

NoSQL databases

Francieli ZANON BOITO

Goal of this class

- To understand the motivations behind NoSQL ("Not only SQL") systems
- An overview of different solutions
- NOT a manual to learn specific NoSQL databases
 - Too many of them
 - For a comprehensive list: <http://nosql-database.org/>
 - Next class and the lab activity: Neo4j

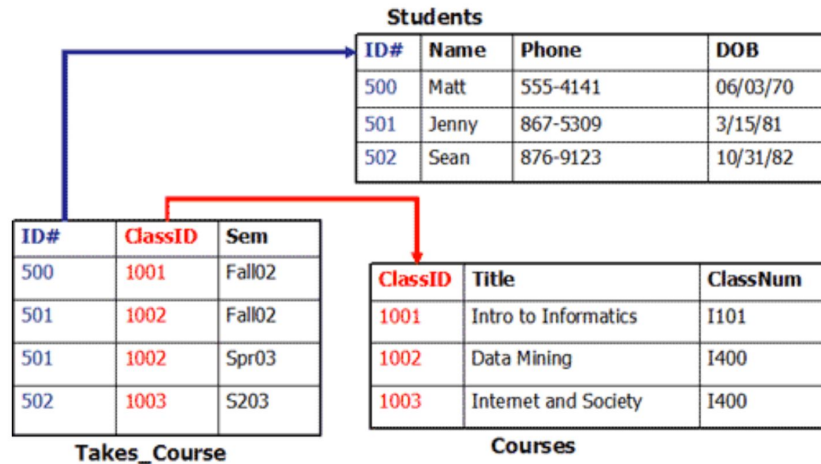
"Traditional" applications

- Months of planning and development
 - Including the schema for the relational database (MySQL, Oracle, PostgreSQL, ...)
- Structured data
- Its scale is known in advance
- Configuration for the servers is chosen accordingly
- Scale-up

Relational databases

- Data organized as tables
 - Row = record, Column = attribute
- Relations between tables
 - Integrity constraints

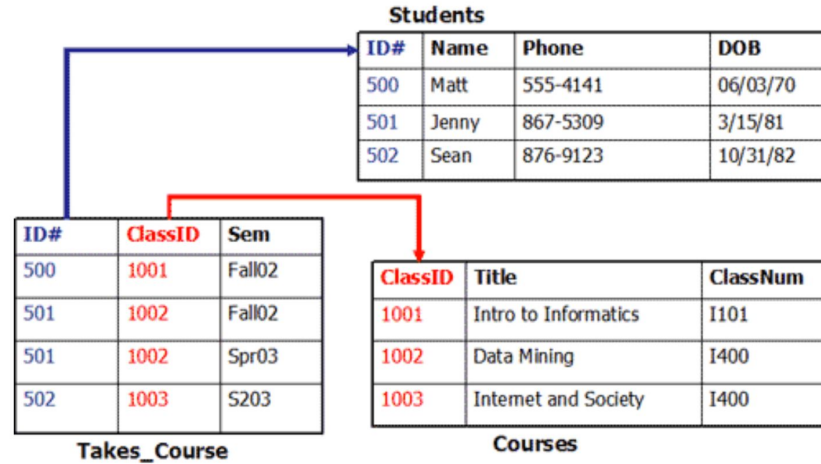
Select title from courses natural join takes_courses group by ClassID having count(*) > 10



The big data era

- Agile development
 - Frequent release of new features, possibly changing the data model
- Data structure can be unknown or variable
- Large amounts of data, thousands to millions of users
- Need to scale-out
- Cloud-based

Select title from courses natural join takes_courses group by ClassID having count(*) > 10



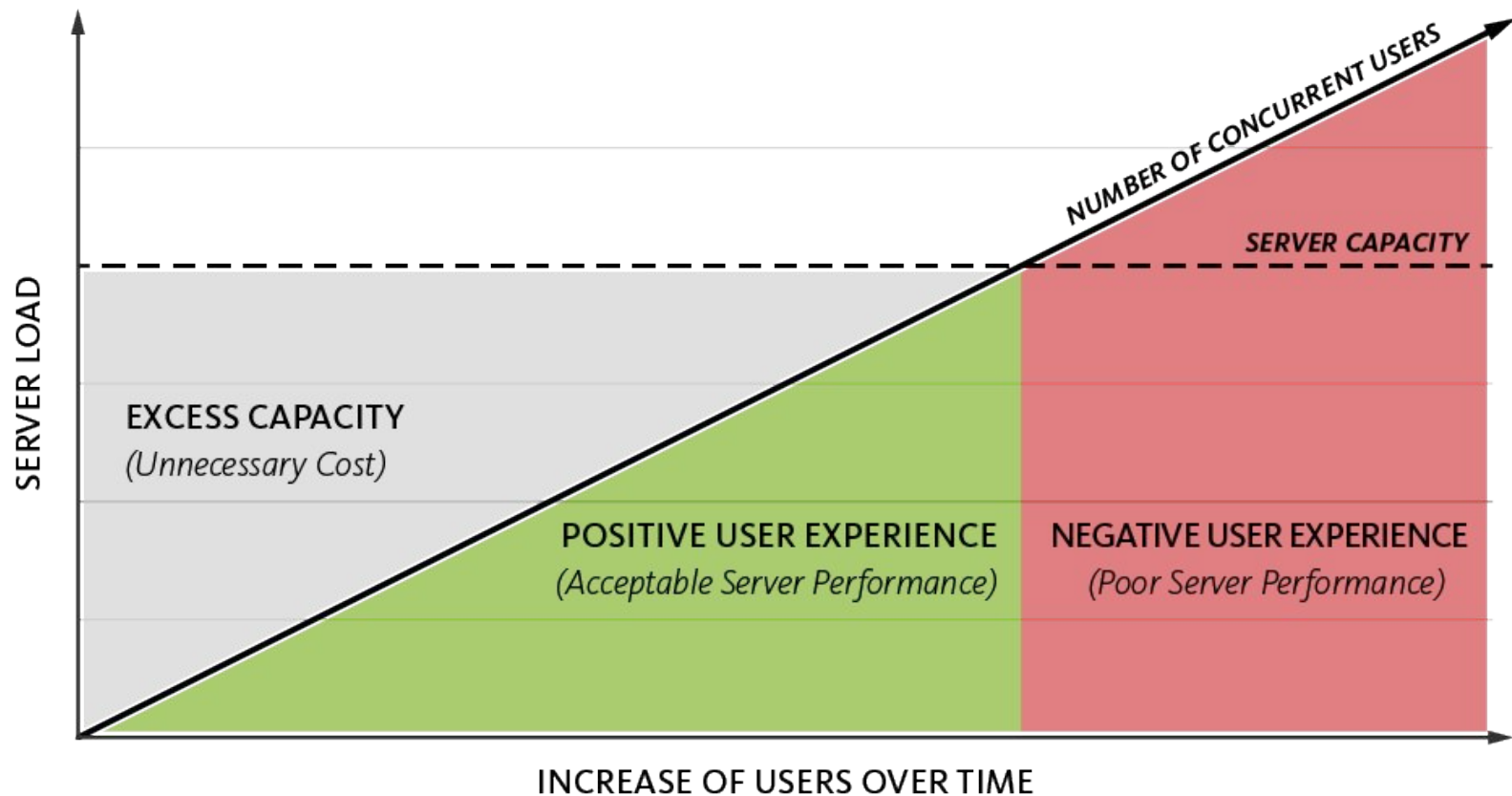


Figure from <https://www.couchbase.com/resources/why-nosql>

SQL relational databases	NoSQL databases
Data is organized in tables	Data is organized in key-value pairs, sparse columns, documents, or graphs
Pre-defined schema	Less rigid formats, documents can have different fields, add as you go
ACID	

ACID properties

- **A**tomicity
 - Transaction are all or nothing (e.g. when adding a bi-directional friendship relation, it's added both ways or not at all)
- **C**onsistency
 - Only valid data written (e.g. cannot say a student takes a course that is not in the courses table)
- **I**solation
 - When multiple transactions execute simultaneously, they appear as if they were executed sequentially (aka serializability)
- **D**urability
 - When data has been written and validated, it is permanent (i.e. no data loss, even in the case of some failures)

→ Easy life for the developer

SQL relational databases	NoSQL databases
Data is organized in tables	Data is organized in key-value pairs, sparse columns, documents, or graphs
Pre-defined schema	Less rigid formats, documents can have different fields, add as you go
ACID	Looser consistency models

CAP theorem (Brewer's theorem)

- **Consistency:** every node returns the same, most recent, successful write (sequential consistency)
 - **Availability:** every non-failed node answer all requests it receives
 - **Partition tolerance:** the system continues to work when network fails
-
- In a centralized system, no need for P, we have CA
 - In a distributed data store, P is essential
 - When the network fails, we need to choose between C and A

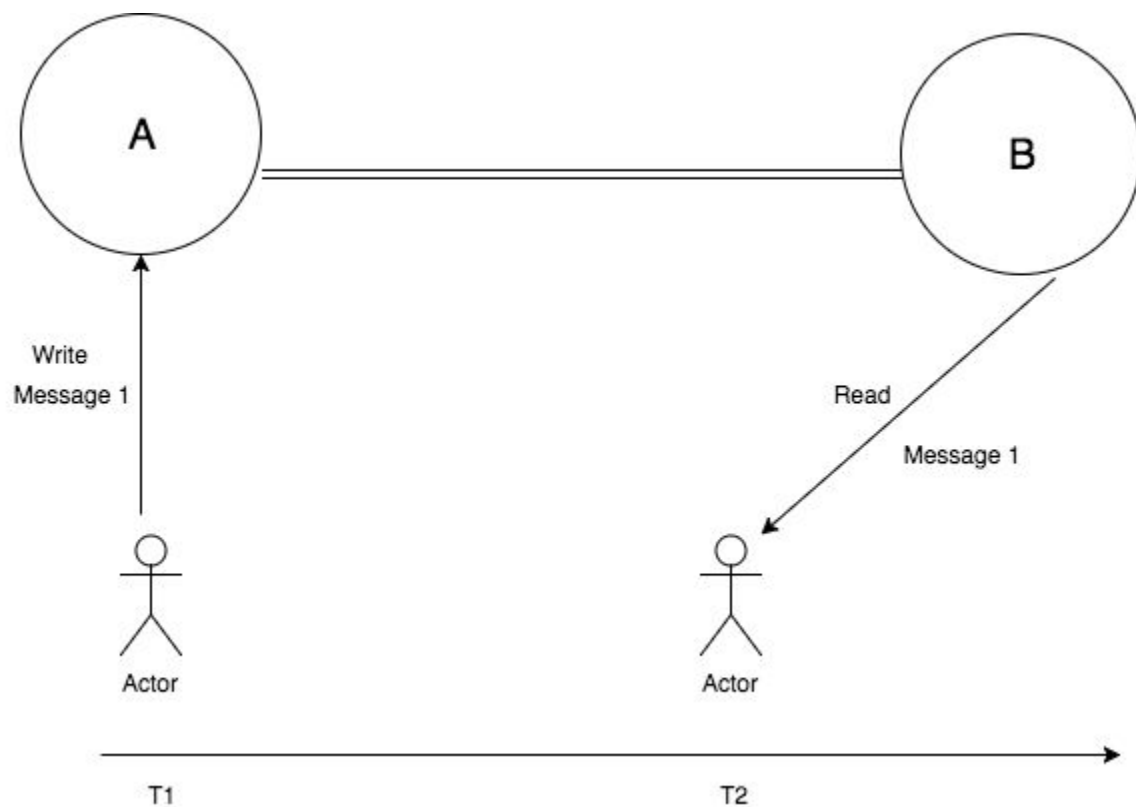


Figure from <https://shekhargulati.com/2018/08/08/week-2-cap-theorem-for-application-developers/>

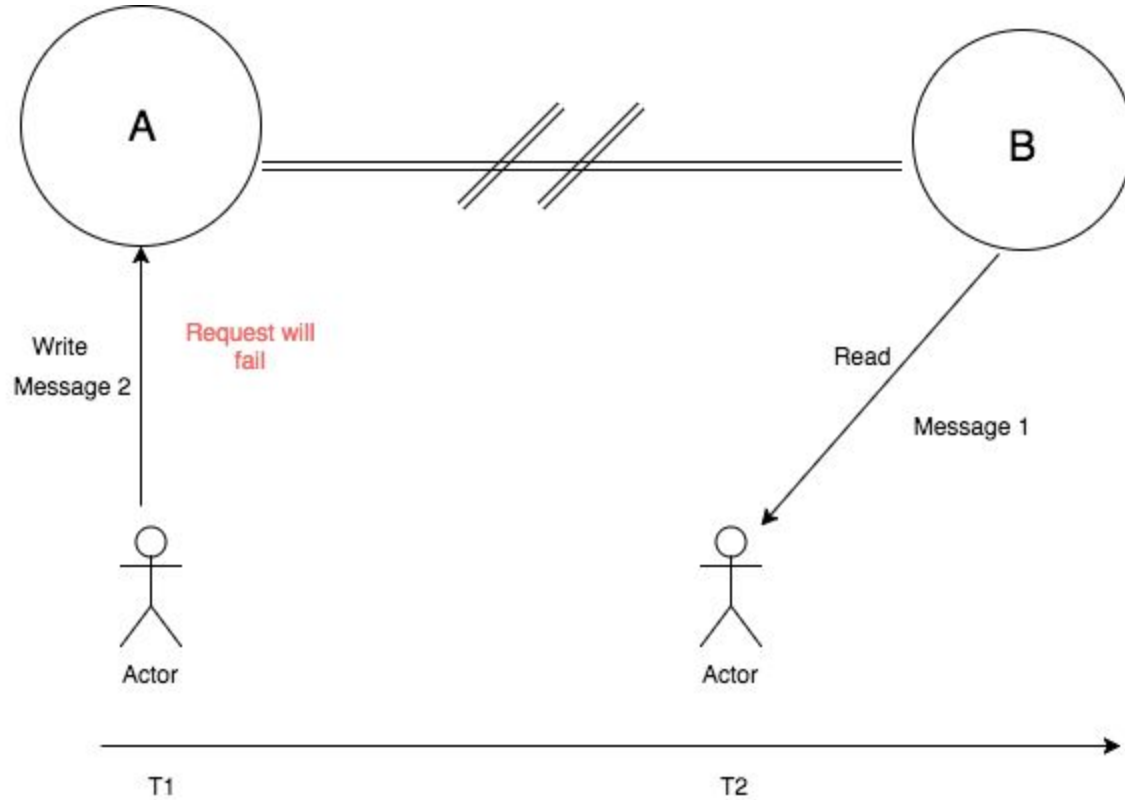


Figure from <https://shekhargulati.com/2018/08/08/week-2-cap-theorem-for-application-developers/>










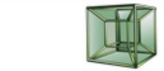



Weak consistency

- Eventual consistency
 - It will be consistent after some time, when there is no network partition
 - Sometimes we could be writing data that is going to be read only later
- Different levels of consistency
 - Causal consistency
 - Read-your-writes consistency
 - Etc
- What to choose? It depends on the application!
- Some databases are not updated very often

SQL relational databases	NoSQL databases
Data is organized in tables	Data is organized in key-value pairs, sparse columns, documents, or graphs
Pre-defined schema	Less rigid formats, documents can have different fields, add as you go
ACID	Looser consistency models
40-year-old standard (from the 70s)	First papers in 2006 and 2007
SQL query language	Diverse query APIs, it can be difficult to migrate between solutions
Query to access small subsets of the data	We often want to process ALL data

SQL or NoSQL?

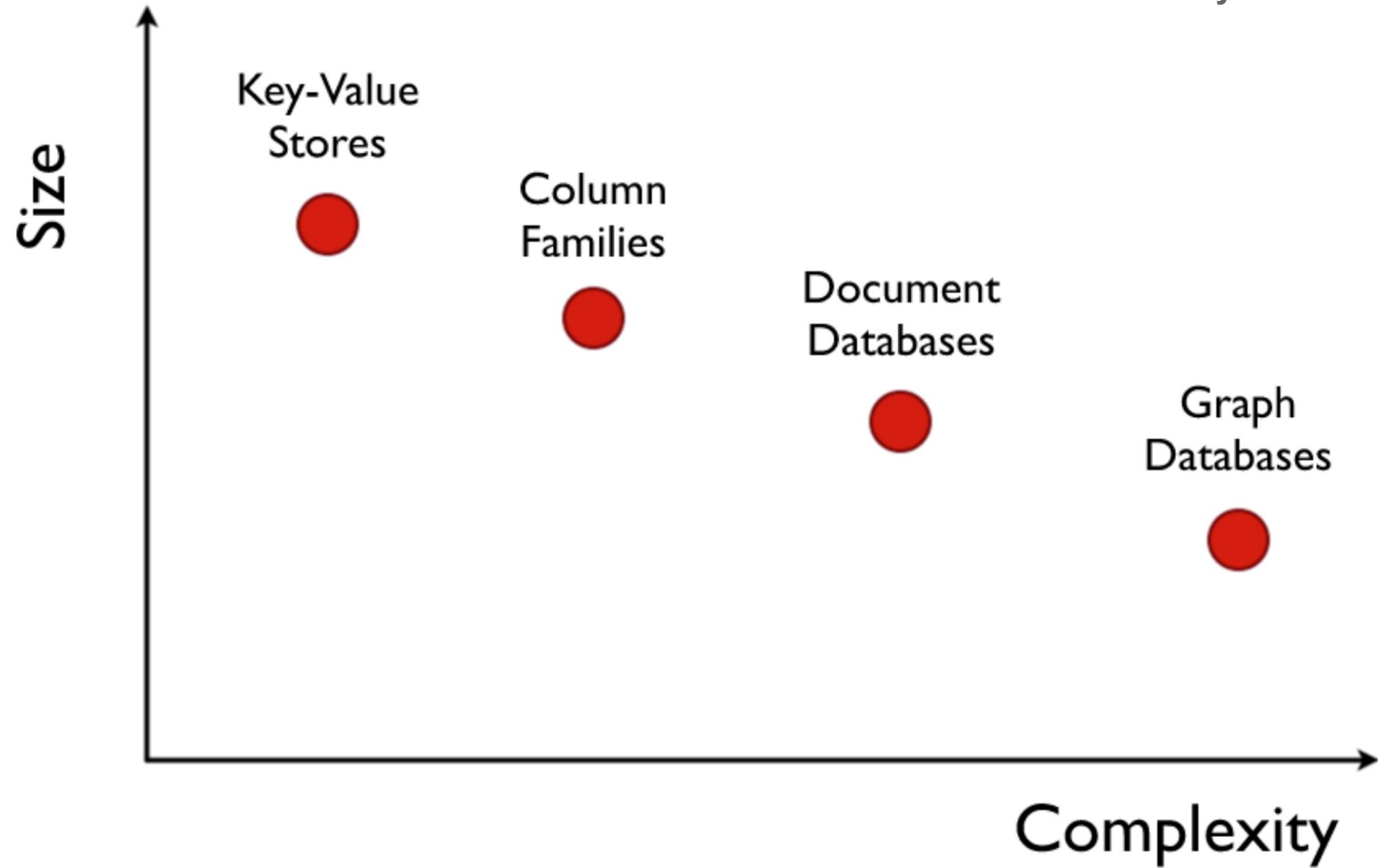
- It depends on the application!
- Snapshot stories use Amazon DynamoDB *
- Facebook and Netflix use/used Apache Cassandra
- Ryanair uses Couchbase for their mobile app (over 3 million users) **

Document Database	Graph Databases
  	 
Wide Column Stores	Key-Value Databases
   	   

@cloudbt <http://www.aryannava.com>

* https://www.youtube.com/watch?v=WUleQzu9l_8

** <https://www.couchbase.com/customers/ryanair>



Key-value store

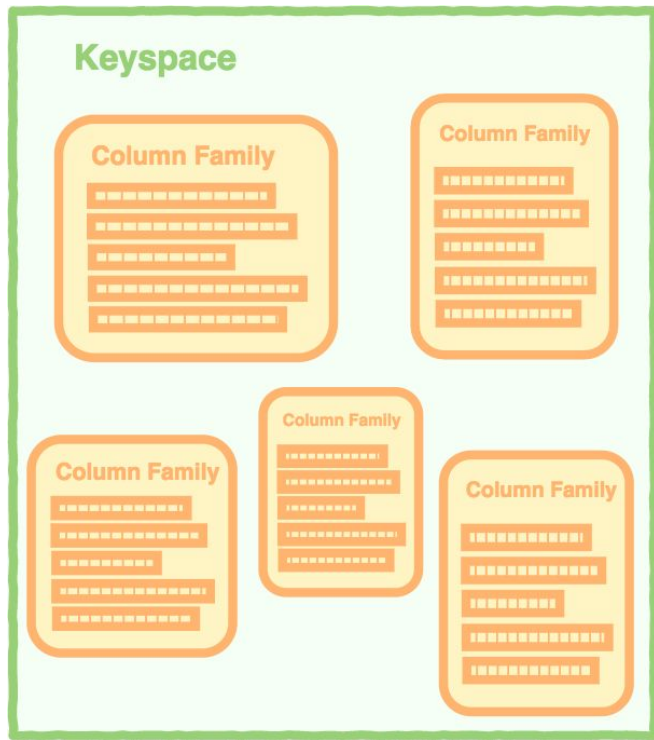
- Data in < key, value > pairs
- Two basic operations (similar to data structures like hashMap and dictionaries)
 - Put(K,V)
 - Get(K)
- Can be used to cache information in memory
- Recent research: accelerate it with hardware



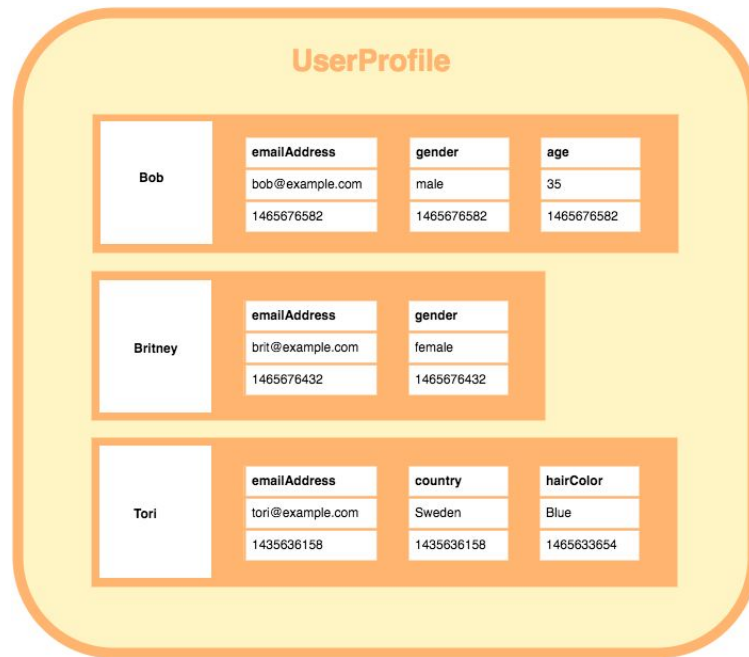
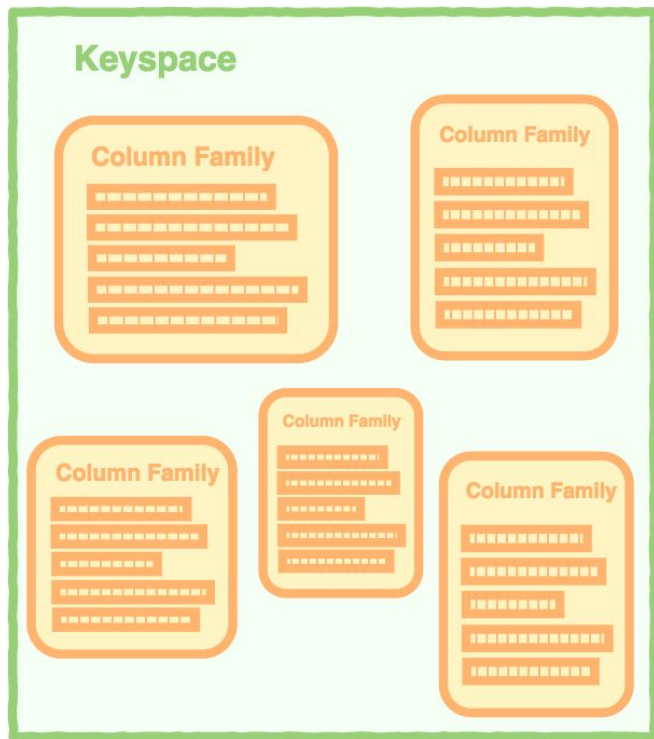
Wide Column/Tabular DB

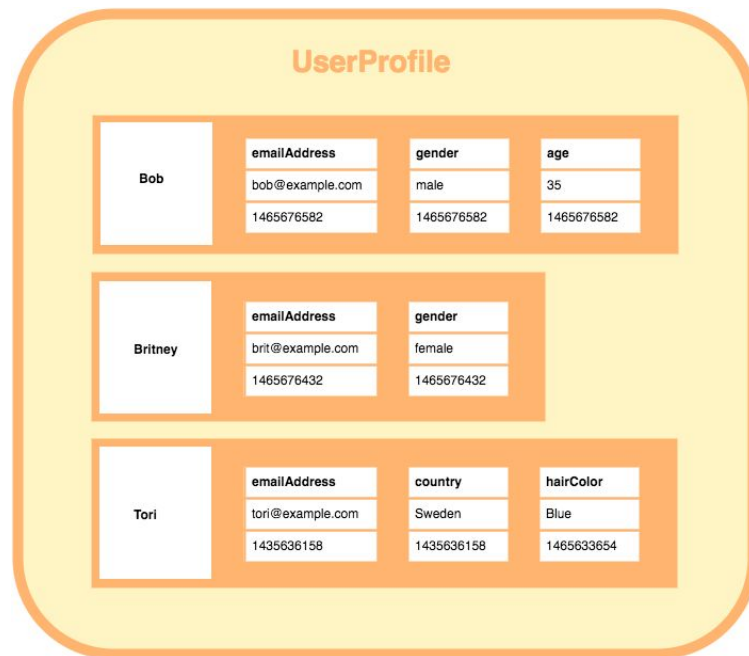
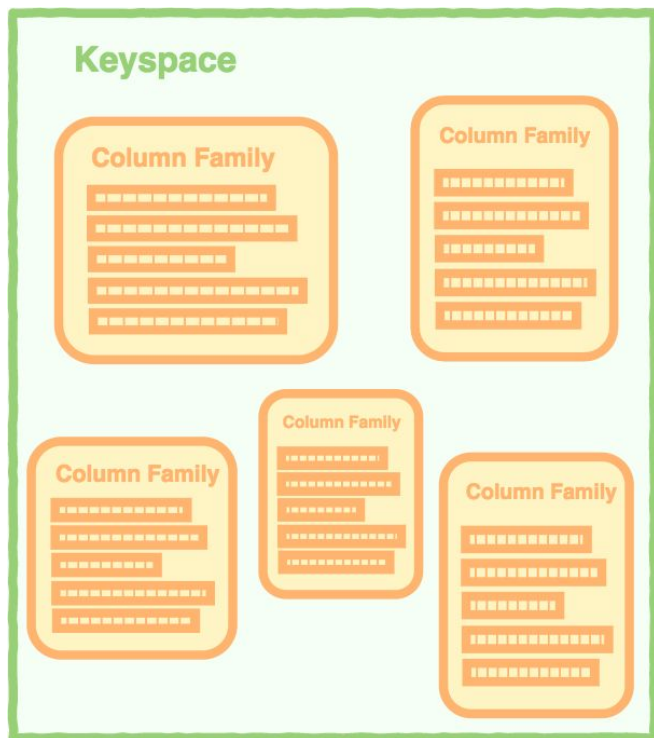
- Data is organized in rows with a primary key
- Stored in a distributed sparse multidimensional sorted map
- Data is retrieved by key per column family



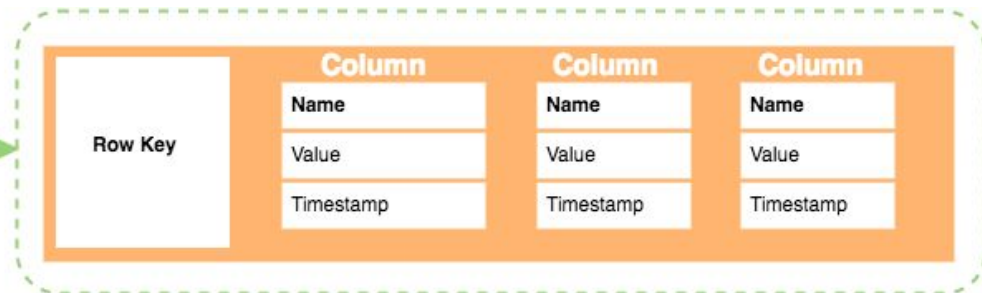


Figures from <https://database.guide/what-is-a-column-store-database/>





Row →



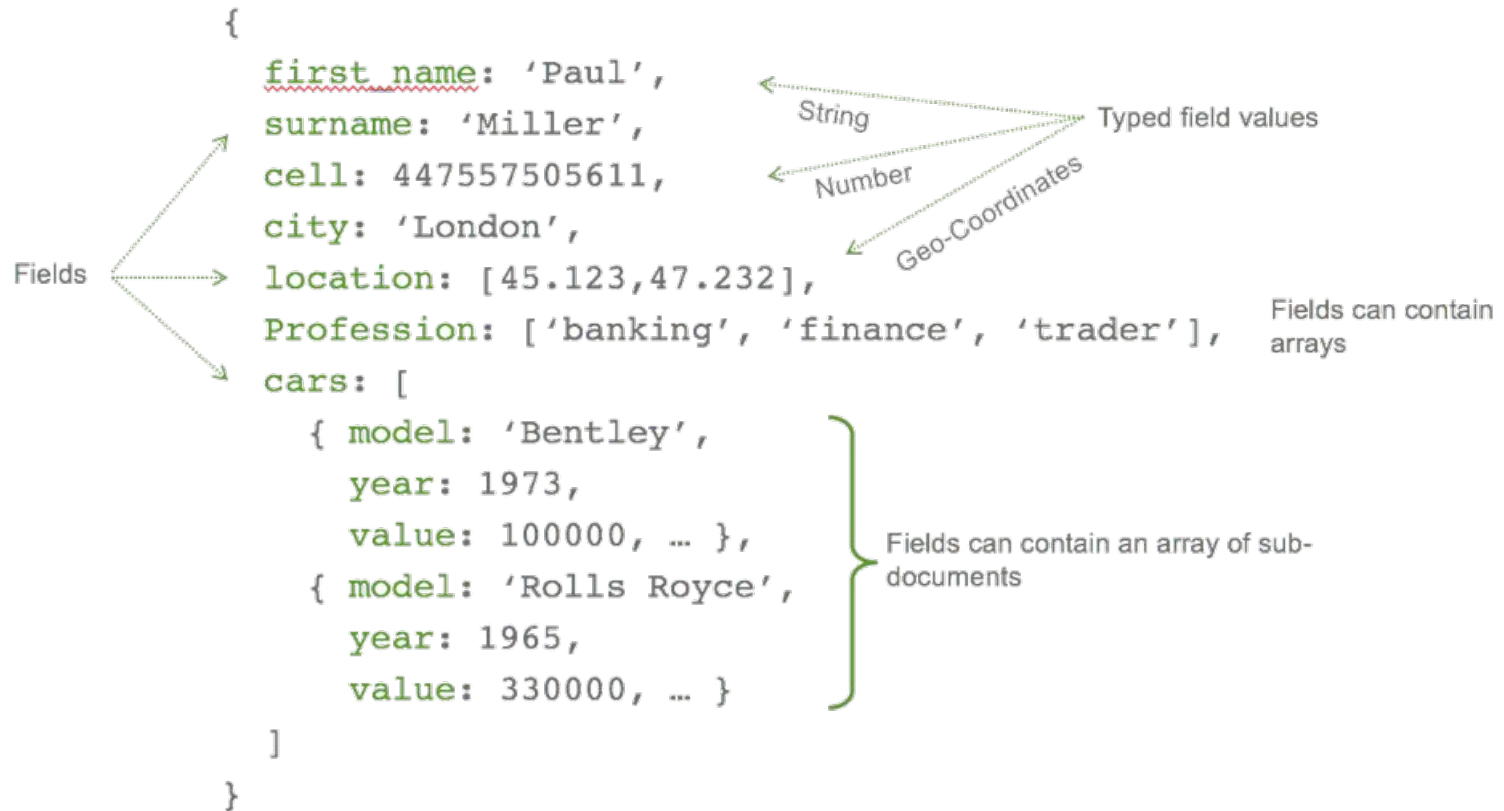
When to use them?

- Key-value and column DB achieve good performance performance
 - Access pattern is simple and the format is opaque -> lots of optimization opportunities
 - Column family DB is good for aggregation queries (average, sum, etc)
- Applications that only query data by a single or a limited range of key

Document DB

- Data stored as documents (often JSON)
 - A document has many fields and their values
 - Documents can be nested
 - They can have different fields
- Queries can be done over any field
- Documents are closely aligned with object-oriented programming
- Performance advantage: instead of having to combine data from multiple tables, everything about an object is in the same document





Graph DB

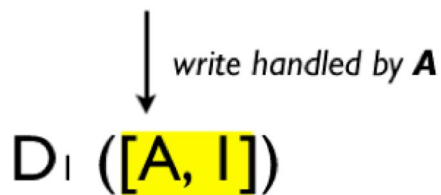
- Data is represented by a graph
 - Nodes and relationships have properties as $\langle \text{key}, \text{value} \rangle$
- Useful when traversing relationships is important
 - For instance: social networks, supply chains, etc
- Can be inefficient for other operations
 - Often coupled with another db to store properties





Vector Clocks

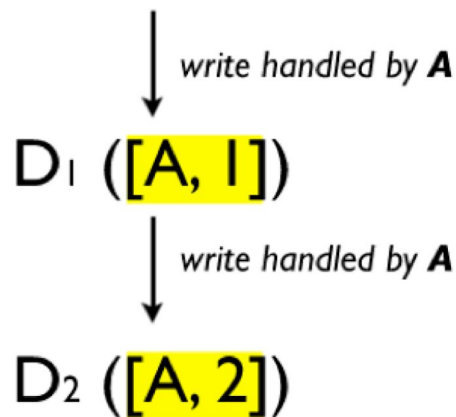
- Classic algorithm for partial ordering of events in distributed systems (from 1988)
- Each process has a vector with clocks for all processes
 - Every internal event, it increases its own clock
 - Every message sent, it increases its own clock and sends the whole vector
 - Every message received, it increases its own clock and merges the vectors (by taking the maximum)



Causality-based partial order over events that happen in the system.

Document version history: a counter for each node that updated the document.

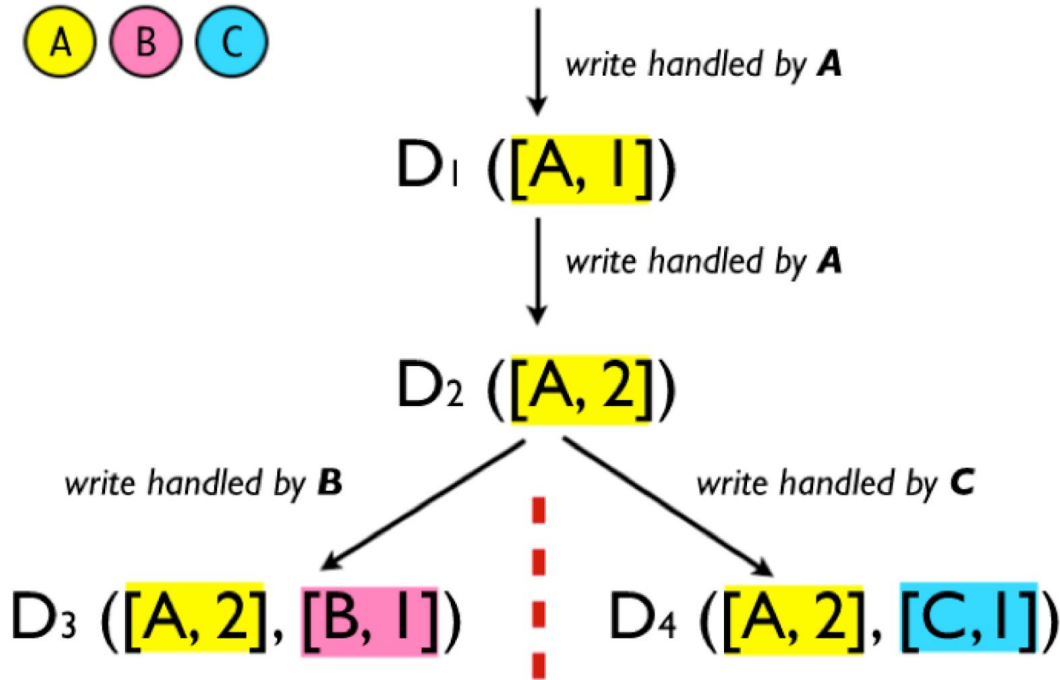
If all update counters in V_1 are smaller or equal to all update counters in V_2 , then V_1 precedes V_2 .



Causality-based partial order over events that happen in the system.

Document version history: a counter for each node that updated the document.

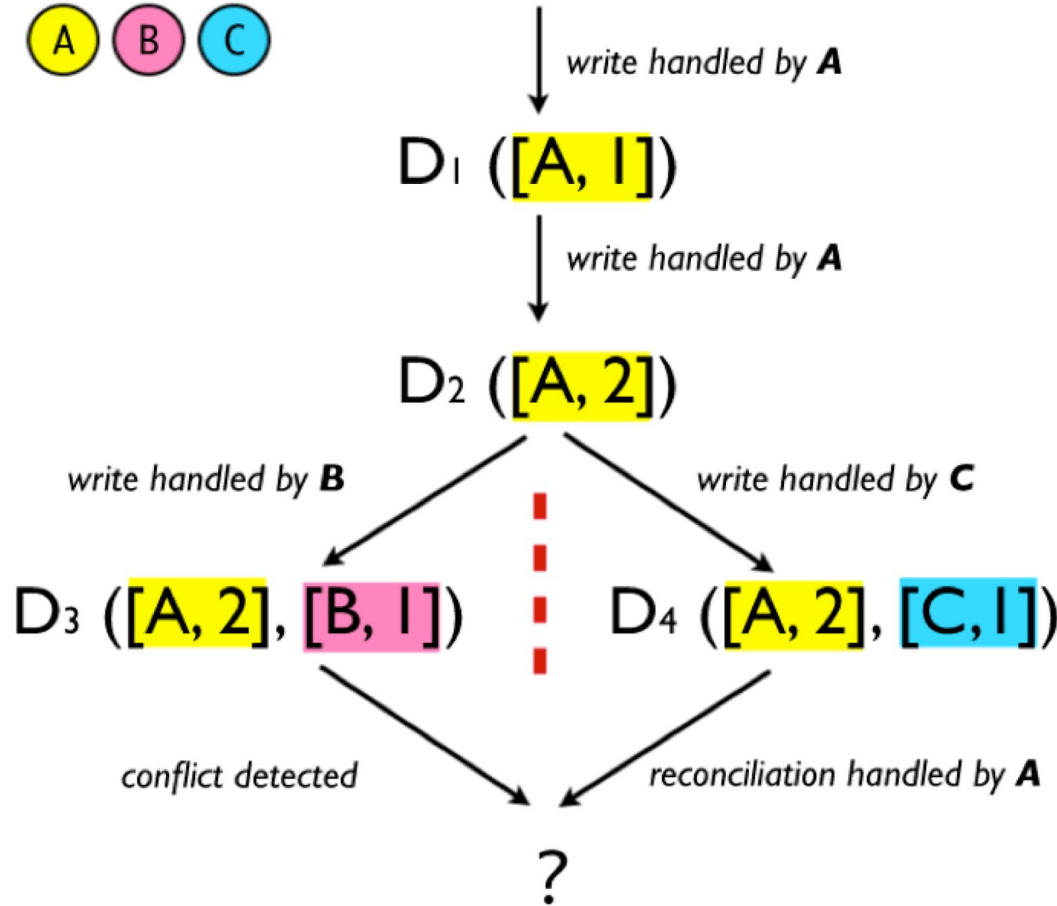
If all update counters in V_1 are smaller or equal to all update counters in V_2 , then V_1 precedes V_2 .



Causality-based partial order over events that happen in the system.

Document version history: a counter for each node that updated the document.

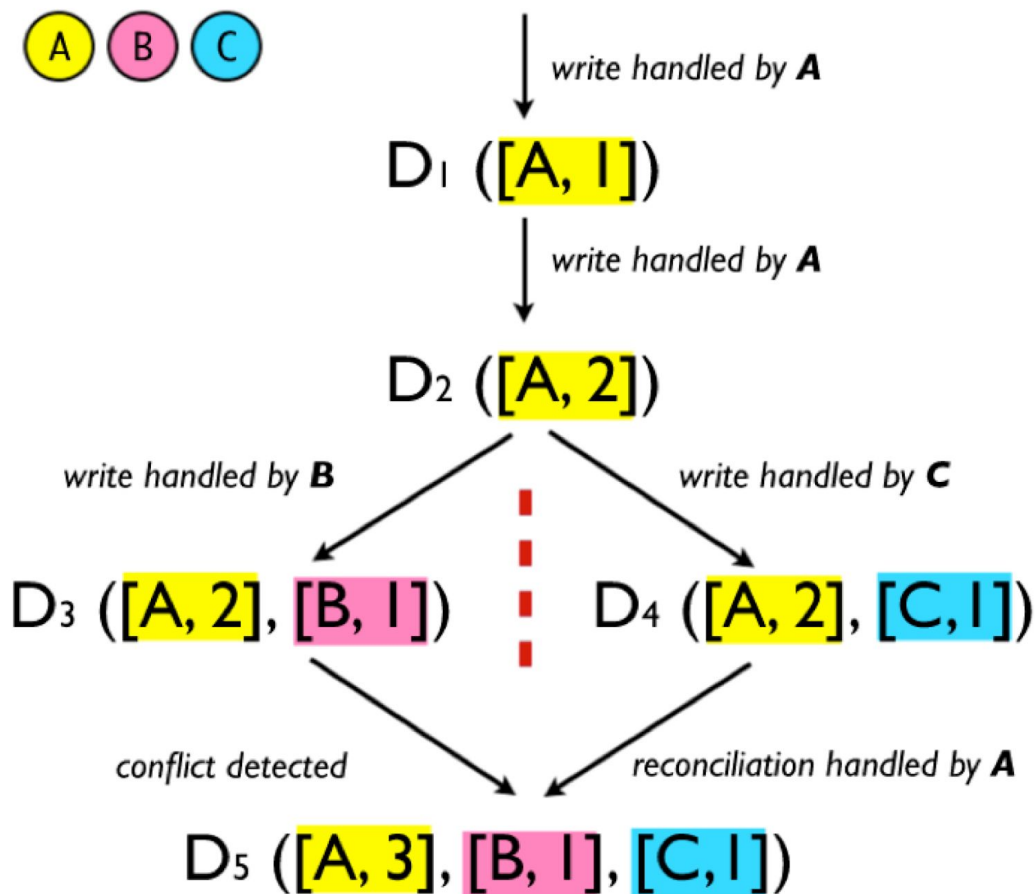
If all update counters in V_1 are smaller or equal to all update counters in V_2 , then V_1 precedes V_2 .



Causality-based partial order over events that happen in the system.

Document version history: a counter for each node that updated the document.

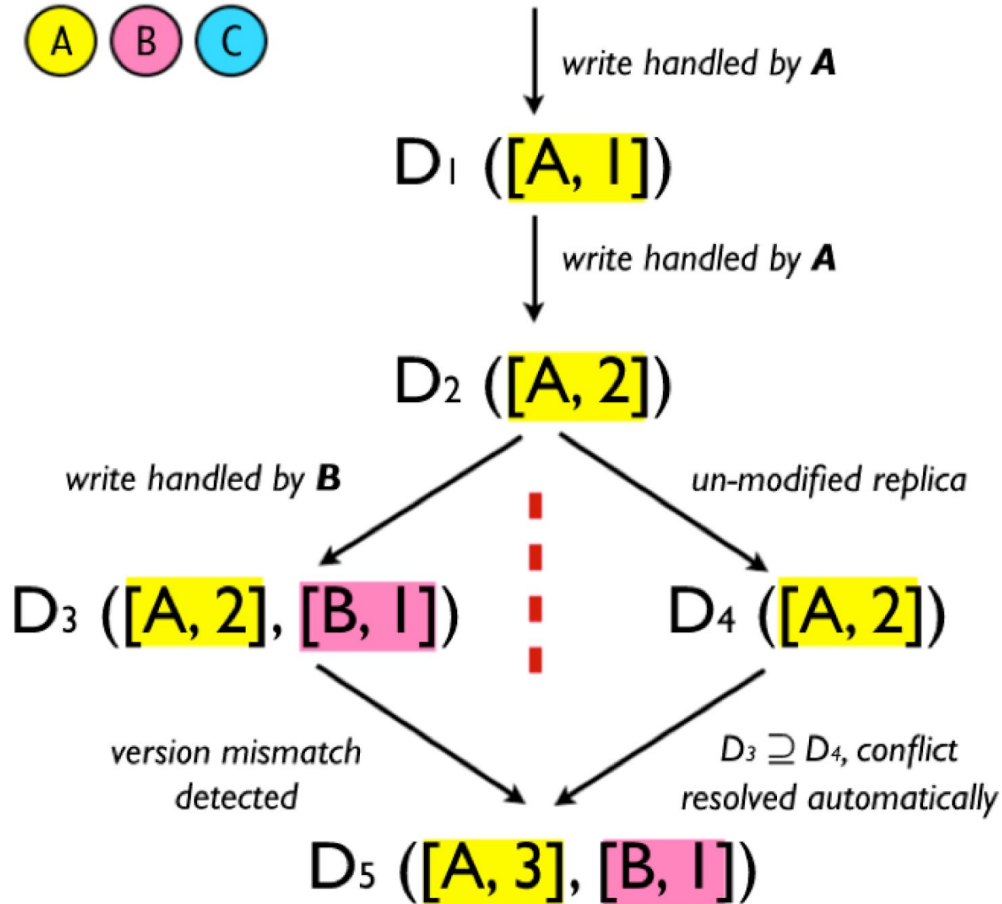
If all update counters in V_1 are smaller or equal to all update counters in V_2 , then V_1 precedes V_2 .



Causality-based partial order over events that happen in the system.

Document version history: a counter for each node that updated the document.

If all update counters in V_1 are smaller or equal to all update counters in V_2 , then V_1 precedes V_2 .



Vector Clocks can detect a conflict. The conflict resolution is left to the application or the user.

The application *might* resolve conflicts by checking relative timestamps, or with other strategies (like merging the changes).

Vector clocks can grow quite large (!)

Reading

- For next class:
 - G. DeCandia et al. "Dynamo: amazon's highly available key-value store"
 - F. Chang et al. "BigTable: A distributed storage system for structured data"
- Illustrated proof of the CAP theorem:
https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/
- Extra:
 - <https://www.mongodb.com/nosql-explained>
 - <https://www.couchbase.com/resources/why-nosql>
 - <http://nosql-database.org/>