databases, data engineering & big data

NoSQL World

ESME SUDRIA

Luc Marchand - luc.marchand.pro@proton.me
Maxence Talon - maxencetallon@gmail.com

Global Syllabus

01 Introduction and main concepts

02 SQL, set up env and practical work

03 **NoSQL world**

04 Introduction to Big Data & Data Engineering

05 Kafka & event driven architectures

06 Spark & Delta

07 Warehouse, DBT & BI

08 IA - MLOps & RAG

Course syllabus

01 Why ? & What is NoSQL ?

02 Overview of the NoSQL data models

03 Choose an architecture

04 Focus on MongoDB

01

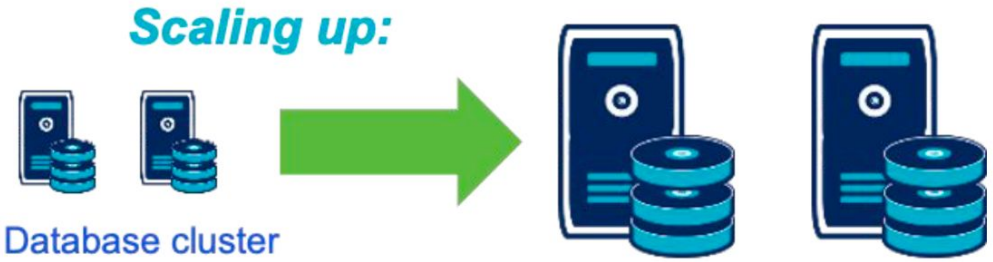
Why ? & What is NoSQL ?

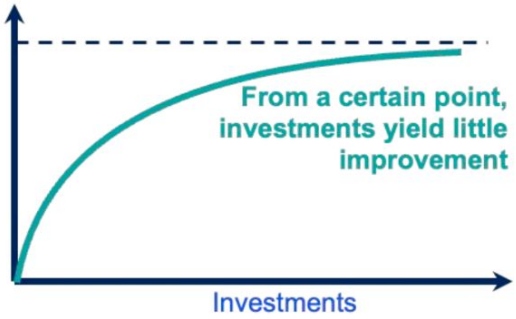
The origins

- The mid and late 2000's were times of major changes in the IT landscape
  - > Hardware capabilities significantly increased
  - > eCommerce and internet trade, in general, exploded

- Some internet companies, so-called the "Internet giants" (Yahoo!, Facebook, Google, Amazon, Ebay, Twitter, …), pushed traditional databases to their limits. Those databases are **by design hard to scale**

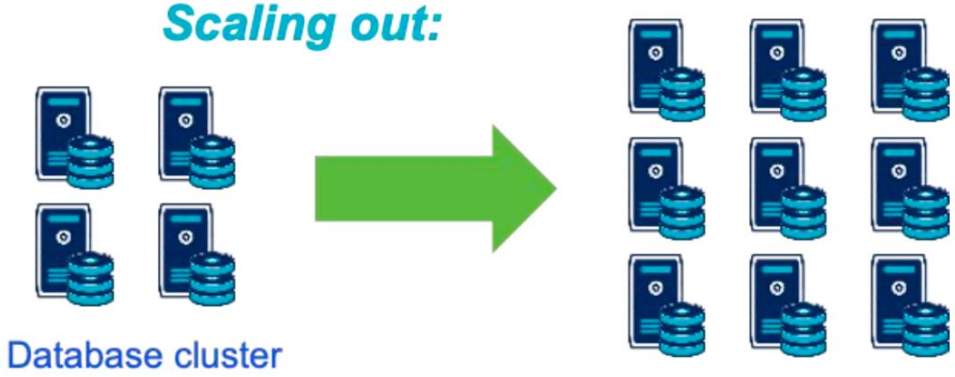
- With relational DBMSes, the only way to improve performance is by **scaling up**, i.e. getting bigger servers (more CPU, more RAM, more disk, …). One eventually hits a hard limit imposed by the current technology

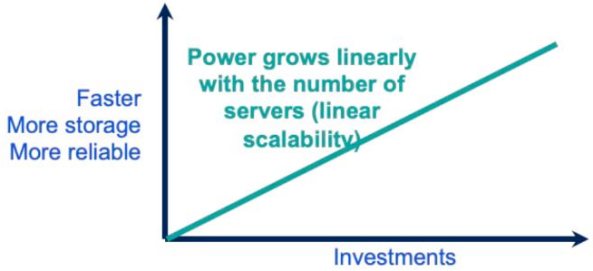




The origins (cont'd)

- By rethinking the architecture of databases, those companies were able to make them scale at will, by adding more servers to clusters instead of upgrading the servers.
  - The servers are not made of expensive, high-end hardware; they are qualified as **commodity servers** (or **commodity hardware**)



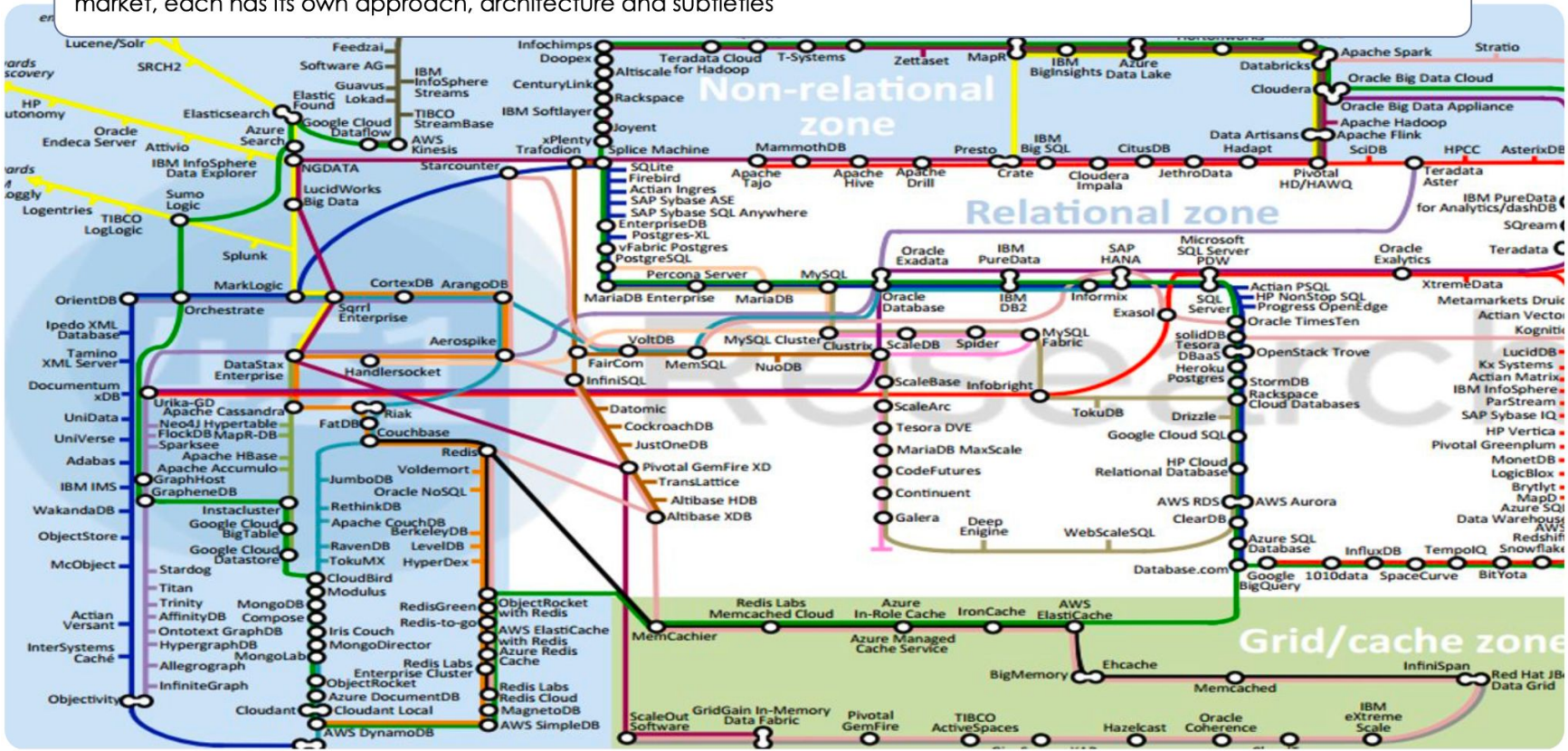


Foreword

- The NoSQL term itself has taken various meanings over the course of time
- It is thus impossible to give an exhaustive description of the NoSQL landscape. Besides, it would need delving into very technical details of the implementations
- The following slides are a description of the **most common patterns** found with some well-known NoSQL databases. Beware that some assumptions may not hold with other databases that are nonetheless considered NoSQL
- For example, Redis is a NoSQL key/value database but it's difficult to distribute
- Those assumptions can be seen as so-called **genes** shared by many databases, forming the **NoSQL DNA**:
  - semi-structured data
  - data distribution,
  - replication,
  - the trade-offs behind the CAP theorem,
  - Etc …
- Understanding those genes is more important than remembering implementation details

A new world, an eldorado

The most salient feature of the NoSQL ecosystem is its variety. There are many NoSQL DBMSes (mostly open source) on the market, each has its own approach, architecture and subtleties



Reminder

Structuration of data

Data comes in 3 flavors : structured, semi-structured and unstructured.

Structured

- The data conforms to a predetermined structure
- The structure is a strong invariant of the data model
- All records of the same entity type share the same format
- **Examples**: tables in the relational model, XML with DTD/XSD
- **Pros**
  - Easy processing since the structure is known in advance
  - Data quality is enforced
- **Cons**
  - Too restrictive for some types of data (emails, documents scanned as images, …)
  - When the structure changes, what about existing data still in the old format?

Unstructured

- The data has no structure; the only known type is 'object'
- Objects are considered opaque buckets of bytes, sometimes called BLOBs (Binary Large OBjects)
- **Examples**: raw files, data streams
- **Pros**
  - Can store anything including heterogeneous or changing data
- **Cons**
  - There is no metadata for the structure, so data cannot be interpreted without an accompanying program that knows how to read and write them

Semi-Structured

- A kind of hybrid between the first two flavors
- One still deals with unstructured objects, but they bear some meaningful metadata
  - Example metadata: tags (an object can have as many tags as necessary), relationships
- **Examples**: XML and JSON documents in general, objects from most NoSQL databases, documents in a search engine
- **Pros**
  - The metadata give meaning to objects, without the rigidity of a full structure (compare Gmail tags with Outlook folders ☺)
- **Cons**
  - Besides metadata, the contents of the objects themselves are unstructured so one still needs to know how to interpret them

DNA1 - NoSQL favors semi-structured data to store versatile documents

Dans la plupart des bases NoSQL, il est possible d'insérer des données dans une collection sans spécifier de schéma ou en spécifiant un schéma partiel.

C'est le cas pour des bases comme :

- mongodb
- cassandra
- elasticsearch

Les moteurs NoSQL privilégient l'inférence de schéma à lors de l'enregistrement plutôt que les schémas rigides. **Le schéma est toujours présent et il est utile d'auditer sa structure lors de la maintenance de ces bases pour vérifier la consistance de son modèle de données.**

SEMI-STRUCTURED

Exemple pour Cassandra

```
INSERT INTO cycling.cyclist_category JSON '{
  "category" : "GC",
  "points" : 780,
  "id" : "829aa84a-4bba-411f-a4fb-38167a987cda",
  "lastname" : "SUTHERLAND"
}';

INSERT INTO cycling.cyclist_category JSON '{
  "category" : "GC",
  "points" : 780,
  "id" : "829aa84a-4bba-411f-a4fb-38167a987cda",
  "lastname" : "SUTHERLAND"
  "ranking" : A+
}';
```

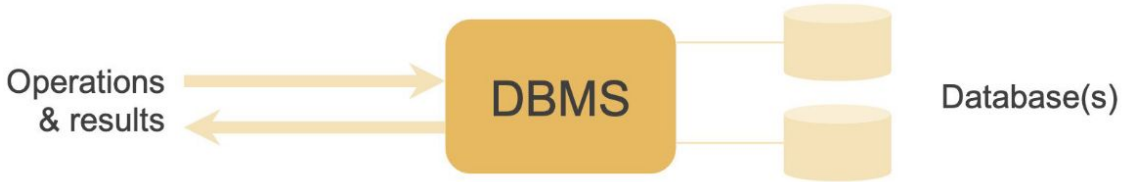
Reminder

Our definition

- We adopt a more generic definition

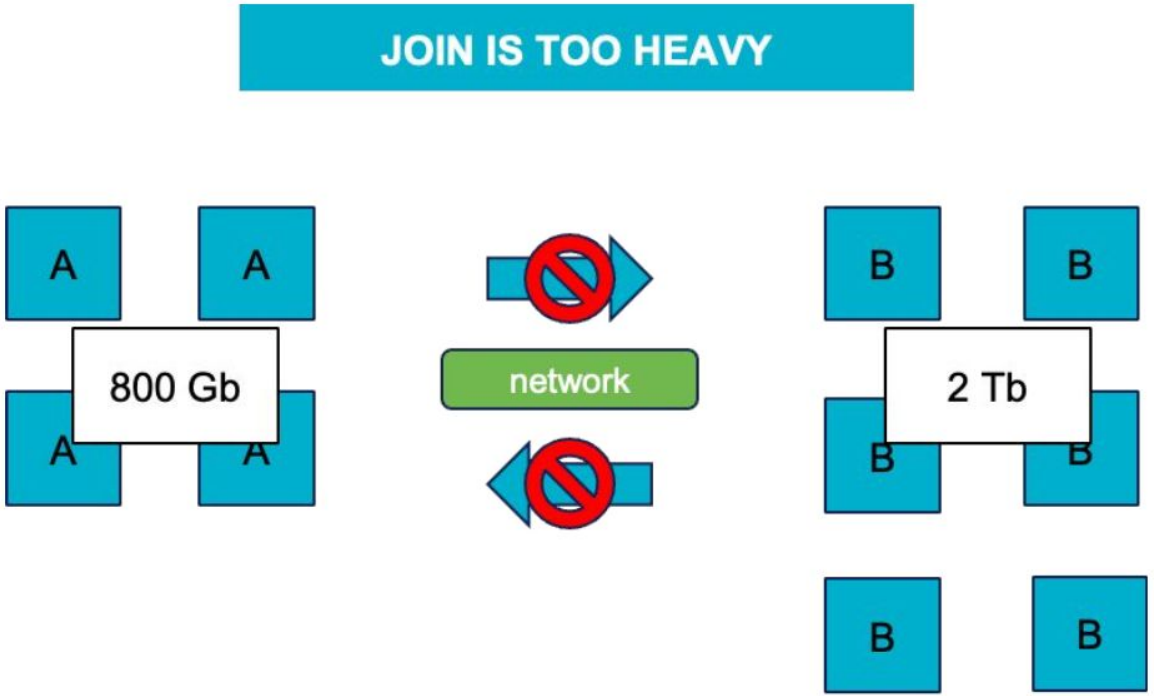
A database is a collection of data that is accessible from a common entry point

- All the other aspects (storage medium, list of operations, concurrent access, existence of a computer application, ...) will depend on the context; most of them are under the responsibility of a **D**ata**B**ase **M**anagement **S**ystem (DBMS)
  - Examples of DBMSes : Oracle, MySQL, MongoDB, ...
  - An operating system is also a kind of DBMS, through its filesystem APIs



The French abbreviation for DBMS is SGBD (**S**ystème de **G**estion de **B**ases de **D**onnées)

DNA2 - A DBMS with a simplified query model to allow distributed queries



Le transfert sur le réseau des données est trop couteux pour effectuer une jointure sur des transactions courtes (< 1 / 2 minutes).

Il existe des options pour effectuer des jointures limitées sur des bases distribués mais ces techniques dépassent le cadre de ce cours.

We accept losing some features…

- The new architectures were made possible by relaxing some of the **core assumptions** of the relational model. Here are some features that most of those new databases consciously left aside (variations exist)

We (often) lose…
ACID transactions. Consistency and Isolation are usually not guaranteed; Atomicity is limited to operations that affect only one entity
Foreign key constraints (consequence of the lack of consistency)
Storage/representation as tables and tuples
Data processing and transformation – the database is focused on storage
And, as a consequence of all the above, the SQL language, including **joins**

- The term **NoSQL** was first coined to unite those new DBMSes under a common banner: an active opposition to the SQL/relational paradigms. It then took the broader meaning of **N**ot **O**nly **SQL**, alluding to the fact that NoSQL DBMSes are just new, alternative ways of storing data

… to get new ones

- The previous discussion implies that we "lose" features when switching to a NoSQL DBMS. This is true, but what to we get instead?

We get…
The ability to scale in our out, by changing the size of the cluster, controlling the operational costs of the database. Cost is a **very important driver** here
Higher availability – with the new architectures it's easier to achieve high availability
The ability to spread data across several data centers (RDBMSes don't do it well)
Better performance, thanks to the distribution of data and the lack of transactions
Simpler development, and more flexibility in the choice of the data model
Simpler administration, targeted at developers and not at highly specialized DBAs

- For the Internet giants, and for many companies running high traffic web sites, availability, performance and cost control are the **most important characteristics**. Each hour of unavailability, each slowdown in query time lead to significant loss of revenue – missed orders, image degradation, churn

Data distribution

- With most NoSQL databases, the data is not stored in one place (i.e. on one server). It is **distributed** among the nodes of the cluster. When created, an object A is assigned to a node in the cluster. This is called **sharding** – the amount of data assigned to a node is called a **shard** (also called **partition**)

- Having more cluster nodes implies a higher risk of having some nodes crash, or a network outage splitting the cluster in two. For this reason, and to avoid data loss, objects are also **replicated** across the clusters
  - The number of copies, called replicas, can be tuned. 3 replicas is a common figure

- The objects may move, as nodes crash or new nodes join the cluster, ready to take charge of some of the objects. Such events are usually handled automatically by the cluster; the operation of shuffling objects around to keep a fair repartition of data is called rebalancing

Data distribution

- With most NoSQL databases, the data is not stored in one place (i.e. on one server). It is **distributed** among the nodes of the cluster. When created, an object A is assigned to a node in the cluster. This is called **sharding** – the amount of data assigned to a node is called a **shard** (also called **partition**)

- Having more cluster nodes implies a higher risk of having some nodes crash, or a network outage splitting the cluster in two. For this reason, and to avoid data loss, objects are also **replicated** across the clusters
  - The number of copies, called replicas, can be tuned. 3 replicas is a common figure

- The objects may move, as nodes crash or new nodes join the cluster, ready to take charge of some of the objects. Such events are usually handled automatically by the cluster; the operation of shuffling objects around to keep a fair repartition of data is called rebalancing

Data distribution

- With most NoSQL databases, the data is not stored in one place (i.e. on one server). It is **distributed** among the nodes of the cluster. When created, an object A is assigned to a node in the cluster. This is called **sharding** – the amount of data assigned to a node is called a **shard** (also called **partition**)

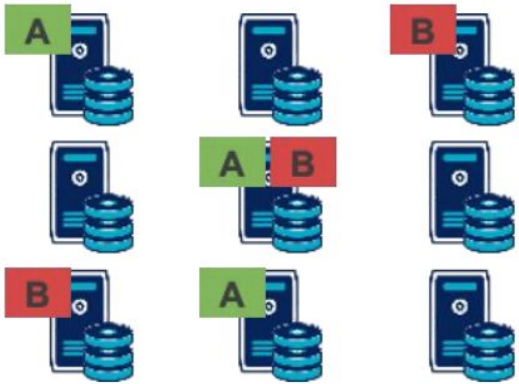
- Having more cluster nodes implies a higher risk of having some nodes crash, or a network outage splitting the cluster in two. For this reason, and to avoid data loss, objects are also **replicated** across the clusters
  - The number of copies, called replicas, can be tuned. 3 replicas is a common figure

- The objects may move, as nodes crash or new nodes join the cluster, ready to take charge of some of the objects. Such events are usually handled automatically by the cluster; the operation of shuffling objects around to keep a fair repartition of data is called rebalancing

Data distribution

- With most NoSQL databases, the data is not stored in one place (i.e. on one server). It is **distributed** among the nodes of the cluster. When created, an object A is assigned to a node in the cluster. This is called **sharding** – the amount of data assigned to a node is called a **shard** (also called **partition**)

- Having more cluster nodes implies a higher risk of having some nodes crash, or a network outage splitting the cluster in two. For this reason, and to avoid data loss, objects are also **replicated** across the clusters
  - The number of copies, called replicas, can be tuned. 3 replicas is a common figure

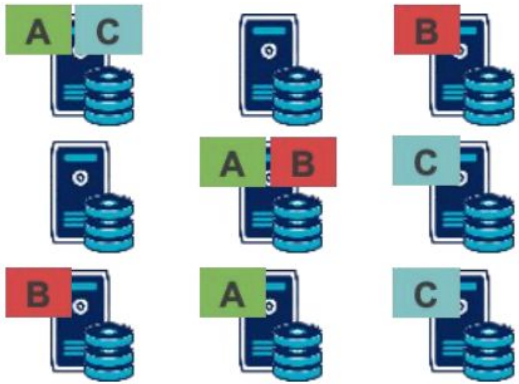


- The objects may move, as nodes crash or new nodes join the cluster, ready to take charge of some of the objects. Such events are usually handled automatically by the cluster; the operation of shuffling objects around to keep a fair repartition of data is called rebalancing

Data distribution

- With most NoSQL databases, the data is not stored in one place (i.e. on one server). It is **distributed** among the nodes of the cluster. When created, an object A is assigned to a node in the cluster. This is called **sharding** – the amount of data assigned to a node is called a **shard** (also called **partition**)

- Having more cluster nodes implies a higher risk of having some nodes crash, or a network outage splitting the cluster in two. For this reason, and to avoid data loss, objects are also **replicated** across the clusters
  - > The number of copies, called replicas, can be tuned. 3 replicas is a common figure

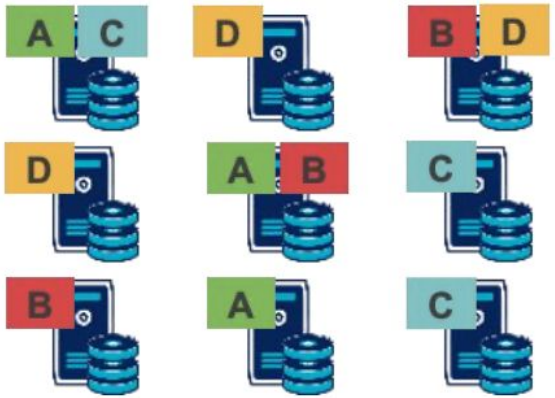


- The objects may move, as nodes crash or new nodes join the cluster, ready to take charge of some of the objects. Such events are usually handled automatically by the cluster; the operation of shuffling objects around to keep a fair repartition of data is called rebalancing

Data distribution

- With most NoSQL databases, the data is not stored in one place (i.e. on one server). It is **distributed** among the nodes of the cluster. When created, an object A is assigned to a node in the cluster. This is called **sharding** – the amount of data assigned to a node is called a **shard** (also called **partition**)

- Having more cluster nodes implies a higher risk of having some nodes crash, or a network outage splitting the cluster in two. For this reason, and to avoid data loss, objects are also **replicated** across the clusters
  - The number of copies, called replicas, can be tuned. 3 replicas is a common figure

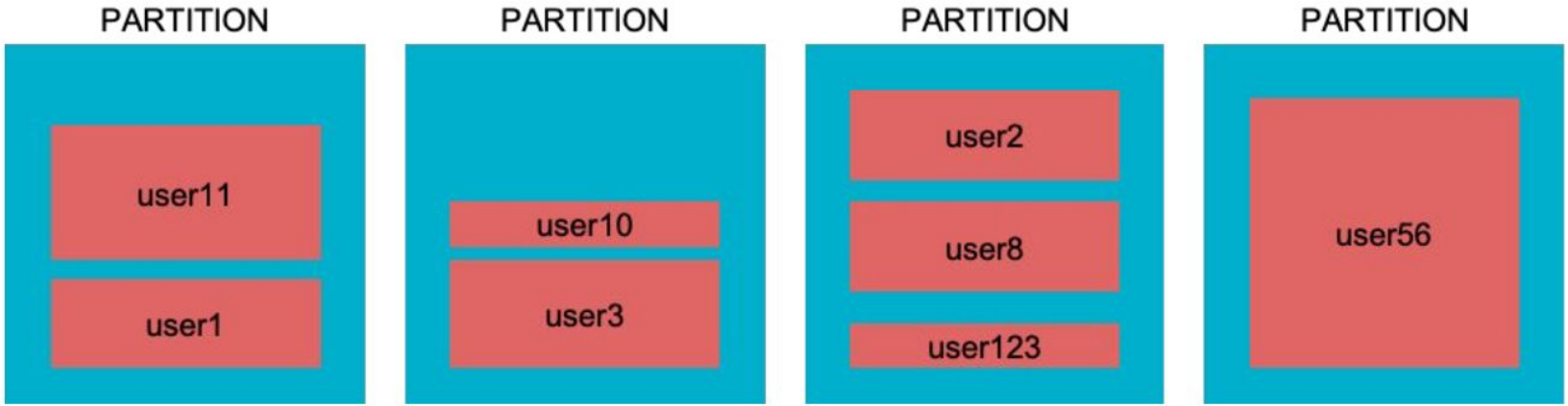


- The objects may move, as nodes crash or new nodes join the cluster, ready to take charge of some of the objects. Such events are usually handled automatically by the cluster; the operation of shuffling objects around to keep a fair repartition of data is called **rebalancing**

Data distribution - principe d'une clé de sharding

La clé de sharding d'un enregistrement détermine la partition d'un cluster où cet enregistrement est écrit.

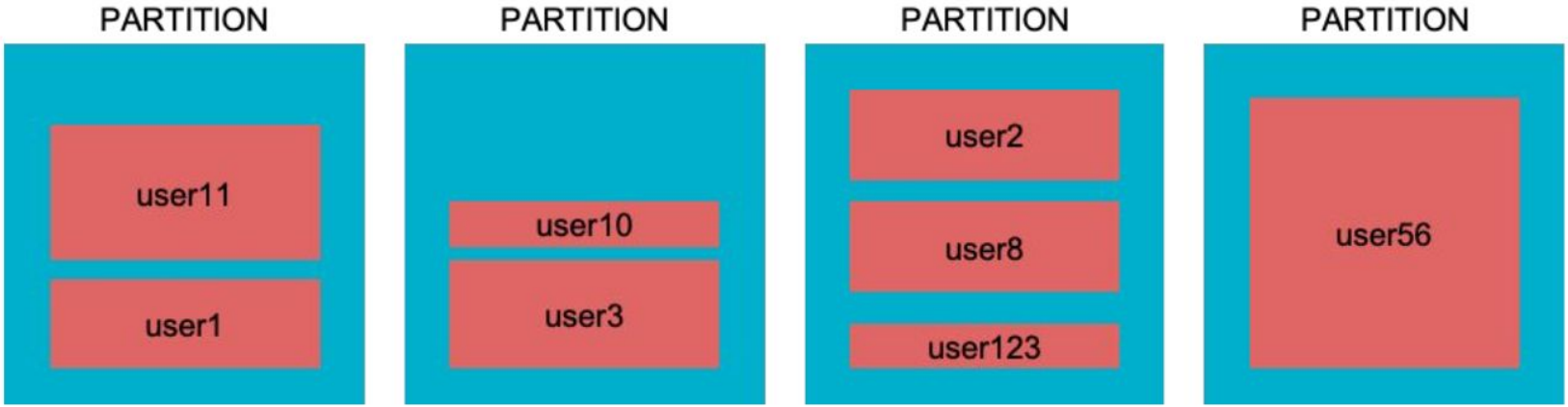
clé de partition	to	from
user1	brandy53@richard.biz	donfleming@white.com
user1	anngardner@hotmail.com	brandy53@richard.biz
user2	brandon93@yahoo.com	markperez@yahoo.com



Data distribution - principe d'une clé de sharding

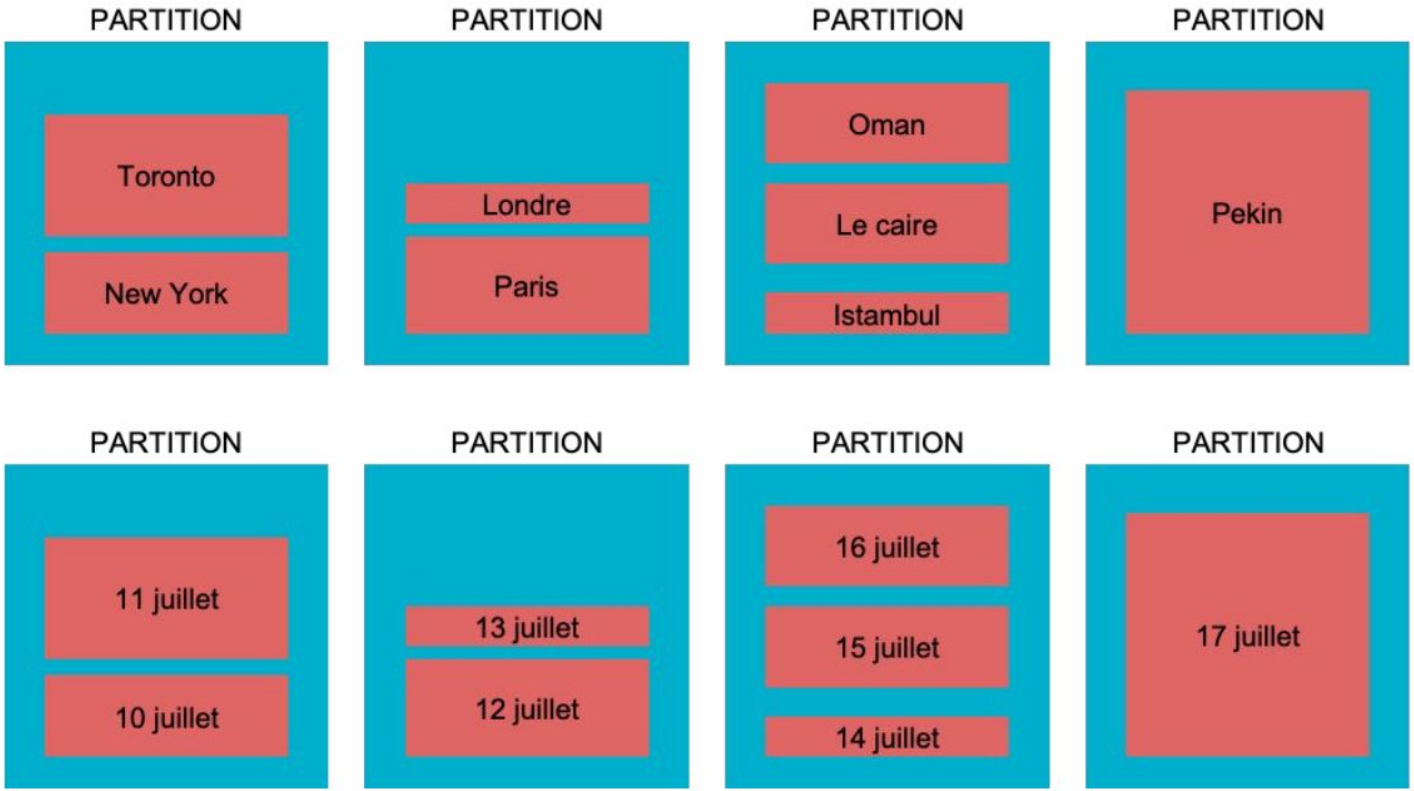
La clé de sharding permet de distribuer le traitement des requêtes à des noeuds de traitement autonome. Ce mécanisme est à la base des applications SaaS grand public.

- retrouver les 50 derniers emails de userX
- rechercher les emails de userX qui contiennent le mot clé Poney
- afficher les emails envoyés entre le 25/12/2019 et le 31/12/2019



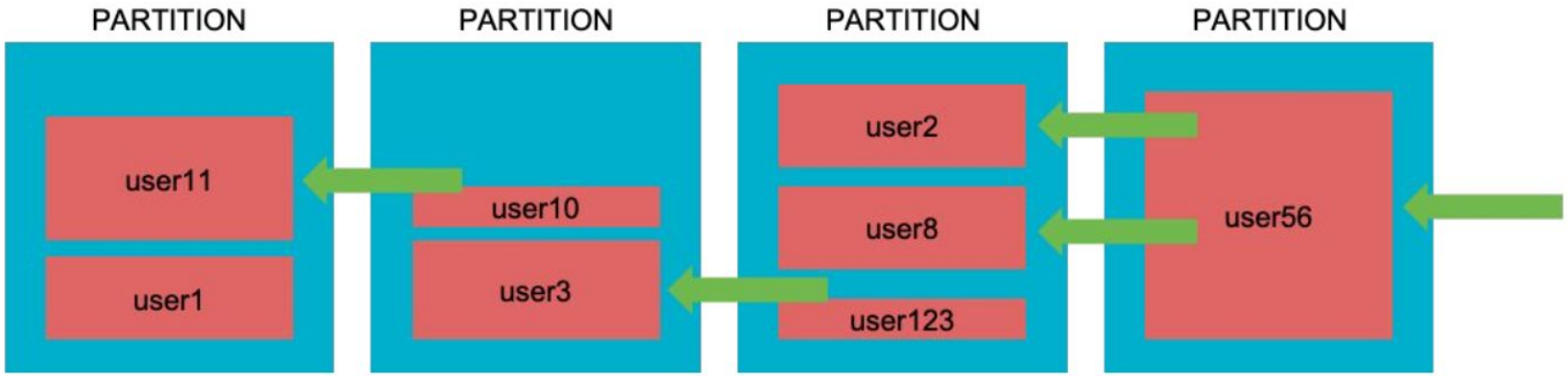
Data distribution - principe d'une clé de sharding

- Des attributs naturels comme la localisation ou la date peuvent servir de clé de sharding.



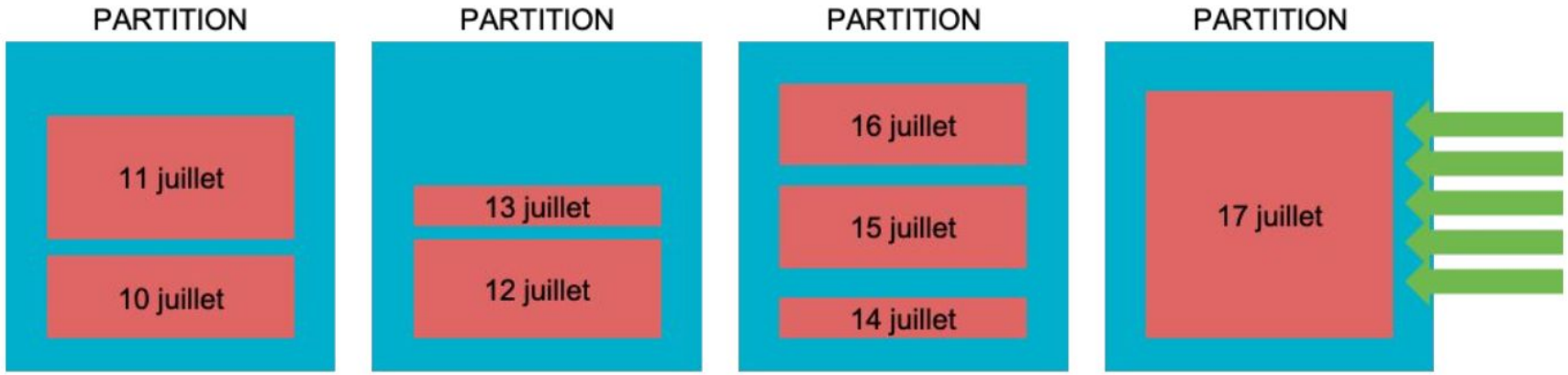
Data distribution - lecture / écriture distribuée

- L'utilisation de la clé de sharding dans les prédicats de la requete permet de distribuer la lecture des enregistrements sur l'ensemble du cluster.



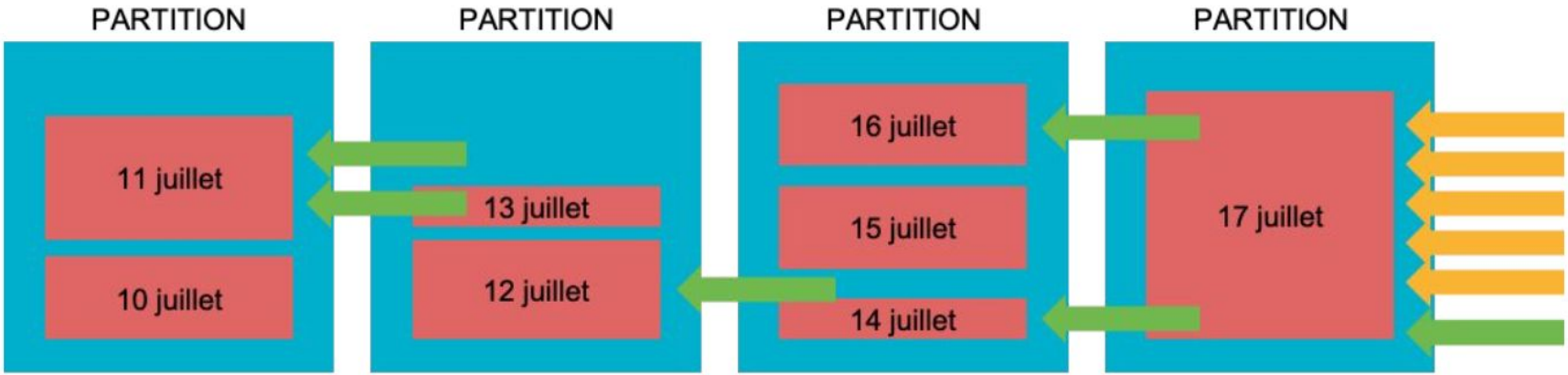
Data distribution - ANTI PATTERN - lecture / écriture concentrée

- L'utilisation de la clé de sharding dans les prédicats de la requete réalise des opérations sur une seule partition. La charge de travail du cluster est concentrée sur un noeud.



Trade off - lecture distribuée / écriture concentrée

- Le modèle de requête diffère entre l'écriture qui se fait sur la partition la plus récente et la lecture qui se fait sur l'ensemble des partitions. C'est un trade-off de design qu'il vous faudra faire lors du design d'une clé de sharding.



Some characteristics of a good sharding key

The perfect shard key would have the following characteristics:

- All inserts, updates, and deletes would each be distributed uniformly across all of the shards in the cluster
- All queries would be uniformly distributed across all of the shards in the cluster
- All operations would only target the shards of interest: an update or delete would never be sent to a shard which didn't own the data being modified
- Similarly, a query would never be sent to a shard which holds none of the data being queried

To Keep in mind

High-level concepts

SEMI-STRUCTURED DATA

- In most of the NoSQL databases, it is possible to insert data in a collection without specifying a schema or by providing a partial schema

SIMPLIFIED QUERY MODEL

- NoSQL database only support simple operation as insert, update, delete and query with projection and selection

DATA DISTRIBUTION

- In order to scale out, data must be distributed on large cluster
- To ensure high availability, data is replicated at least 3 times on different node

PARTITION & SHARDING KEY

- Partition is a logical unit that allow to distributed data on a cluster
- Statistic distribution of sharding key is a requirement to ensure scale out of distributed database

CAP theorem to popularise NoSQL

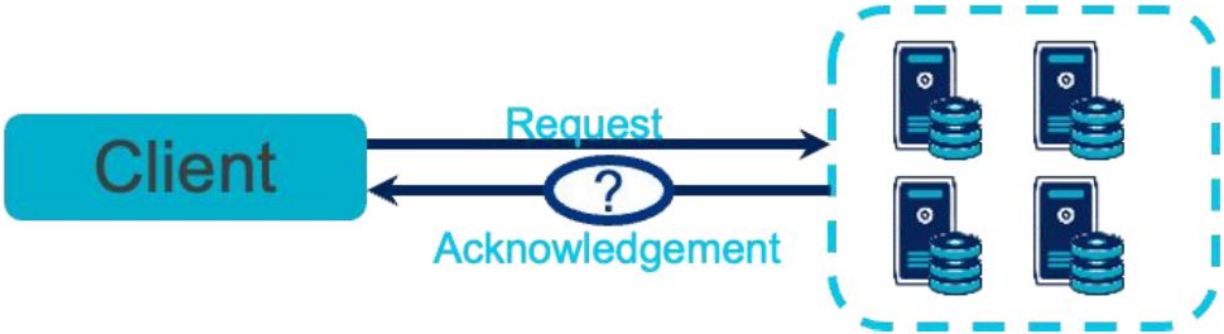
- Availability
- Consistency
- Partition Tolerance

Availability

- Availability (or lack thereof) is a property of the database cluster. The cluster is **available** if a request made by a client is always acknowledged by the system, i.e. it is guaranteed to be taken into account
- That doesn't mean that the request is processed immediately. It may be put on hold. An available system will at a minimum acknowledge it
- Practically speaking, availability is usually measured in percents. For instance, 99.99% availability means that the system is unavailable at most 0.01% of the time, that is, at most 53 min per year

Availability

- Availability (or lack thereof) is a property of the database cluster. The cluster is **available** if a request made by a client is always acknowledged by the system, i.e. it is guaranteed to be taken into account
- That doesn't mean that the request is processed immediately. It may be put on hold. An available system will at a minimum acknowledge it
- Practically speaking, availability is usually measured in percents. For instance, 99.99% availability means that the system is unavailable at most 0.01% of the time, that is, at most 53 min per year



Consistency

- When talking about distributed databases, like NoSQL, consistency has a meaning that is somewhat more precise than in the relational context

- It refers to the fact that all replicas of an entity, identified by a key in the database, **have the same value whatever the node queried**

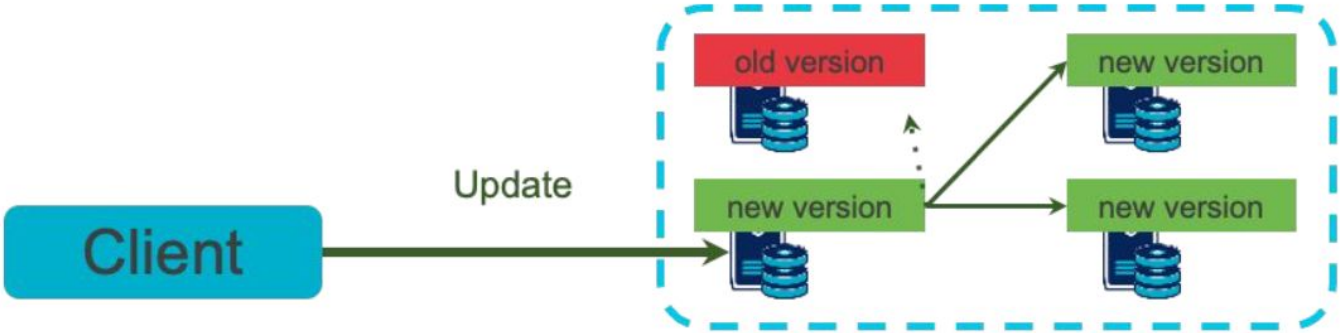
- With many NoSQL databases, updates take a little time to propagate across the cluster. When an entity's value has just been created or modified, there is a short span during which the entity is not consistent. However the cluster guarantees that it will eventually be, when replication has occurred. This is called **eventual consistency**

Consistency

- When talking about distributed databases, like NoSQL, consistency has a meaning that is somewhat more precise than in the relational context

- It refers to the fact that all replicas of an entity, identified by a key in the database, **have the same value whatever the node queried**

- With many NoSQL databases, updates take a little time to propagate across the cluster. When an entity's value has just been created or modified, there is a short span during which the entity is not consistent. However the cluster guarantees that it will eventually be, when replication has occurred. This is called **eventual consistency**



Partition Tolerance

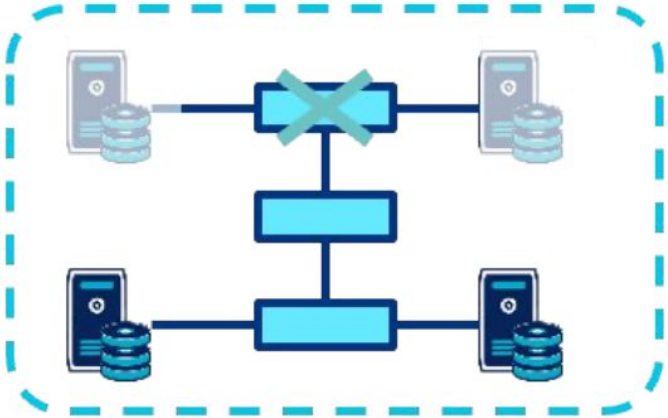
- This property is verified if a system made of several interconnected nodes can stand a **partition** of the cluster. In other words, the system is tolerant if it continues to operate when one or several nodes disappear. This happens when nodes crash or when a network equipment is shut down, taking a whole portion of the cluster away

- Partition tolerance is related to availability and consistency, but it is different still. It states that the system continues to function internally (e.g. ensuring data distribution and replication), whatever its interactions with a client

Partition Tolerance

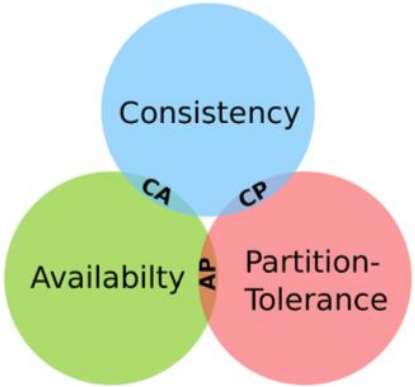
- This property is verified if a system made of several interconnected nodes can stand a **partition** of the cluster. In other words, the system is tolerant if it continues to operate when one or several nodes disappear. This happens when nodes crash or when a network equipment is shut down, taking a whole portion of the cluster away

- Partition tolerance is related to availability and consistency, but it is different still. It states that the system continues to function internally (e.g. ensuring data distribution and replication), whatever its interactions with a client



The CAP Theorem

- The previous 3 properties, **C**onsistency, **A**vailability and **P**artition tolerance, are not independent. The CAP theorem, or Brewer's theorem, states that a distributed system **cannot guarantee** all 3 properties at the same time



- This is a **theorem**. That means it is formally true, but in practice it is less severe than it seems
  - > The system or a client can often choose CA, AP or CP according to the context, and "walk" along the chosen edge by appropriate tuning
  - > Partition splits happen, but they are rare events (hopefully)

- **Rule of thumb**
  - > Traditional relational DBMSes are CA or CP – consistency is a must, in case of a problem either bring the cluster down or split it and expect heavy synchronization later
  - > Many NoSQL DBMSes are AP – availability is a must, and with big clusters failures happen so better live with it. Consistency is only eventual

02

Overview of the NoSQL data models

Example of NoSQL data models

Document-oriented (e.g. MongoDB)

OBJECTS

```
{ '_id': 123456,
  'type': 'product',
  'name': 'computer',
  'features': {
     'cpu_GHz': 3,
     'ram_GB': 8,
     'brand': 'Dell'
  }
}
```

```
{ '_id': 123457,
  'type': 'product',
  'name': 'blender',
  'features': {
     'rpm': 10000,
     'voltage': '220V 50 Hz'
  }
}
```

```
{ '_id': 123458,
  'type': 'user',
  'login': 'choupi92',
  'password': 'AZnx403==',
  'shopping_history': [...]
}
```

Column-family aka BigTable (e.g. Cassandra)

PRODUCTS

id	name	features		
123456	computer	cpu_GHz=3	ram_GB=8	brand=Dell
123457	blender	rpm=10000	voltage=220V 50 Hz	

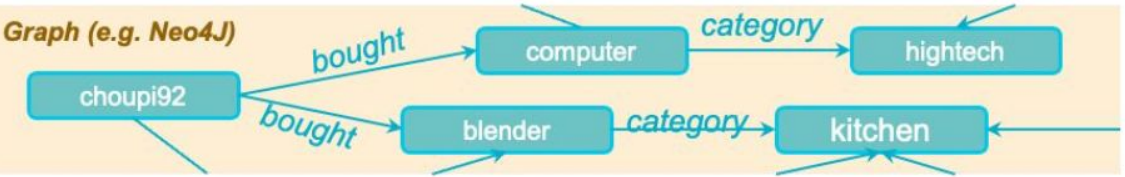
USERS

id	authent		shopping_history		
123458	login= choupi92	password=AZnx4 03==	08/09/13=…	10/09/13=…	•••

Key/Value pairs (e.g. Redis)

obj_123456	"type=product;name=computer;cpu_GHz=3;…"
obj_123457	"type=product;name=blender;rpm=10000;…"
obj_123458	"type=user;login=choupi92;password=…"

Graph (e.g. Neo4J)



Key-Value

- The key/value is the simplest model

- The developer has 2 **operations** at hand:
  - put(key, value)
  - get(key)

- The database is like a big, persistent **hash table** that stores (key, value) pairs. Keys and values are like opaque BLOBs. When the database is distributed, the placement of an object is determined by a hash of its key

- **Pros**
  - For the client, simplicity of the API
  - For the server, simplicity of the storage model. In particular sharding is easy
- **Cons**
  - Nothing prevents writing garbage into keys or values: the application must deal with it when unpacking the data
  - No indexing by content: how to easily retrieve all objects that have an rpm property?
- **When to use**
  - When the data to store is made of simple and independent objects, and uniform enough to be unpacked easily
  - When no complex queries are necessary (all processing occurs on the client)

obj_123456	"type=product;name=computer;cpu_GHz=3;…"
obj_123457	"type=product;name=blender;rpm=10000;…"
obj_123458	"type=user;login=choupi92;password=…"

Document-oriented

- In this model, objects are **documents**, i.e. trees of values
  - Each document has a root and attributes
  - Attribute values are scalars (integers, strings), lists or other objects
  - Each object has a unique ID, a conventional property whose value serves as a key

- Objects are organized into **collections**. Objects in the same collection need not have the same schema – there is no mandatory structure

- **Pros**
  - All programming languages support JSON or XML, which are natural representations for objects
  - Sharding is easy
  - Complex indexing on content (property values) is possible
  - Updates on a particular object can be made atomic
- **Cons**
  - Object-to-object references must be emulated by remembering IDs; following pointers is not efficient
  - When frequent updates occur inside objects, fragmentation occurs
  - Analytic queries (averaging the product prices for example) is not efficient because all documents must be read and inspected
- **When to use**
  - When storing documents in the broader sense: web pages, blog posts, machine logs, …
  - When inter-document relationships are not essential

Document-oriented (e.g. MongoDB)

OBJECTS

```
{ '_id': 123456,
  'type': 'product',
  'name': 'computer',
  'features': {
    'cpu_GHz': 3,
    'ram_GB': 8,
    'brand': 'Dell'
  }
}
```

```
{ '_id': 123457,
  'type': 'product',
  'name': 'blender',
  'features': {
    'rpm': 10000,
    'voltage': '220V 50 Hz'
  }
}
```

```
{ '_id': 123458,
  'type': 'user',
  'login': 'choupi92',
  'password': 'AZnx403==',
  'shopping_history': [...]
}
```

Column family (BigTable)

- The column-family model looks a bit like the relational model
  - > Data is organized into **tables** and **rows**
  - > Tables have fixed **column families**, in which arbitrary columns are nested
  - > For a given row, the contents of a column can thus be seen as a hash table with arbitrary (key, value) pairs
  - > Each row in a table is uniquely identified by a key

- **Pros**
  - > Very efficient on write
  - > Queries are efficient if data are stored in a row. column families are stored together on disk (reading only one column does not require reading the whole row, hence less I/O)
- **Cons**
  - > Storing documents à la MongoDB requires a modelization effort
- **When to use**
  - > When large objects with many columns fit the column family model
  - > When the application makes frequent accesses to portions of a row
  - > Time series

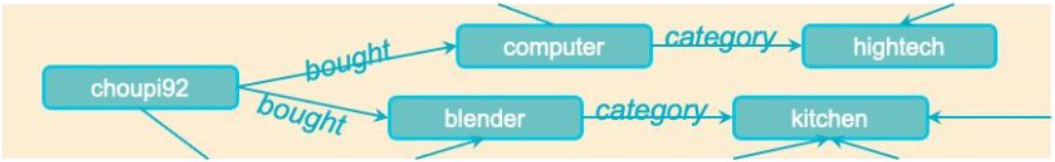
USERS					
id	authent		shopping_history		
123458	login= choupi92	password=AZ nx403==	08/09/13=…	10/09/13=…	•••

Graph

- The data is made of entities linked together by relationships. This is actually a **graph**, with the nodes (or **vertices**) being the entities, and the **edges** the relationships

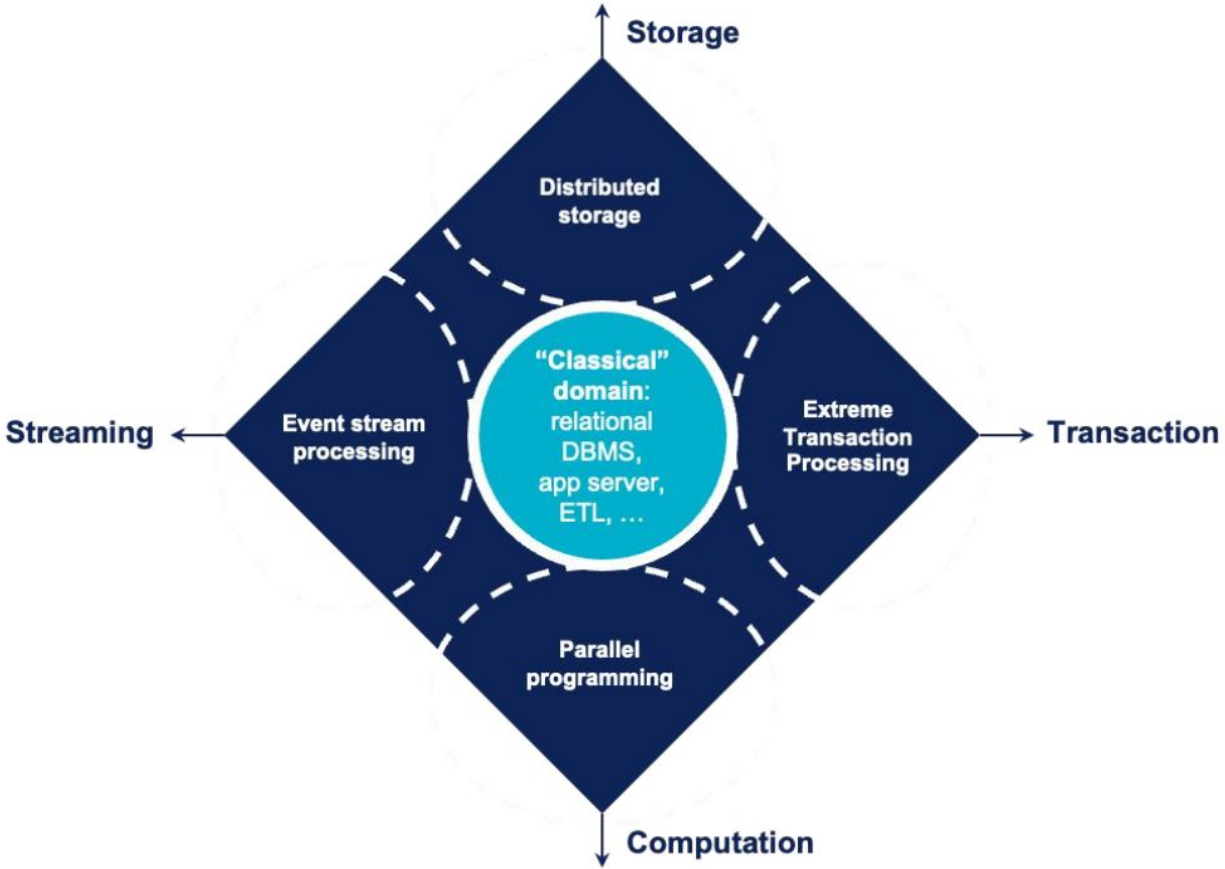
- Vertices and edges have a dictionary of properties
- For example, a user vertex bears a login, a last connection date, … and a user-to-product relationship has a type (bought, visited, …) and other properties like the date of the interaction, the context in which it occurred, and so on

- **Pros**
  - > This representation can model almost anything; in particular it can simulate all the other models. The entities need not be of the same type or belong to collections of related objects
  - > Traversing the graph is efficient because the DBMS optimizes those scenarios
  - > Graph theory is very rich and readily applicable: shortest path calculation, degree distribution, centrality measures, …
- **Cons**
  - > Traversal is not as efficient if many edges link vertices that are on different shards, and finding an optimal distribution is computationally hard. Graph DBMSes are not yet very good at this but they're getting better…
- **When to use**
  - > When one must store data that exhibits a lot of mutual relationships between entities
  - > Graphs are a hot topic, they'll soon be omnipresent (internet of things, smart cities, …)

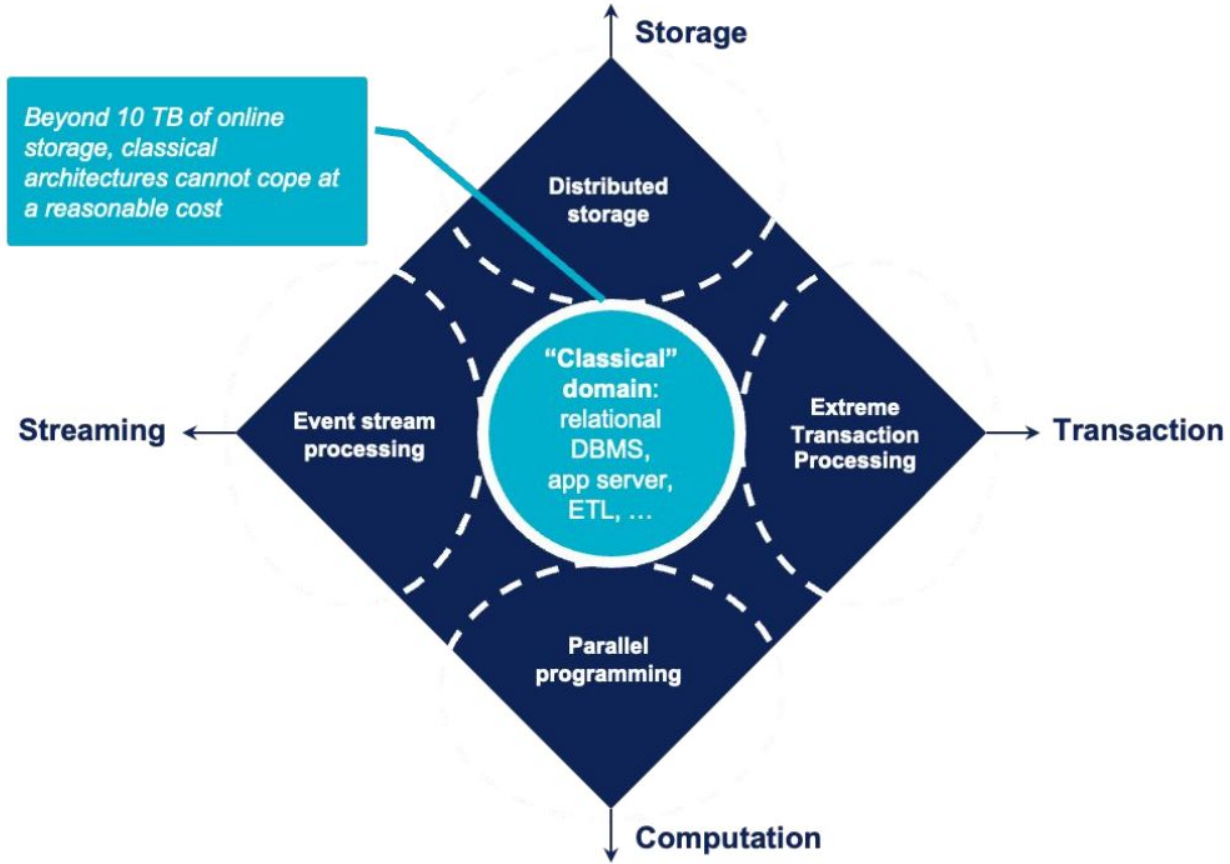


Neo4j
the graph database

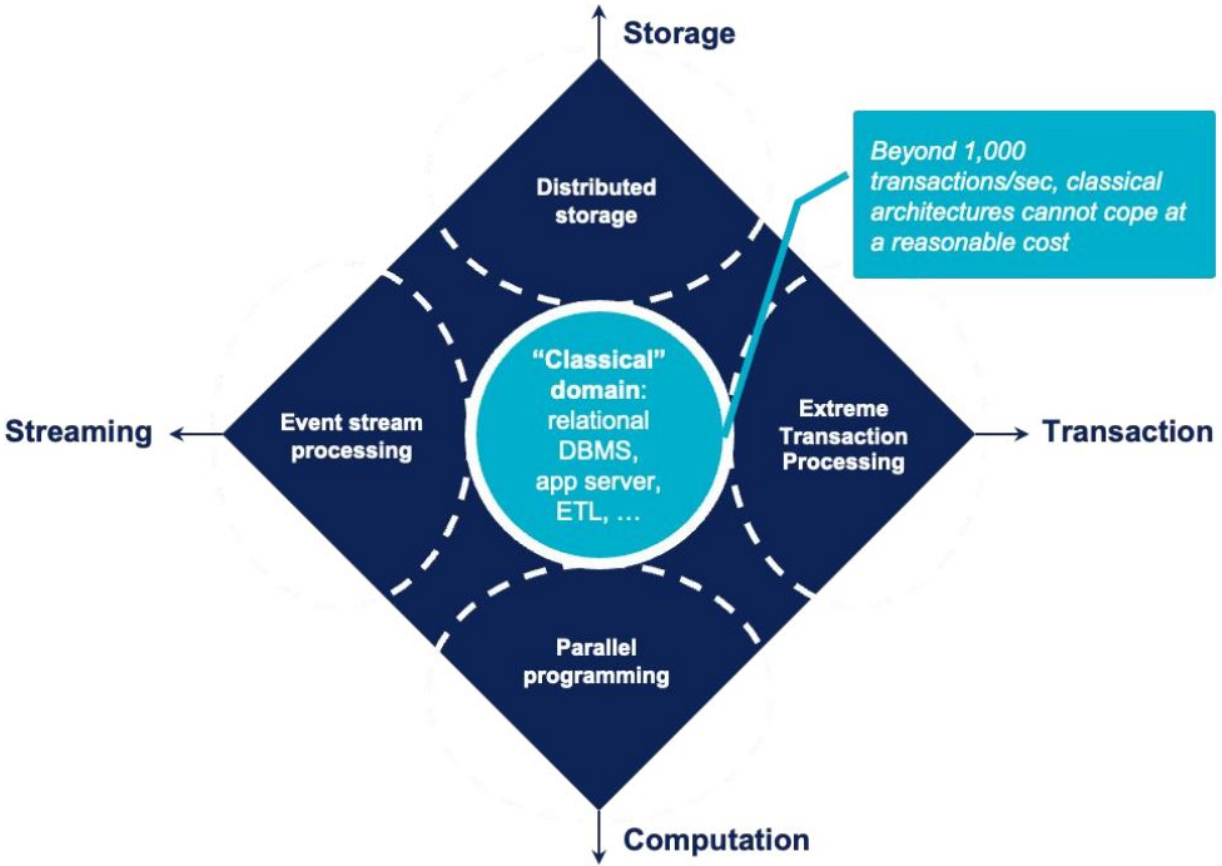
Choisir une famille technologique à partir du modèle en diamant de limitations des DBMS classiques



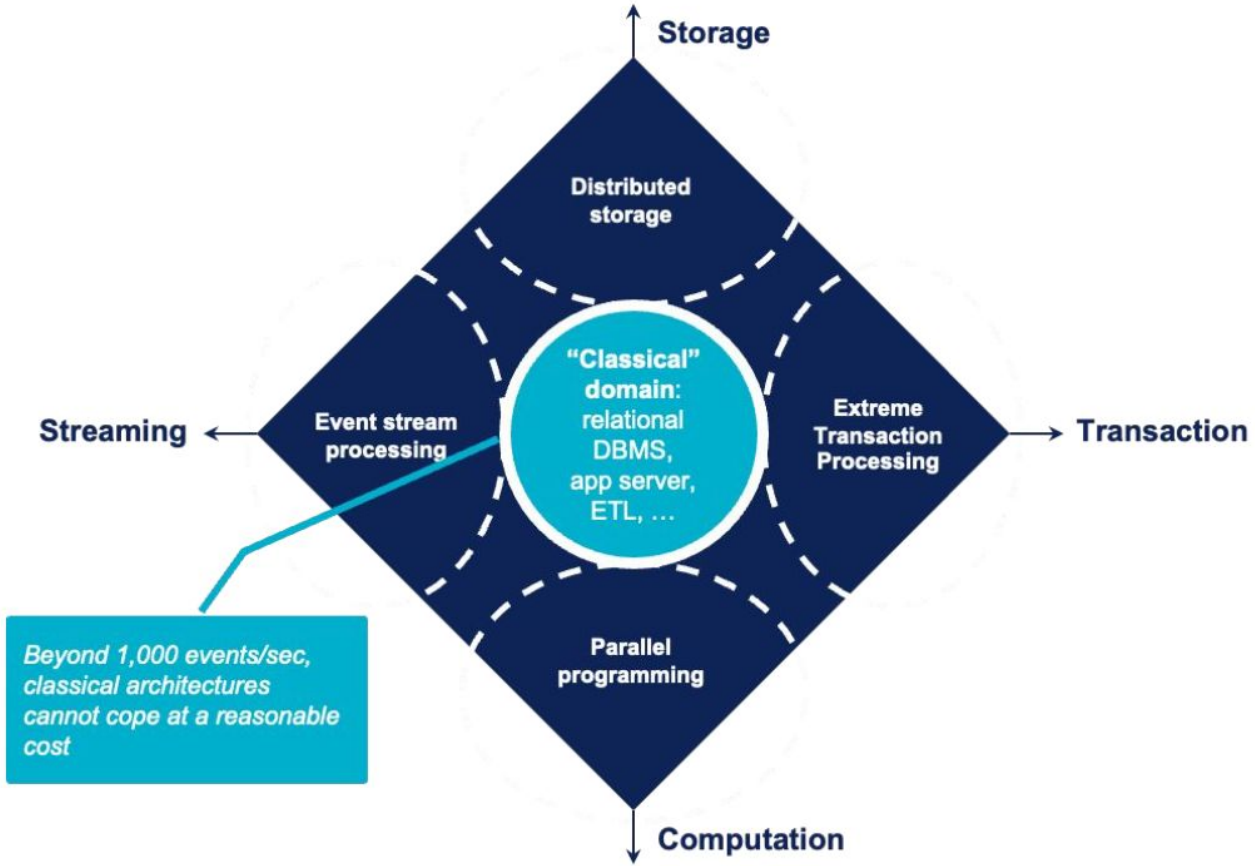
Choisir une famille technologique à partir du modèle en diamant de limitations des DBMS classiques



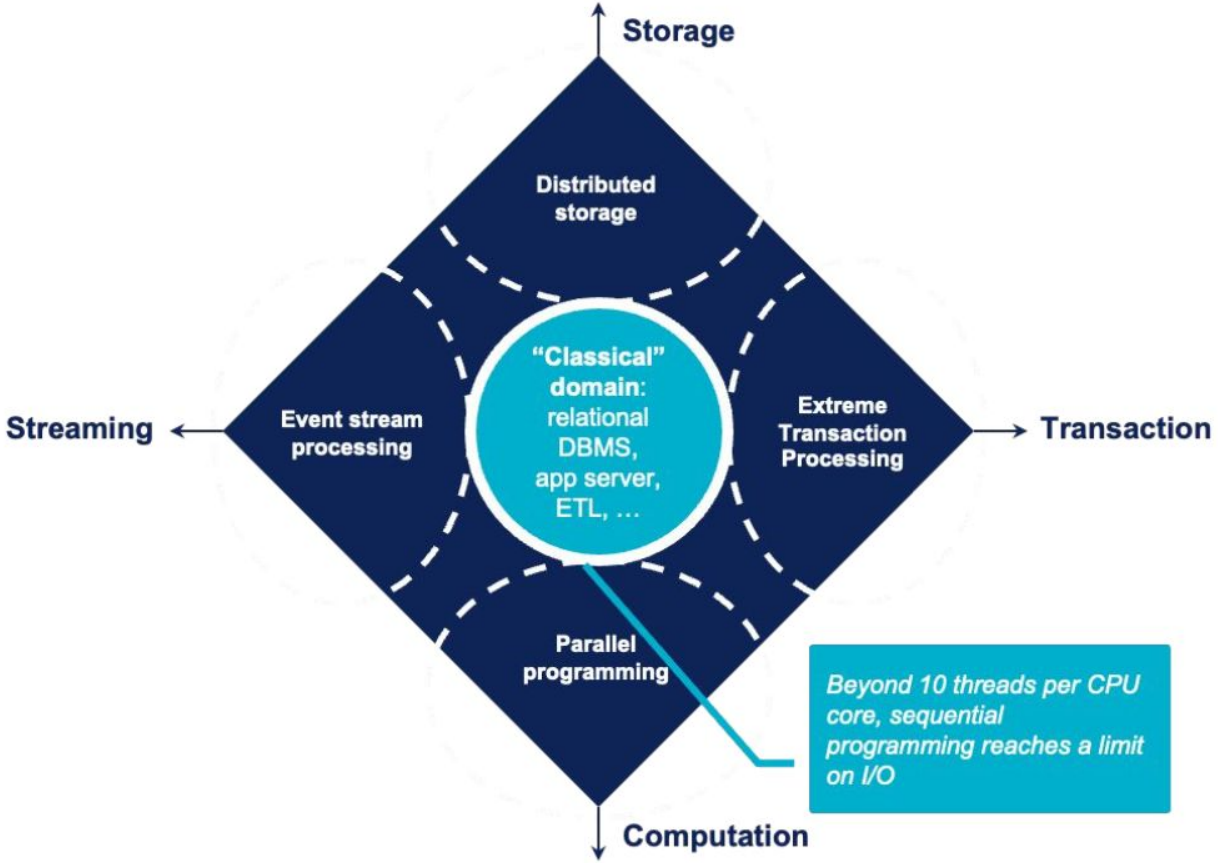
Choisir une famille technologique à partir du modèle en diamant de limitations des DBMS classiques



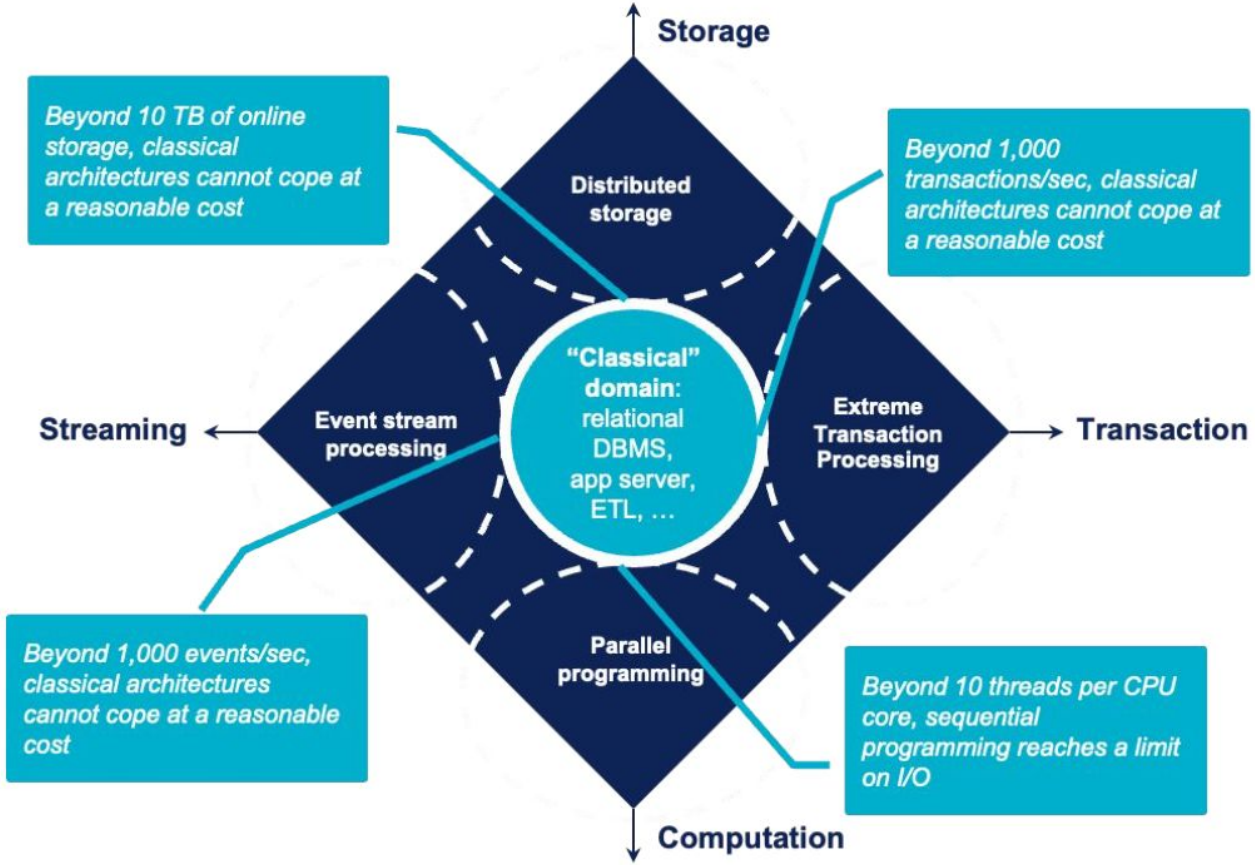
Choisir une famille technologique à partir du modèle en diamant de limitations des DBMS classiques



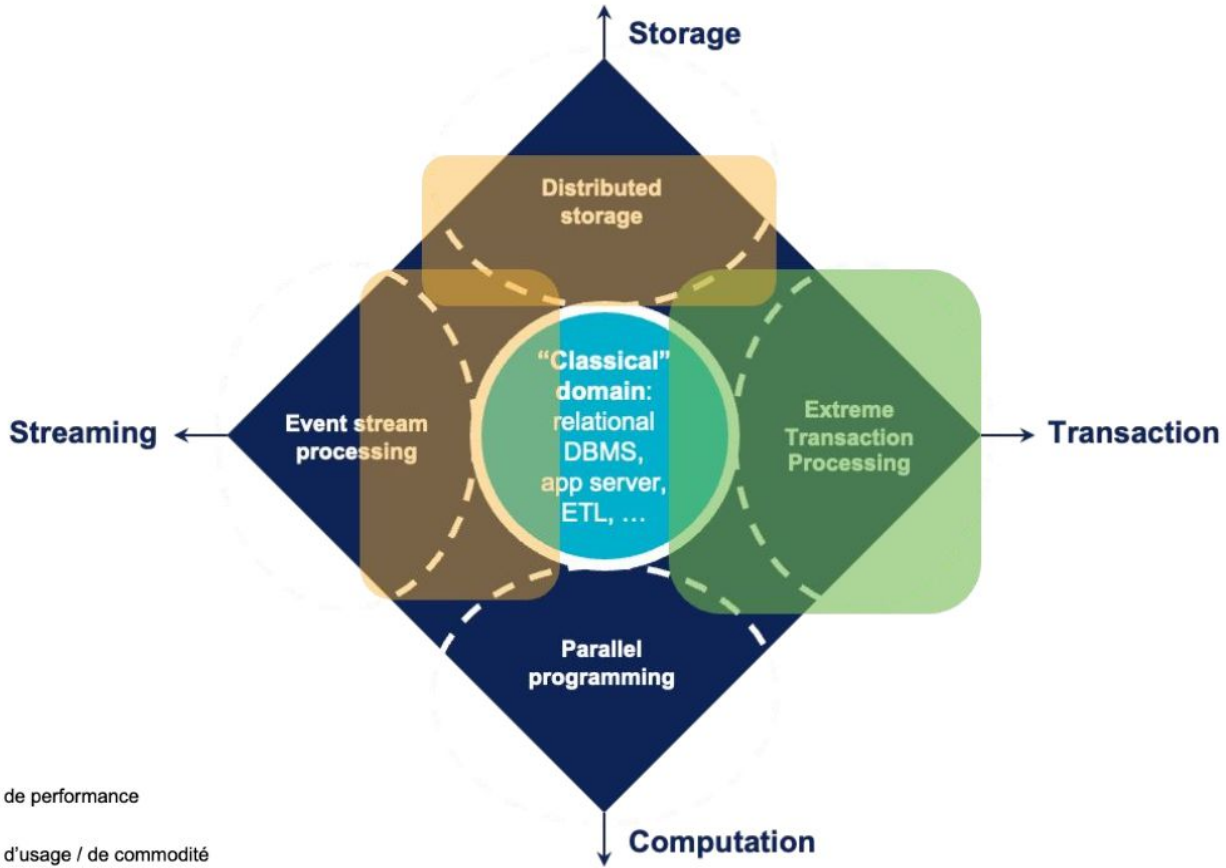
Choisir une famille technologique à partir du modèle en diamant de limitations des DBMS classiques



Choisir une famille technologique à partir du modèle en diamant de limitations des DBMS classiques



NoSQL usages'



zone de performance

zone d'usage / de commodité

03

Choose an architecture

Rule 1: KEEP IT SIMPLE

- Some database engines as Postgresql, Oracle or MSSql offers a lot of features as document storage, full text research, … to help you to design a system.
- They allow to move later the architecture design decision. This flexibility is a benefits when the usage of the system is not clear enough.

KISS : a system should be simple enough to be understandable by an average developer of your team in case of urgent modification using the tool he have.

- NoSQL databases are great assets to fulfill very specific needs. If the need change, take time to introduce a new system able to handle it. They have limitation.
- Document database is not design to handle many-to-many relation. If you are using it for that, it may only work when the number of document is low.
- Column Family database is not design to handle well secondary index. For Cassandra, the feature exists but the documentation is clear about the limitation.

Rule 2: COMPOSE A DATA STORAGE SOLUTION

- When you are designing some part of an information system, the use case may be too complex to fit on one database solution.

- Differents data models fits different usage.

usage	pattern	Database engine
Click stream	Mostly write simple events	Column family database
Click stream (analytic)	Read query on massive dataset	Columnar database
Blog display	Mostly read consistent document	Document database
Blog search	Scoring on full text	Search engine
…	…	…

- This system requires careful design for data management. Data consistency may become a nightmare. *A good practice is to have a* ***single source of truth*** *for data type* and ensure same data replicate on other system (as search engine) derives from this single source of truth.

Les Patterns De Choix Étendus

- Design by query
  - > Problème : connaître la localisation des données est indispensable pour avoir des performances linéaires
- Map - Reduce
  - > Problème : appliquer des calculs d'agrégation sur une base distribuée
- Massively Parallel Processing
  - > Problème : appliquer un prédicat sur l'ensemble d'une base de donnée
- Schema on Write
  - > Problème : assurer la qualité de la donnée à l'écriture

04

Focus on MongoDB

Overview of MongoDB

- MongoDB is one of the most well-known **document-oriented databases**. It can store and retrieve millions of documents with a very good performance

- Documents that are related together are stored in the same **collection**; one can define as many collections as necessary to accomodate for the various "types" of documents

- Documents are represented as JSON (**J**ava**S**cript **O**bject **N**otation) in MongoDB's native Javascript framework (BSON to be more precise)

- In-memory approach

- The storage engine is the component of the database that is responsible for managing how data is stored, both in memory and on disk
  - Latest storage engine : WiredTiger (from MongoDB 3.0)

- Several distributions exists :
  - MongoDB Community Edition
  - MongoDB Professional & Enterprise Advanced

MongoDB Vocabulary

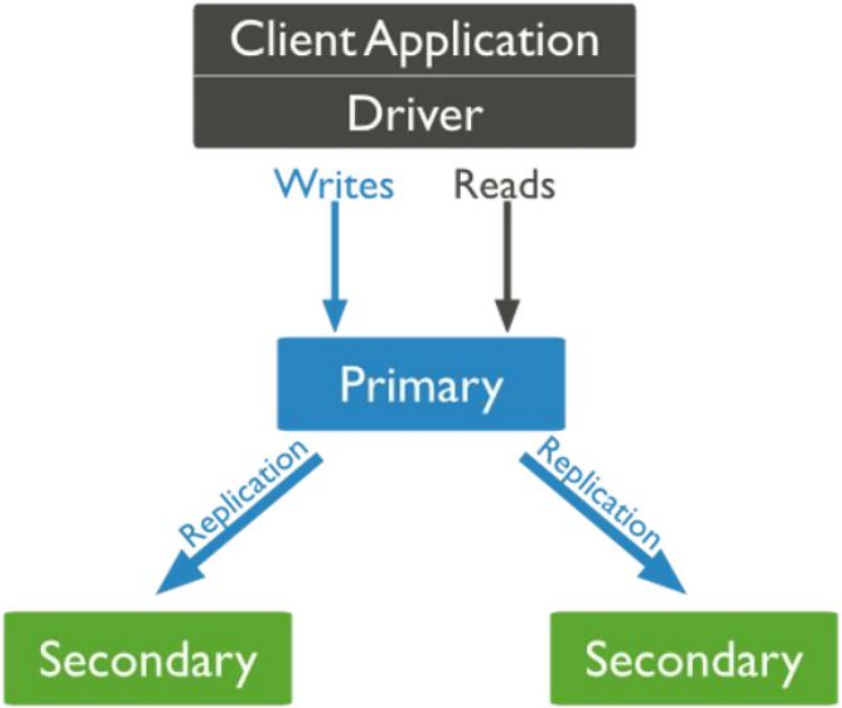
RDBMS (Oracle)	MongoDB
database	database
Table	Collection
Row	Document (Bson)
Index	Index
Join	Embedded & Linking
Partition	Shard
Partition key	Sharding key

Source : https://www.mongodb.com/docs/manual/reference/sql-comparison/

MongoDB's Services

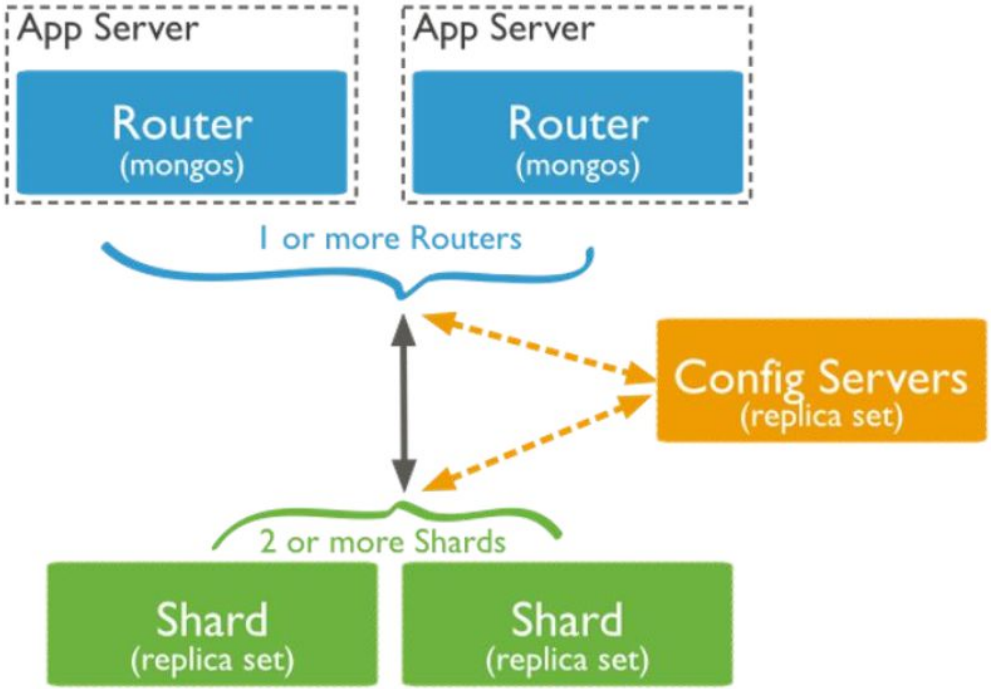
- **mongod** (*data node*)
  - storage service
  - within a Replica Set, a mongod can be a primary (master) or secondary (slave)
- **mongos** (*routing service*)
  - sharding service
  - routes the requests towards the proper servers
- **Config server** (*mongod*)
  - Stores the cluster's topology du cluster in a sharded cluster
  - without this information the cluster would be useless
  - must be at least 3 in order to ensure resilience
- **Arbiter** (*mongod*)
  - allows the reelection of the primary in a replica set with an even number of voting members
  - special mongod instances that do not store any data

MongoDB, Focus on the replica set



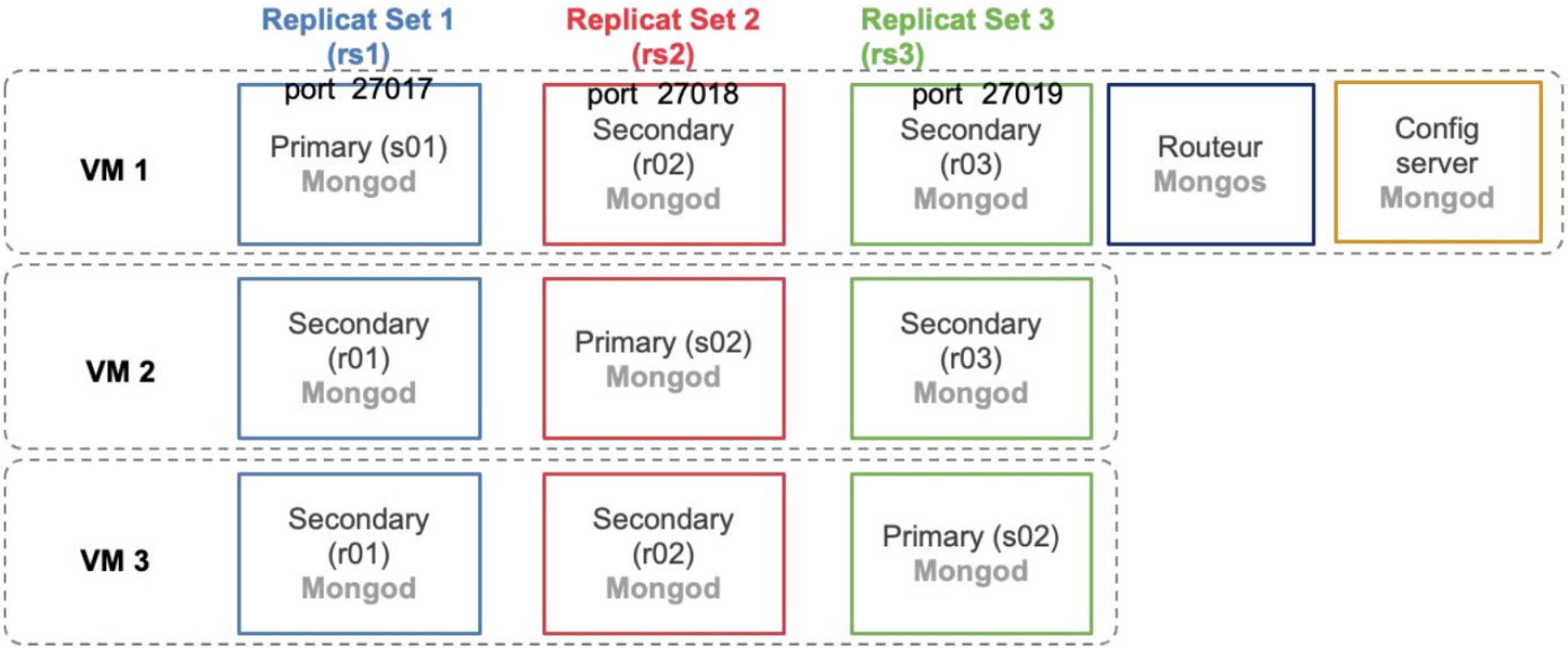
Source : https://www.mongodb.com/docs/manual/replication/

MongoDB shared cluster architecture

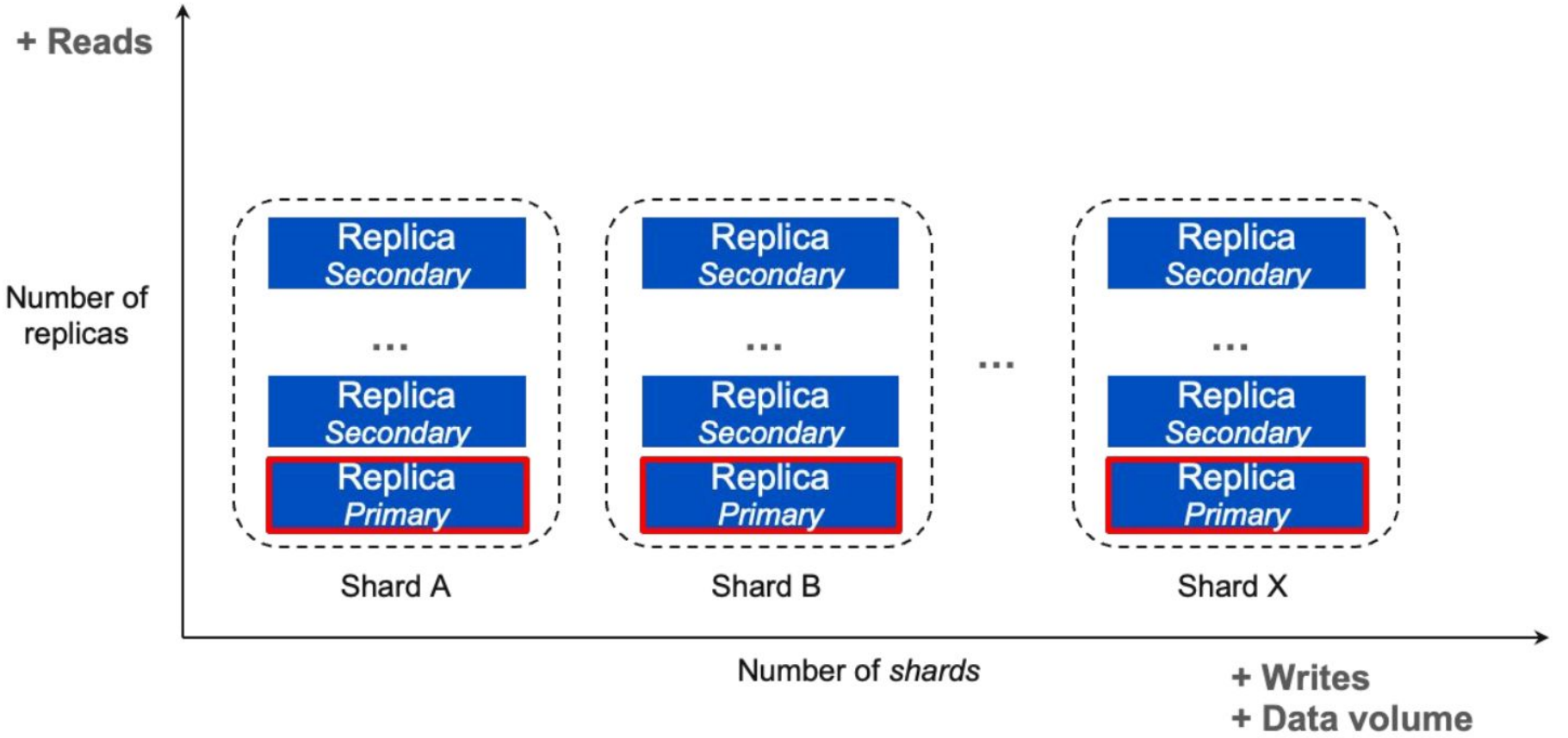


Source : https://www.mongodb.com/docs/manual/sharding/

MongoDB Architecture setup for a POC



MongoDB scalability

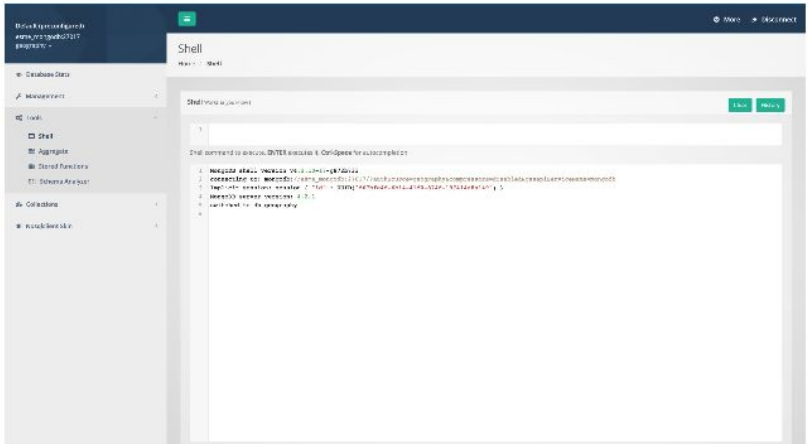


Tools

Besides the server and administration executables, MongoDB comes with a Javascript shell, called **mongo**, accessible on the command-line

The shell lets one issue queries as Javascript commands, like CRUD operations, map/reduce jobs, etc.

We will use a third-party GUI front-end to the shell, called **Robo3T**, **Studio3T** or **NosqlClient**



JSON in a nutshell

- JSON is the syntax used by Javascript to instantiate anonymous objects, without having to define a class first. In MongoDB, we use bson, an extension of this langage.

```
var myObject = {
   anIntegerProperty: 1,
   aStringProperty: "This is a string",
   anArray: [1, 2, 3, 4],
   aSubObject: {
      iHavePropertiesToo: "Yes indeed",
      anArrayOfObjects: [
         { shortObject: 42 },
         { moreComplex: "embedded",
           evenMore: {
              question: "How deep am I?",
              answer: 5
           }
         },
         { andSoOn: "Down to any level" }
      ]
   }
};
myObject.aSubObject.anArrayOfObjects[1].evenMore.quest
ion;
```

Creating objects in the database

- In the shell, there is an object called db, that exposes the data manipulation operations. All such operations occur in a collection and are prefixed by db.collectionName
- Example of an insertion

```
db.products.save( { name: "Computer", price: 1000 } );
```

- When doing that, MongoDB automatically assigns an ID to the new object: it provides a *key*. The ID attribute has the special name _id. We can also provide the _id ourselves, provided it doesn't already exist in the collection
- If there was an object with _id 123, it has been overwritten. This can be prevented by using the insert() method instead of save(): **insert() will fail if it is provided an already existing _id**

```
db.products.save( { _id: 123, name: "Computer", price: 1000 } );
```

```
db.products.insert( { _id: 123, name: "Computer", price: 1000 } );
```

will return an error because we wanna insert the same id

Updating objects in the database

- Overwriting an object whose _id we know is simply a matter of calling save(). The new "version" of the object can have totally different properties

```
db.products.save( { _id: 123, hddSizesInGB: [60, 250, 250] } );
```

- Otherwise, we use the update() function with an **object template**, and a $set operation

```
db.products.update(
  { name: "Computer", price: 1000 },
  { $set: { ramInGB: 8 } }
);
```

Note the braces around $set: this is an object passed as an argument to update()

- This will add a property ramInGB to one object whose properties name and price match the template. If we want to update all matching objects, we add a parameter:

```
db.products.update(
  { name: "Computer", price: 1000 },
  { $set: { ramInGB: 8 } },
  multi = true
);
```

Updating objects in the database (cont'd)

- $set will add or modify properties of matching objects. There are other operators to perform more elaborate operations
  - $unset will remove a property from an object
  - $inc will increase the value of a property
  - $add, $addToSet, $push, $pull, $pop, … will manipulate array-valued properties

- To update nested objects, we can use the following notation

```
db.products.update(
  { name: "Computer", price: 1000 },
  { $set: { ramInGB: 8 } },
  multi = true
);
```

Note the quotes around the nested object's property names. They are syntactically necessary because of the dot in the middle of the name

  - The notation is valid in the object template, and in the operation itself ($set here)

- More on object templates in the next section

Querying

- Querying is done using the find() method
- With no arguments or an empty object template, find() returns all objects in the collection

```
db.products.find();
db.products.find( {} );
```

- Finding objects by equality on property values

```
// Find all products whose price is 1000
db.products.find( { price: 1000 } );

// Find all products who have an cpu whose GHz property is 2.7
db.products.find( { "cpu.GHz": 2.7 } );

// Combine 2 criteria with an AND operation
db.products.find( { price: 1000, brand: "Dell" } );

// Combine with an OR operation
db.products.find( { $or: { price: 1000, brand: "Dell" } } );
```

Querying (Cont'd)

- Finding objects by inequality

```
// Find all products whose price is not equal to 1000
// There also is $gt (>), $gte (>=), $lt (<), $lte (<=)
db.products.find( { price: { $ne: 1000 } } );

// Find all products whose price falls in a range
db.products.find( { price: { $gt: 500, $lt: 2000 } } );

// Find all products whose RAM amount takes a value in a list
// There also is a $nin (not in) operator
db.products.find( { ramInGB: { $in: [4, 8] } } );
```

- Matching nested objects

```
// Find all products that have an CPU exactly equal to
// { GHz: 2.7, cores: 4 }
// (exact same fields, in the same order, with the same values)
// Note the difference with the "cpu.GHz" notation)

db.products.find( { cpu: {
   GHz: 2.7,
   cores: 4
} } );
```

Querying (Cont'd)

- Restricting the number of properties in output (projection)

```
// Only show product name, price and _id (added by default)
db.products.find( {}, { name: 1, price: 1 } );

// Hide the _id
db.products.find( {}, { name: 1, price: 1, _id: 0 } );

// Show all properties except name and price
// You can't mix 1's and 0's, except for the special _id field
db.products.find( {}, { name: 0, price: 0 } );
```

- Paginating and sorting the results

```
// Only get 10 products
db.products.find().limit(10);

// Skip the first 5 products, and return the 10 next ones
db.products.find().skip(5).limit(10);

// This is really useful when some order is applied to the
// results first. Get the 10 most expensive products. The '-1'
// means descending order; '1' would mean ascending
db.products.find().sort( { price: -1 } ).limit(10)
```

Querying (Cont'd)

- By default, the find() method (and the implicit query in the update() method) will scan the whole collection, match each object against the object template, and process it if necessary
- This can of course be very inefficient if there is a large number of objects in the collection. The I/O cost will be very high
- If a query is frequently executed, the fields participating in a query may need to be indexed (as with SQL, don't do it if it's not necessary!)

```
// Index the name field of every object that has it
db.products.ensureIndex( { name: 1 } );
db.products.find( { name: "Computer" } ); // Fast!

// Same with a nested object's property
db.products.ensureIndex( { "cpu.GHz": 1 } );
db.products.find( { "cpu.GHz": 2.7 } );
```

ensureIndex() *is called on the collection.*
This is why collections should group objects that are related somehow, i.e. share the same indices

> You can also index a whole nested object, for queries that match those (see previous slide)
> The _id property is automatically indexed, no need to call ensureIndex() on it

Aggregations

- Aggregations are specified as a pipeline, i.e. an array of operations such as
  - Selecting objects ($match), much like a SQL WHERE or a HAVING clause (depending on its position in the pipeline)
  - Grouping them ($group), like a GROUP BY clause
  - Manipulating intermediate or final results ($limit, $skip, $sort)

- Unlike SQL, all the operations above can be involved as many times as necessary in the aggregation process. Lists of objects sift through the pipeline

```
// Get the top 10 most expensive computer brands, by average price
db.products.aggregate([
1    { $match: { name: "Computer" } },
2    { $group: { _id: "$brand", avgPrice: { $avg: "$price" } } },
3    { $sort: { avgPrice: -1 } },
4    { $limit: 10 }
]);
```

DOJO

Sujet

1. Design SQL requests for each question in pgAdmin
2. Implement these requests in the python backend

THE END

See you next week ;-)