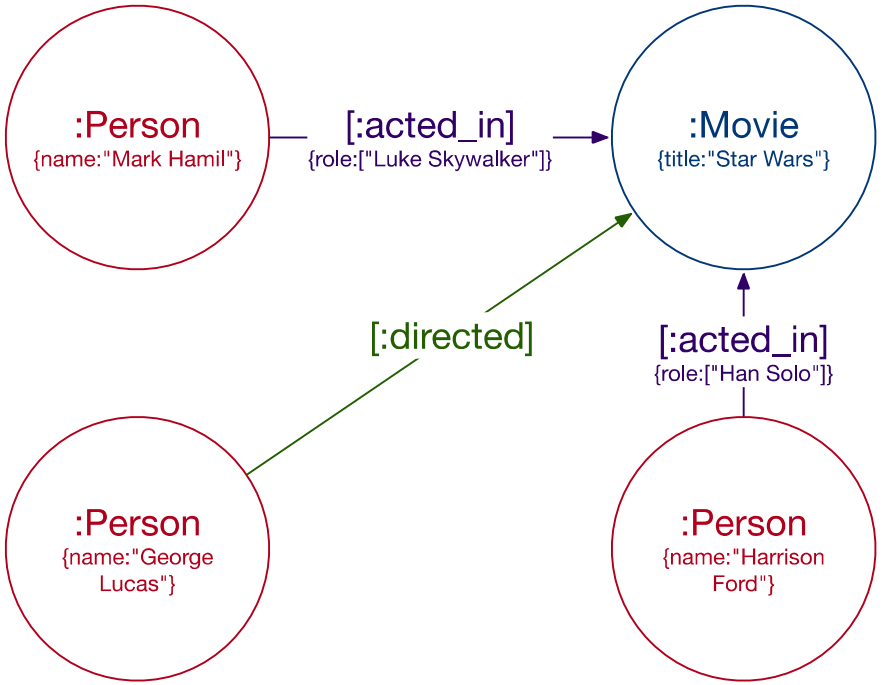
Output is "relational" no more a graph

- Other clauses : OPTIONAL MATCH, LIMIT, ORDER BY, MERGE, CREATE, ON CREATE, ON MATCH, SET, DELETE

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Simple queries

1. List of Persons linked to something
   - MATCH (p:Person) --> () RETURN p
2. List of Persons with their relationship to a Movie
   - MATCH (p:Person) -[r]-> (:Movie) RETURN p, r
3. List of Persons who acted in something
   - MATCH (p:Person) -[:acted_in]-> () RETURN p
4. List of Persons who acted in something which was directed by something
   - MATCH (p:Person) -[:acted_in]-> (m) <-[:directed]- (d)
     RETURN d.name, p.name, m.title
5. Titles of things linked to a Person who's name is Mark Hamill
   - MATCH (mh:Person{name:"Mark Hamill"}) --> (m) RETURN m.title
6. Titles of things in which a Person has acted in and who's role was Luke Skywalker
   - MATCH (:Person) -[r:acted_in]-> (m)
     WHERE "Luke Skywalker" IN r.role RETURN m.title
7. Titles and name of things who acted in a thing with a Person who's name is Mark Hamill (without himself)
   - MATCH (mh) -[:acted_in]-> (m) <-[:acted_in]- (p)
     WHERE mh.name = "Mark Hamill" and mh != p
     RETURN m.title AS title, p.name AS name



jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Pattern queries equivalence

- MATCH (p:Person) -[:acted_in]-> (m) <-[:directed]- (d)
  RETURN d.name, p.name, m.title

⇔

- MATCH (p:Person) -[:acted_in]-> (**m**), (**m**) <-[:directed]- (d)
  RETURN d.name, p.name, m.title

⇔

- MATCH (p:Person) -[:acted_in]-> (**m**), (d) -[:directed]-> (**m**)
  RETURN d.name, p.name, m.title

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Complex queries (1/2)

- **Group by** with aggregate functions
  - MATCH (a) –[:acted_in]-> (m) <-[:directed]- (d)
    RETURN a.name, d.name, COUNT(*) as NB

a.name	d.name	NB
Mark Hamill	George Lucas	1
Harrison Ford	George Lucas	2
Harrison Ford	Steven Spielberg	4

  - MATCH (a) –[:acted_in]-> (m) <-[:directed]- (d)
    RETURN a.name, d.name,
    COLLECT(m.title) as movies

a.name	d.name	movies
Mark Hamill	George Lucas	SW1
Harrison Ford	George Lucas	American Graffiti, SW1
Harrison Ford	Steven Spielberg	Indiana Jones 1, IJ2, IJ3, IJ4

Implicit "group by" on non-aggregate keys (here: a.name, d.name)

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Complex queries (2/2)

- Complex path chaining
  - Actors who played with Harrison Ford and also directed a movie

```
MATCH (hf:Person) -[:acted_in]-> (m:Movie), (ad) -[:acted_in]-> (m)
WHERE hf.name="Harrison Ford" AND (ad) -[:directed]-> ()
RETURN DISTINCT ad.name
```

  - Actors who played with Harrison Ford but not when Mark Hamill played

```
MATCH (hf:Person) -[:acted_in]-> (m:Movie), (ad) -[:acted_in]-> (m), (mh:Person)
WHERE hf.name="Harrison Ford" AND mh.name="Mark Hamill"
                AND (ad) -[:directed]-> ()
                AND NOT (mh) -[:acted_in]-> (m)
RETURN DISTINCT ad.name
```


jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Miscellaneous

- *UNWIND* – temporal "relational" step (normalizing).
- *type(r)* – give the node/relationship type
- *a.name CONTAINS 'Ford'*
  - STARTS WITH / ENDS WITH
  - ~regex
- *OPTIONAL MATCH (a) --> (r)*
  - The relationship is not mandatory
- *RETURN DISTINCT a.name*
  - Distinct values
- *ORDER BY a.name*

RefCard : https://neo4j.com/docs/cypher-refcard/current/

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Some training

- Produce the Cypher queries for those sentences
  - **Person who has directed the movie Star Wars**
  - **Title and name of persons who acted in the movies he directed**
  - **Actors name who played with Tom Hanks and older than him**
  - **Movies' title where Tom Hanks and Kevin Bacon played together**
  - **Idem with a common director for two movies**
  - **Top 5 couples of actors who played together**

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

ADVANCED GRAPH DATA MODELS AND PATTERN MATCHING QUERIES

ESILV

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

From bi-partite to mono-partite graphs

- **MATCH** (a1:Person) --> (m:Movie) <-- (a2:Person)
  MERGE (a1) **-[k:knows]-** (a2) //undirected relationship
  ON CREATE k.nb = 1
  ON MATCH k.nb = k.nb + 1;

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Hard queries (1/2)

- **Unlimited path** length (possible connections)

```
MATCH (mh:Person{name:"Mark Hamill"}) -[:knows*]- (p:Person)
WHERE NOT (mh) -[:knows]- (p)
RETURN p.name
```

- **Path length** from 2 to 3 – the most recommended Person in the neighborhood

```
MATCH (mh:Person{name:"Mark Hamill"}) -[k:knows*2..3]- (p:Person)
WHERE NOT (mh) -[:knows]- (p)
UNWIND p.name AS personName, k AS rels
RETURN personName, SUM(rels.NB) AS NB
ORDER BY NB DESC LIMIT 5
```

- **(All) Shortest path** between two nodes – distance between two Persons or

```
MATCH p=shortestpath( (mh:Person{name: "Mark Hamill"}) -[:knows*]-> (em:Person{name: "Ewan McGregor"}) )
RETURN size( relationships(p) )
```


jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Cypher – Hard queries (2/2)

- Graph manipulations:
  - GDS / Graph algorithms: paths (shortestpath, SpanningTree), centralities (PageRank, betweenness), communities (Louvain, Infomap), GNN (Node2Vec, GraphSAGE), etc.
  - APOC: graph refactoring/updates/data structures, temporal functions, math operations, comparing graphs
  - ETL: Relational to Graph (using JDBC), Kettle (build CSV files)
  - Neosemantics: XML/RDF, Ontology, Shapes Constraint Language (SHACL), Inferencing

- External libraries import
  - Modify config file: $Neo4j_folder/conf/neo4j.conf
  - Add the extension name: server.unmanaged_extension_classes=XXX.extension=/XXX
  - Call the lib: CALL XXX.*function_name*.(*params*)

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

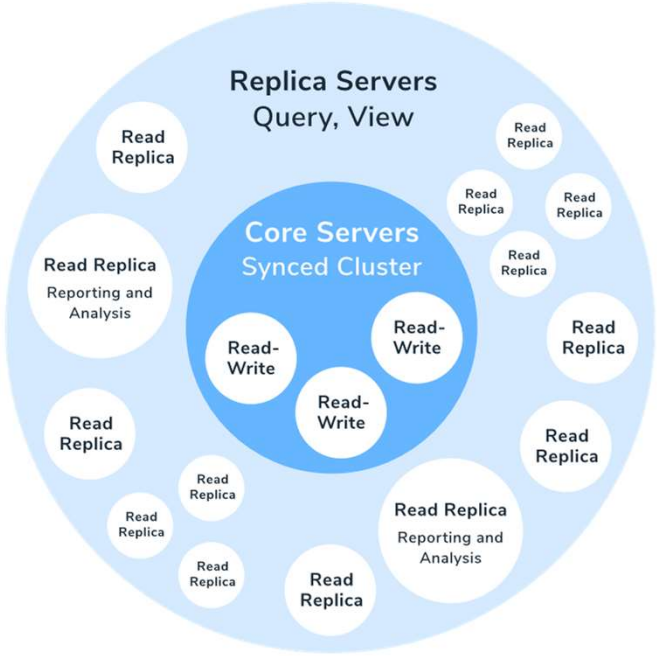
How to distribute a graph?

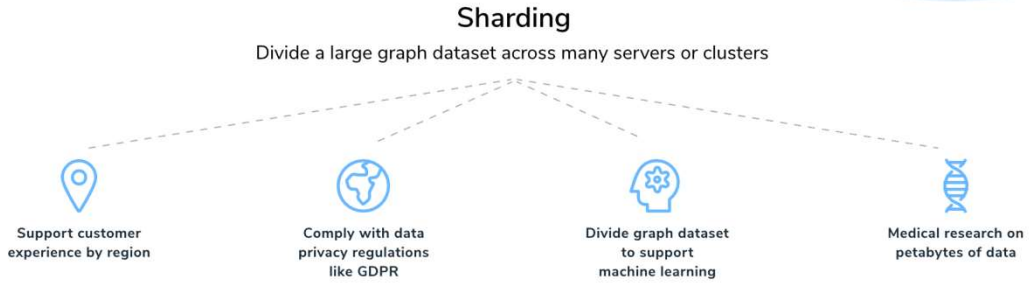
SHARDING WITH NEO4J

jihane.mali@devinci.fr
nicolas.travers@devinci.fr

Sharding: Distribution & Replication

- **Causal Clustering**
  - Causal cluster Architecture
    - Core servers
      - Causal Consistency
    - Replicates
- **Multi-clustering**
  - Multitenant database
    - Multiple dbs
    - Shared discovery service
  - Consensus consistency
    - Locally: causal clustering





https://neo4j.com/docs/operations-manual/current/clustering-advanced/multi-clustering/introduction/

Evolutions

- **V1.3 (2011)**
  - Store formats, Lucene Index, cache management, transactions (ACID)
- **V2.0 (2013)**
  - Node labels, Cypher enhancement, Neo4j Browser
- **V3.0 (2015)**
  - Bolt (network protocol for drivers), Cypher cost planner, spatial functions, variables
- **V3.5 (05-2018)**
  - Multi-clustering (partitioning graphs with flag/geo), Data-Type for Space and Time, Performances on indexes (nodes&rel, full-text, sorting), Security enhancement
- **V4.0 (01-2020)**
  - Sharding, Granular security, Multi-database, data pipeline
- **V5.0 (10-2022)**
  - Query optimization, indexes enhancement (patterns)
  - Automated clustering
- **V5.5 (02-2023)**
  - Packages enhancement (apoc, bloom, browser, GDS, Ops Manager)
  - Read and Export enhancement

jihane.mali@devinci.fr
nicolas.travers@devinci.fr