DMIF, University of Udine

# Introduction To NoSQL

Andrea Brunello
andrea.brunello@uniud.it

May, 2022

# Outline

# What is NoSQL ?

The term NoSQL (Not only SQL) refers to data stores that are **not relational databases**, rather than explicitly describing what they are.

A possible (rather general) definition: *"Next Generation DBs mostly addressing some of the points: being non-relational, distributed, open source and horizontally scalable"*.

NoSQL storage technologies have very **heterogeneous** operational, functional, and architectural characteristics.

NoSQL proposals have been developed starting from 2009 trying to address new challenges that emerged in that decade.

# The Rise of NoSQL - 1

Most enterprise level applications ran on top of a relational databases (MySQL, PostgreSQL, etc).

Over the years data got bigger in volume, started to change more rapidly, and to be in general more structurally varied than those commonly handled with traditional RDBMS.

As the *volume* of data increases dramatically, so does query execution time, as a consequence of the size of tables involved and of an increased number of join operations (*join pain*).

# The Rise of NoSQL - 2

Data *velocity* is rarely a static metric. Internal and external changes to a system and to the context in which it operates can have a considerable impact on the rate at which data have to be managed.

Variable velocity, coupled with large volume, requires data stores to handle sustained levels of high read and write loads, and also peaks.

Data is nowadays far more varied than the data typically managed in the relational world.

*Variety* can be defined as the degree to which data is regularly or irregularly structured, dense or sparse, connected or disconnected.

# Different kinds of data

We broadly distinguish between the following kinds of data:

- **Structured** data
- **Semi-structured** data
- **Unstructured** data

# Structured data

Structured data can be seen as tabular data, represented by **rows** and **columns**:

- Each row belonging to a same table has a fixed format, think about an Excel file
- For instance, personal data regarding customers
- Easy to store and process by means of relational DBMSs

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| 1 | Name | Surname | Birth date | Gender | Address | Phone number |
| 2 | John | Doe | 12/07/74 | M | 660 North Gonzales Ave. Canyon City, CA 91387 | 202-555-0123 |
| 3 | Mary | Jane | 15/08/90 | F | 7772 Shore St. Zion, IL 60099 | 202-555-0176 |
| 4 | Darell | Smart | 07/03/83 | M | 2 Pine Ave. Royal Oak, MI 48067 | 202-555-0197 |
| 5 | Jessica | Miller | 03/11/95 | F | 7015 Wilson St. South Portland, ME 04106 | 202-555-0197 |
| 6 | Belinda | Barton | 05/12/67 | F | 22 Wakehurst Street Jackson, NJ 08527 | 202-555-0197 |
| 7 | Ned | Fitzgerald | 26/10/77 | M | 12 Sage Rd. Hope Mills, NC 28348 | 202-555-0197 |

# Semi-structured data

Semi-structured data do not have a fixed format, but still have some structuring:

- They contain **tags** or other markers to separate semantic elements and enforce hierarchies of fields within the data
- For example, this can be the case of closed form answers given to telephone surveys

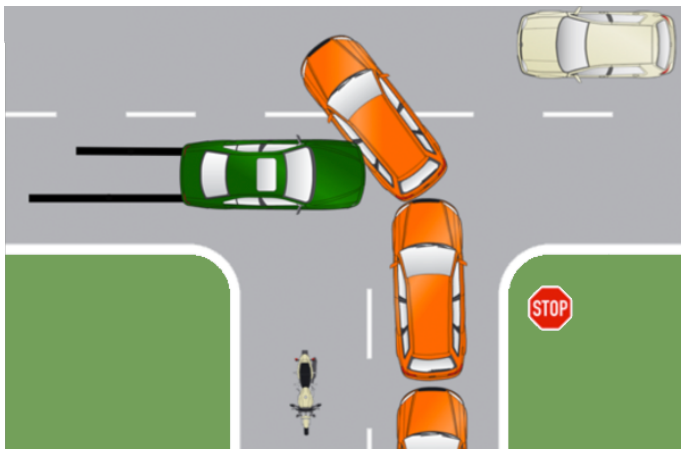
```
<Phone campaign>
    <Survey ID = 1>
        <Name> John Easter </Name>
        <Question ID = 1> yes </Question>
        <Question ID = 2> yes </Question>
        <Question ID = 2.1> 5 </Question>
        <Question ID = 2.2> 7 </Question>
        <Question ID = 3> no </Question>
    </Survey>
    <Survey ID = 2>
        <Name> Mary Christmas </Name>
        <Question ID = 1> yes </Question>
        <Question ID = 2> no </Question>
        <Question ID = 3> yes </Question>
        <Question ID = 3.1> yes </Question>
        <Note> The person seems to be rather interested in the offered product </Note>
    </Survey>
</Phone campaign>
```

# Unstructured data

Unstructured data is **not organized** in any predefined manner

- Free text, e.g., natural language descriptions of accidents
- Audio and visual materials
- Difficult to store, index, and analyse

> *"Vehicle A crossed the road failing to give way to vehicle B, which subsequently hit it on its left side."*



# Benefits of NoSQL

Elastic Scaling:

- RDBMS **scale up**: bigger load $\Rightarrow$ bigger server
- NoSQL **scale out**: distribute data across multiple nodes, adding/removing them seamlessly

DBA Specialists:

- RDBMS require highly trained experts to monitor DB
- NoSQL require less management: automatic repair and distribution and simpler data models

Huge increase in size and heterogeneity of data:

- RDBMS: capacity and constraints at their limits
- NoSQL systems are specifically designed to cope with that

# Drawbacks of NoSQL

Several default constraints of RDBMS are not supported at the database level (e.g., foreign keys).

If not properly managed, the absence of a fixed structure may become an issue.

The design process is not as straightforward and consolidated as in the relational model.

Lack of SQL (though there are some SQL-inspired languages).

# ACID versus BASE

Many people that encounter NoSQL are already familiar with relational databases.

In addition to clear differences in data and query models, also the consistency models used by NoSQL stores can be quite different from those employed by relational databases.

Many NoSQL databases use different consistency models to support the differences in volume, velocity, and variety of data discussed earlier.

# ACID Properties

*Atomicity* All operations in a transaction succeed, or every operation is rolled back (like nothing happened).

*Consistency* On transaction completion, the database is moved from a consistent state to a different, but always consistent state (wrt the defined constraints).

*Isolation* Transactions do not interfere with one another. Contentious access to data is moderated by the database so that transactions appear to run sequentially.

*Durability* The results of a transaction are permanent, even in the presence of failures of the system. Results can be changed only by a subsequent successful transaction.

# BASE Properties

In order to better support the characteristics of the novel kinds of data, NoSQL systems rely on the BASE properties instead:

- *Basically available* The store appears to be accessible most of the time (for instance, tolerance to node failures)
- *Soft state* Stores do not need to be write-consistent, nor do different replicas have to be mutually consistent all the time
- *Eventual consistency* The system eventually exhibits consistency at some later point

Of course, this kind of relaxed consistency would be a problem, for instance, in a banking database. In other cases, it is perfectly acceptable, e.g., for social networks.

# The CAP Theorem

The CAP theorem (or Brewer's theorem) states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees (however, see caveat in the next slide):

- *Consistency*: every read receives the most recently written data or an error
- *Availability*: every request receives a (non-error) response, with no guarantee that it contains the most recently written data
- *Partition tolerance*: the system continues to operate despite of network failures (e.g., when a network partition failure happens)

# The CAP Theorem - 2

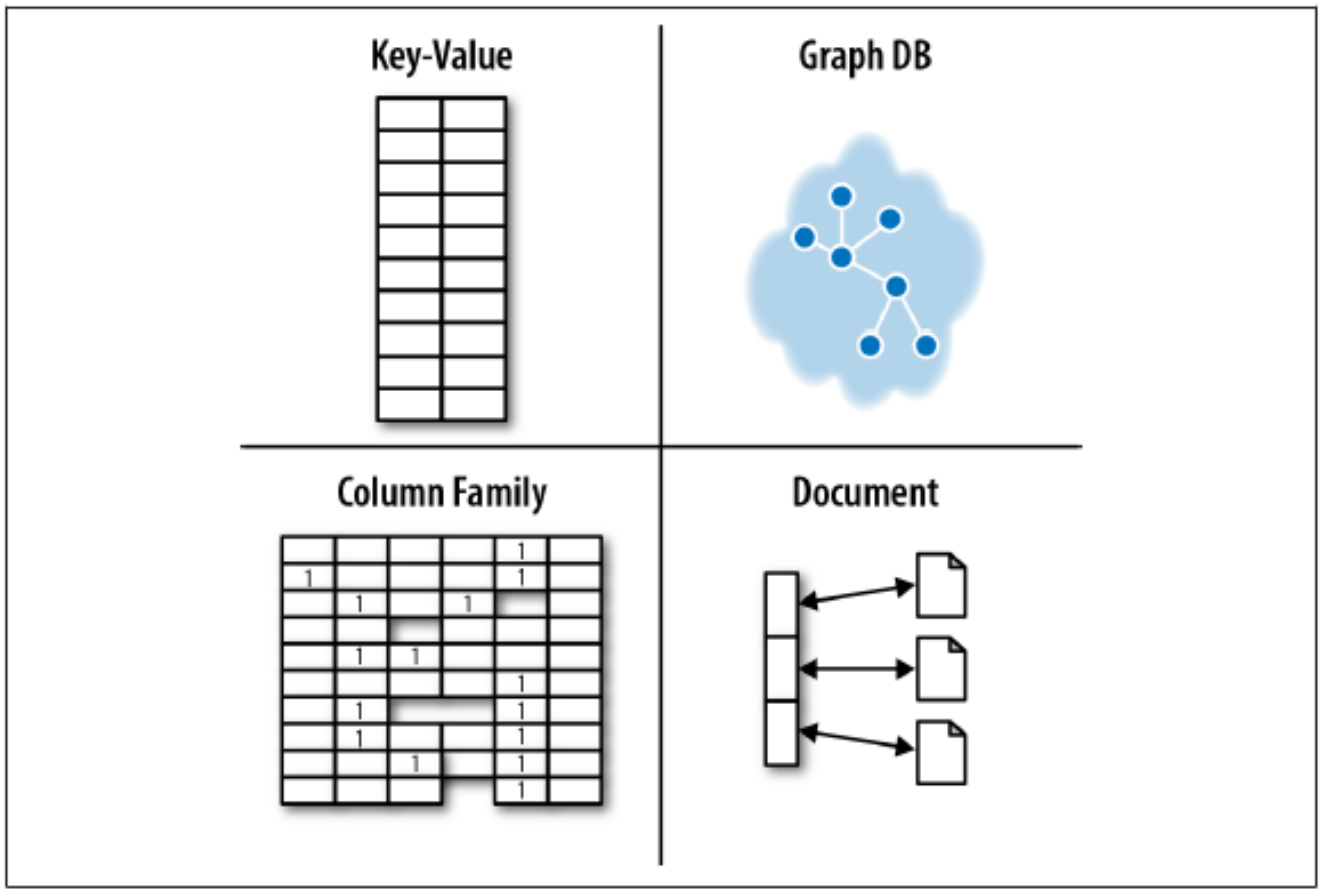
In fact, the choice is really between consistency and availability when a network partition failure happens; at all other times, no trade-off has to be made.

This means that, once a network partition happens, systems can behave in two different ways:

- *AP*: nodes are always online, but they may not get you the latest data; however, they sync whenever the lines are up
- *CP*: prevents data going out of sync, e.g., answering with an error to user interactions

In distributed databases, network partitions and node crashes can always happen. This means that partition tolerance cannot be neglected, thus there is actually no *AC* distributed system.

# The NoSQL Quadrant



# Key/Value Stores

# Key/Value Stores

Key/value stores are based on the concept of *associative array* (i.e., dictionary, or map), a data structure that contains couples of <key, values>; the key is used to access the values.

Associative arrays are implemented via *hash table* with a *hash function* on keys; this function tries to map keys across the available buckets where the values are stored, in a uniformly distributed way.

E.g., in a database storing employee data, there could be arrays <SSN, phone>, <SSN, dept_name>, <dept_name, budget>.

Operations allowed over an associative array are:

- *Add*: add an element to the array
- *Remove*: remove an element from the array
- *Modify*: change the value associated to a key
- *Find*: search for a value by the key

# Hash Table

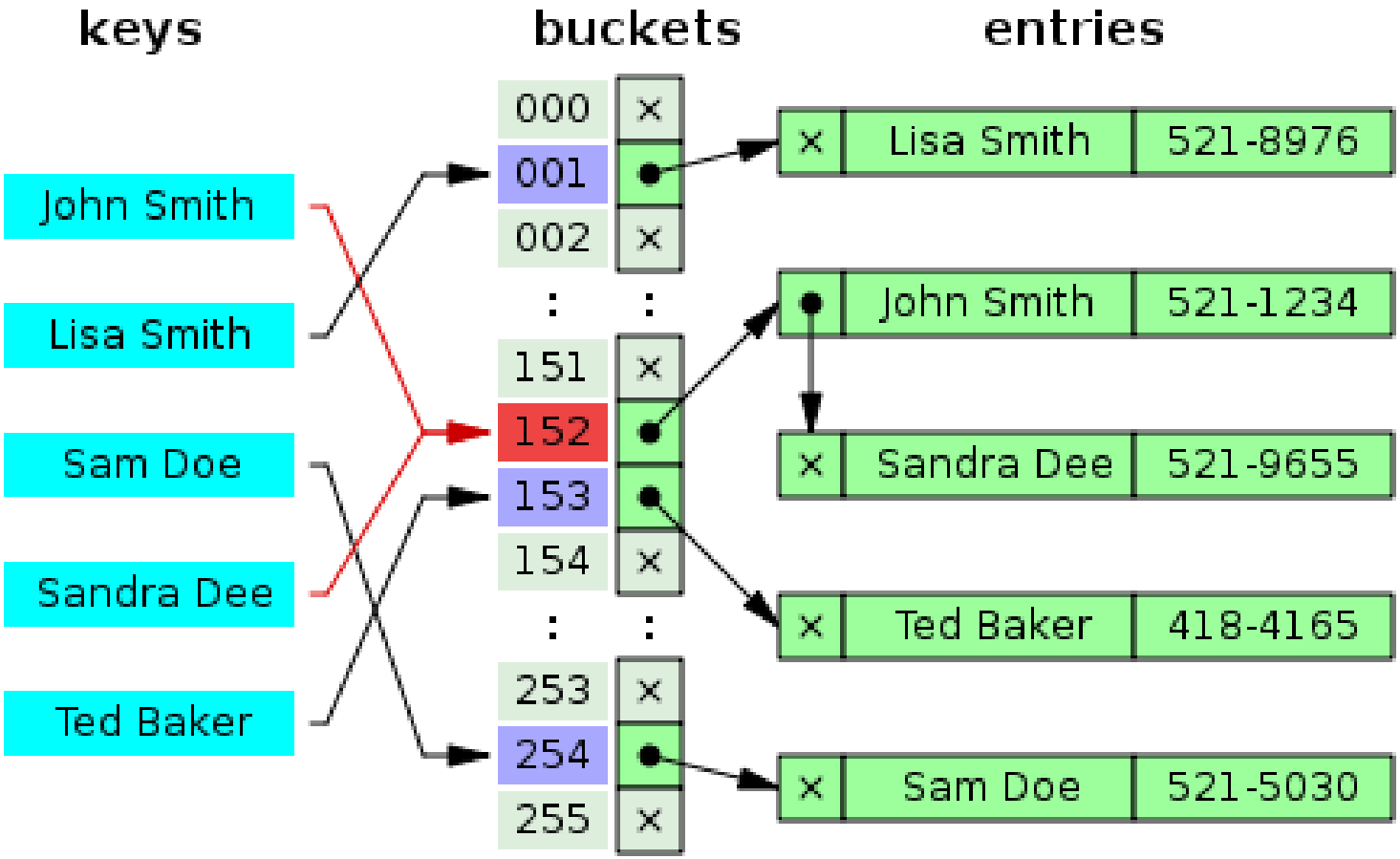
The hash table is a data structure that consists of an array of values and a hash function that maps any given key to a position inside the array, the latter indexed by a number.

The hash function transforms a string into a number, allowing to immediately find the value associated to the key.

The hash function must be deterministic in order to always return the same output with the same input.

There is also the need to handle *collisions* as different values of the key may lead to the same hash value (since the number of array locations is typically lower than the number of possible keys); a solution is given by *chaining*.

# Hashing - Example of Chaining



# Key/Value Store - Focus

Key/values stores offer just the add/remove/modify/find operations on **very simple data**, which nevertheless are executed at a fixed computational cost, thanks to the associative array.

Some typical relational operations, such as complex filters and JOINS are not possible.

Also, RDBMS functionalities like foreign keys are not supported.

Key/value stores are very simple pertaining to to their data model, but they can be extremely sophisticated regarding horizontal scalability.

# Key/Value Store - Examples

Use cases:

- Storage of profiles, preferences and configurations
- Storage of multimedia objects

Exemplary DBMSs:

- **Berkeley DB** hi-performance key/value database from Oracle, with implementations in C, Java and XML/C++
- **Project Voldemort** is a (LinkedIn-born) distributed key/value store based on Berkeley DB, designed for performance, horizontal scalability and fault protection, through data partitioning and replication
- **Redis**: `https://redis.io/`
- **Aerospike**: `https://aerospike.com/`

# Document-oriented Databases

# Document-oriented Databases

Document databases store, retrieve and manage document oriented information, also known as semi-structured data.

Documents do not have a fixed structure, but nonetheless contain **tags** or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. Therefore, they can be considered as a kind of **self-describing structure**.

Documents can be encoded into formats like XML and JSON.

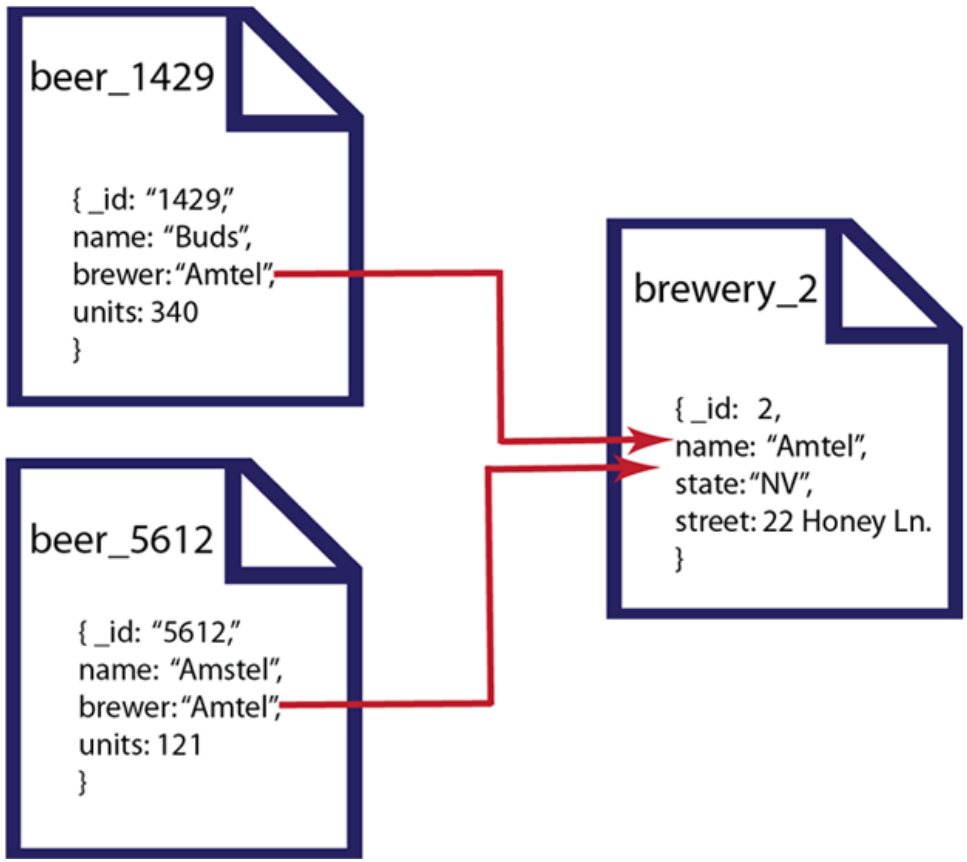
Document stores represent a step up with respect to key-value stores, defining a structure over keys and values, thus allowing the users to operate on the internal document structure.

# Document Example – 1

```
<Phone campaign>
    <Survey ID = 1>
        <Name> John Easter </Name>
        <Question ID = 1> yes </Question>
        <Question ID = 2> yes </Question>
        <Question ID = 2.1> 5 </Question>
        <Question ID = 2.2> 7 </Question>
        <Question ID = 3> no </Question>
    </Survey>
    <Survey ID = 2>
        <Name> Mary Christmas </Name>
        <Question ID = 1> yes </Question>
        <Question ID = 2> no </Question>
        <Question ID = 3> yes </Question>
        <Question ID = 3.1> yes </Question>
        <Note> The person seems to be rather interested in the offered product </Note>
    </Survey>
</Phone campaign>
```

# Document Example – 2

Red arrows typically implemented by means of a custom script. . . not naatively supported by the DBMS!



# Document-oriented Databases – Operations

The core operations that a document-oriented database supports for documents and are named as CRUD:

- *Creation*: of a new document
- *Retrieval*: based on key, content, or metadata
- *Update*: the content or metadata of the document
- *Deletion*: of a document

Each document in the database is uniquely identified by a **key**, which can be used to retrieve the document from the database. Indexes can be defined (on keys as well as document attributes) to speed up the operations.

Typical use cases: similar to key-value stores, but more complex data can be handled (product data management, inventory).

→ **MongoDB** is an example of a document-oriented database.

# Column-oriented Stores

# Column-oriented Stores

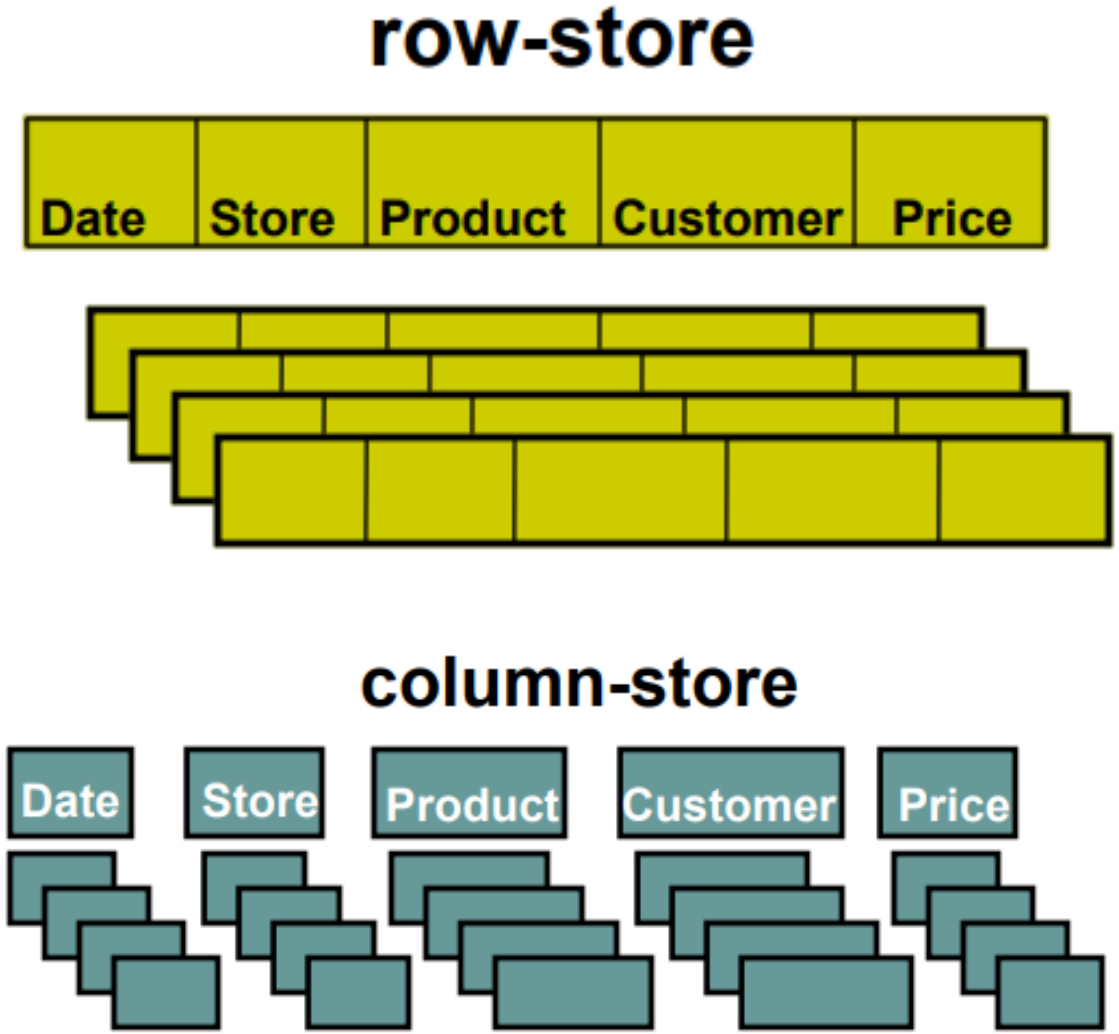
The roots of colum-store DBMSs can be traced back in the 1970s.

However, due to market needs and non favorable-technology trends it was not until the 2000s that they took off.

A column-oriented DBMS stores each database table column separately, in different disk locations, or even different machines.

Values belonging to the same column are packed together, as opposed to traditional DBMSs that store entire rows one after the other.

# Rows vs Columns



# Rows vs Columns – 2

In a traditional relational database each field is stored adjacent to the next in the same block on the hard drive:

```
512,Seabiscuit,Book,10.95,201712241200,goodreads.com
513,Bowler,Apparel,59.95,201712241200,google.com
514.Cuphead,Game,20.00,201712241201,gamerassaultweekly.com
```

In a columnar database, blocks on the disk for the above data might look like this:

```
512,513,514
Seabiscuit,Bowler,Cuphead
Book,Apparel,Game
10.95,59.95,20.00
201712241200,201712241200,201712241201
goodreads.com,google.com,gamerassaultweekly.com
```

# Column Store Pros

Column stores are beneficial when the typical application is reading a subset of columns, or performing **aggregate functions** over them (AVG, MIN, MAX, . . . ).

They offer efficient storing capabilities due to an easier **compression of data**, which is performed by column (columns have a low information entropy).

Also, they are good for storing sparse data (no need to store NULL values).

Columnar databases are **very scalable**, and well suited for *massively parallel processing* (MPP), which involves having data partitioned and spread across a large cluster of machines.

# Column Store Cons

The main drawbacks of column-based stores are related to write operations and tuple (re)construction.

Newly inserted tuples have to be **broken down** to their component attributes, and each attribute must be written separately.

Also tuple construction is considered problematic, since information about a logical entity is stored in multiple locations, which brings an **overhead** for queries that access many attributes from an entity.

→ **Cassandra** is an example of a column store database.

# Compression with column stores

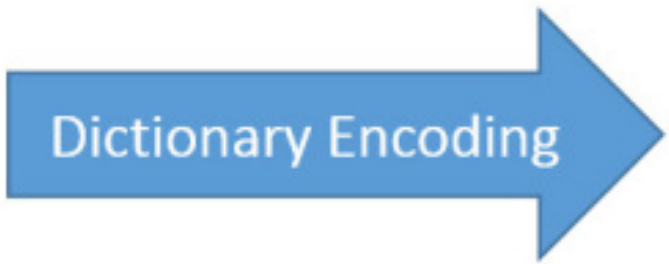
Column databases are very good at compressing data.

Compression also enhances reading performances, saving I/O costs, and since in many cases operations can be done directly, without decompression (e.g., SUM on data compressed with run-length encoding).

# Dictionary Encoding

Replacing datatypes with dictionary and indexes

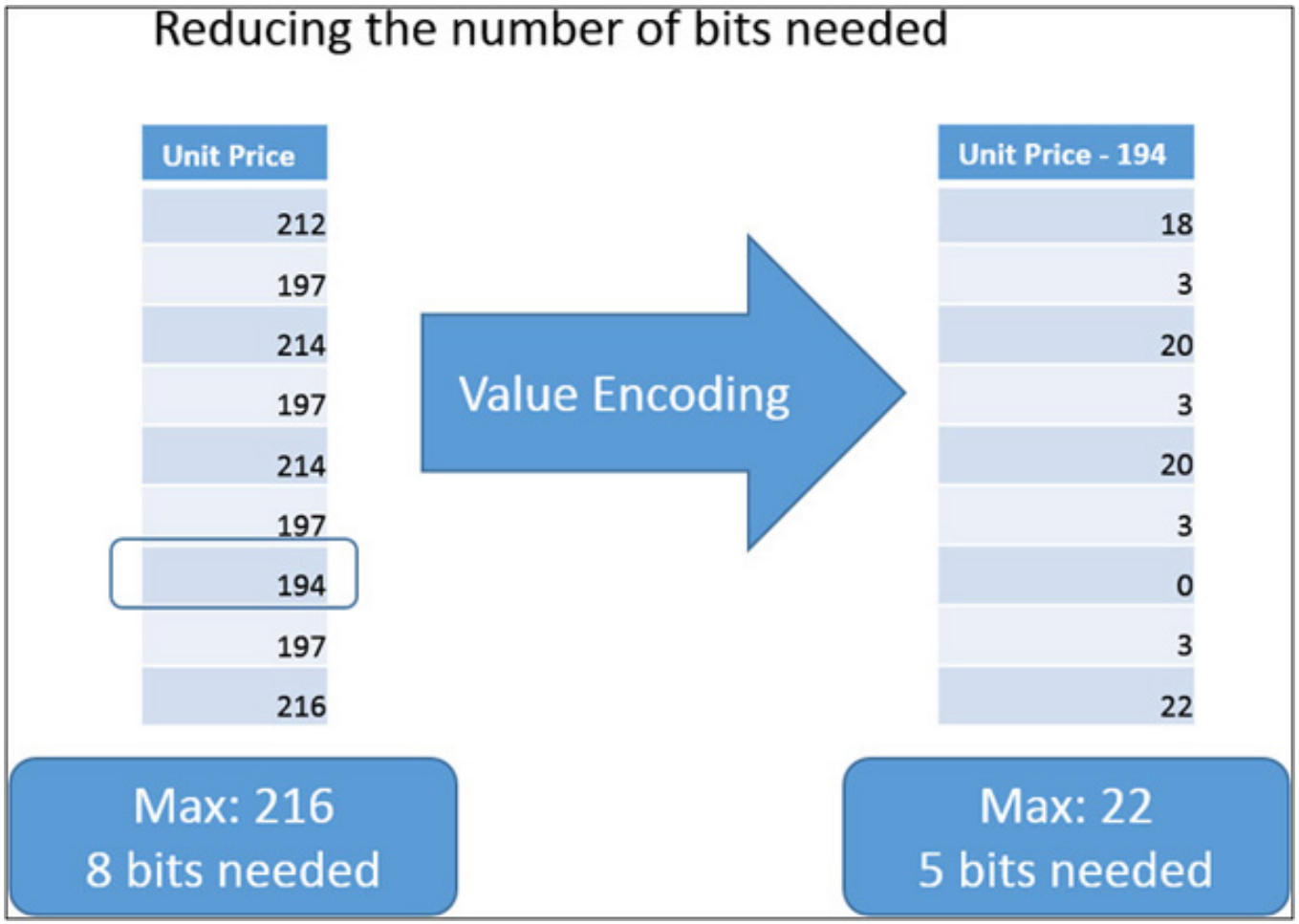
| Color |
|---|
| Red |
| Red |
| White |
| Black |
| Blue |
| Red |
| Blue |
| Black |
| Black |




| Color Id |
|---|
| 0 |
| 0 |
| 1 |
| 2 |
| 3 |
| 0 |
| 3 |
| 2 |
| 2 |

| Id | Color |
|---|---|
| 0 | Red |
| 1 | White |
| 2 | Black |
| 3 | Blue |

# Value Encoding



# Run-length Encoding

| Name | Last Name |
|---|---|
| Mark | Simpson |
| Mark | Donalds |
| John | Simpson |
| Andre | White |
| Andre | Donalds |
| Andre | Simpson |
| Ricardo | Simpson |
| Mark | Simpson |
| Charlie | Simpson |
| Mark | White |
| Charlie | Donalds |

| Name | Last Name |
|---|---|
| Mark | Simpson |
| Mark | Donalds |
| Mark | Simpson |
| Mark | White |
| John | Simpson |
| Andre | White |
| Andre | Donalds |
| Andre | Simpson |
| Ricardo | Simpson |
| Charlie | Simpson |
| Charlie | Donalds |

| Name | Last Name |
|---|---|
| Mark:4 | Simpson:1 |
| John:1 | Donalds:1 |
| Andre:3 | Simpson:1 |
| Ricardo:1 | White:1 |
| Charlie:2 | Simpson:1 |
| | White:1 |
| | Donalds:1 |
| | Simpson:3 |
| | Donalds:1 |

Efficiency strongly depends on the sort order of the table. So, in large tables it may be important to determine the best sorting of the data (obviously, all the columns in the same table are sorted in the same way).

# Cassandra Column Store

Apache Cassandra is an open-source, distributed column store, designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

- It is linearly scalable, fault-tolerant, and eventually consistent
- Developed at Facebook, first released in 2008
- Used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, eBay, Twitter, Netflix, and more
- As for the CAP theorem, it can be considered to be an *AP* system

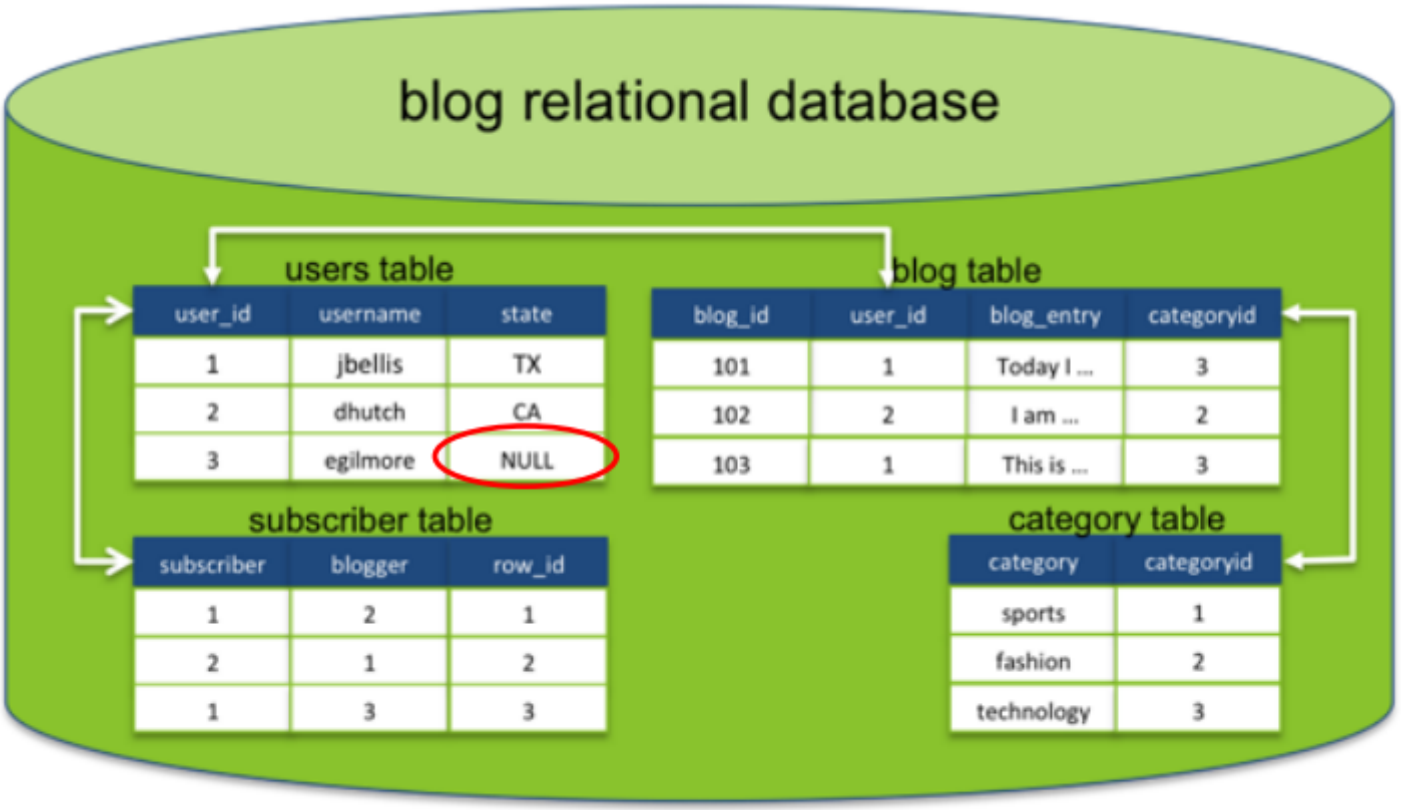
# Cassandra - RDBMS Comparison

## Terminology

| RDBMS | Cassandra |
|---|---|
| database instance | cluster |
| database schema | keyspace |
| table | column family |
| row | row |
| column (same for all rows) | column (can be different per row) |

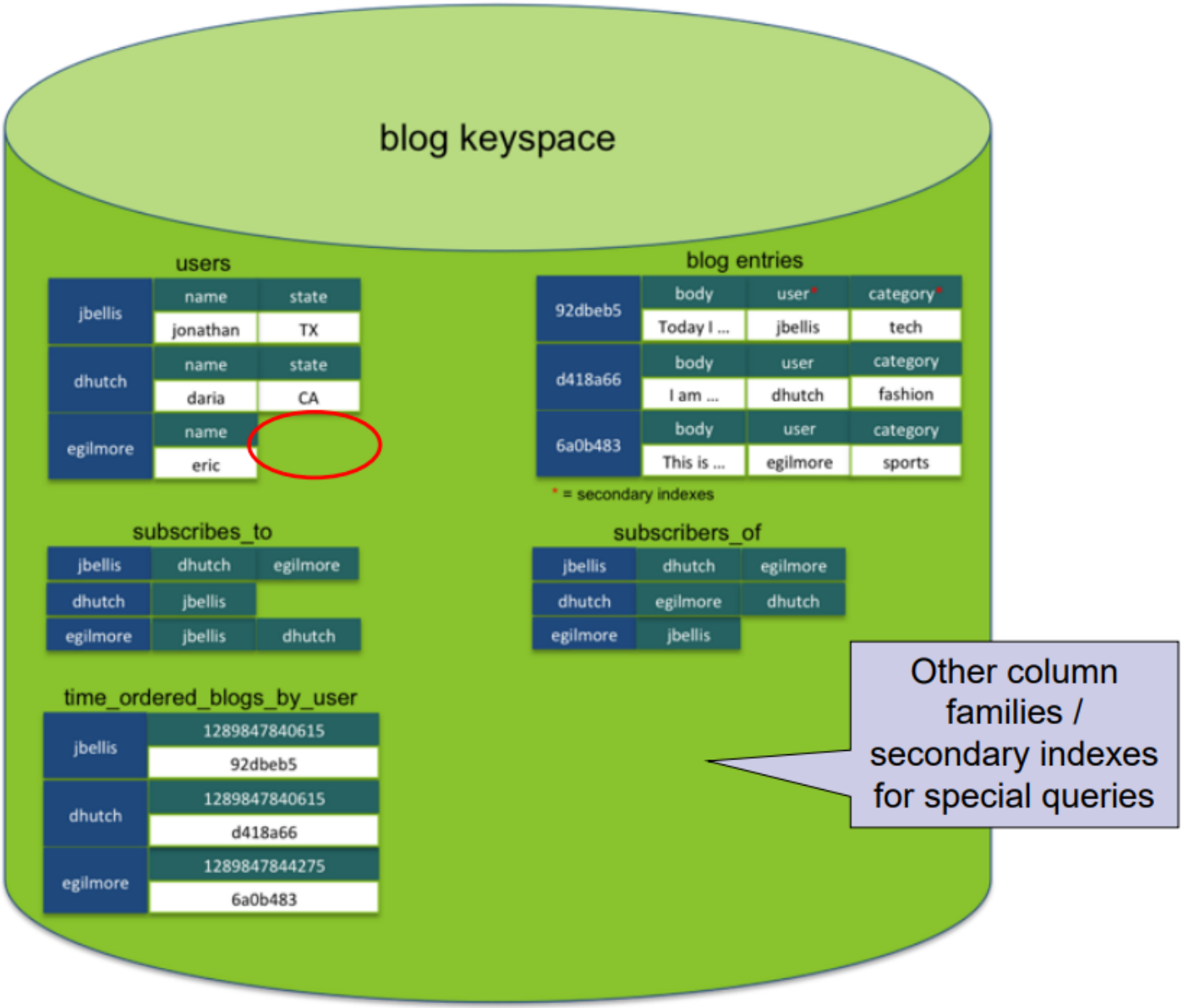
Usually one per application

# RDBMS Blog Example




NULL values still use some storage space.

# Keyspace Blog Example



# Query Language - CQL

Cassandra offers a specific language to interact with the data, Cassandra Query Language (CQL):

- SQL-like commands: CREATE, ALTER, UPDATE, DROP, DELETE, TRUNCATE, INSERT, ...
- Much simpler than SQL:
  - Does *not allow* joins or subqueries
  - WHERE clauses are simple (e.g., can only use AND operator with key columns)

Cassandra features also Hadoop integration, with MapReduce, Apache Pig and Apache Hive support.

# CQL - Examples

```
Example query 1:
        CREATE TABLE playlists(
        Id uuid,
        Song_order int,
        Song_id uuid,
        title text,
        album text,
        artist text,
        PRIMARY KEY (id, song_order));

Example query 2:
        INSERT INTO playlist (id, song_order, song_id, title, artist, album)
          VALUES (62c36092-82a1-3aoo-93d1-46196ee77204, 4,
                        7db1a490-5878-11e2-bcfd-o8oo2ooc9a66,
                        'ojo Rojo', 'Fu Manchu', 'No One Rides for Free');
Example query 3:
        SELECT * FROM playlists;
```

# CQL Data Types

| | |
|---|---|
| `ascii` | ASCII character string |
| `bigint` | 64-bit signed long |
| `blob` | Arbitrary bytes (no validation) |
| `boolean` | true or false |
| `counter` | Counter column (64-bit long) |
| `decimal` | Variable-precision decimal |
| `double` | 64-bit IEEE-754 floating point |
| `float` | 32-bit IEEE-754 floating point |
| `int` | 32-bit signed int |
| `text` | UTF8 encoded string |
| `timestamp` | A timestamp |
| `uuid` | Type 1 or type 4 UUID |
| `varchar` | UTF8 encoded string |
| `varint` | Arbitrary-precision integer |

# Other Column Family Stores

- *Bigtable*: compressed, high performance, proprietary data storage system built on Google File System
- *HBase*: part of Apache Hadoop framework, and modeled after Google's Bigtable
- *Vertica*: commercial, column-oriented relational DBMS with standard SQL support
- *Druid*: column-oriented, open-source, distributed data store written in Java, used by many companies such as Alibaba, Airbnb, and Cisco
- *Accumulo*: built on top of Apache Hadoop and based on Google's Bigtable, it is the third most popular NoSQL column store, behind Apache Cassandra and HBase

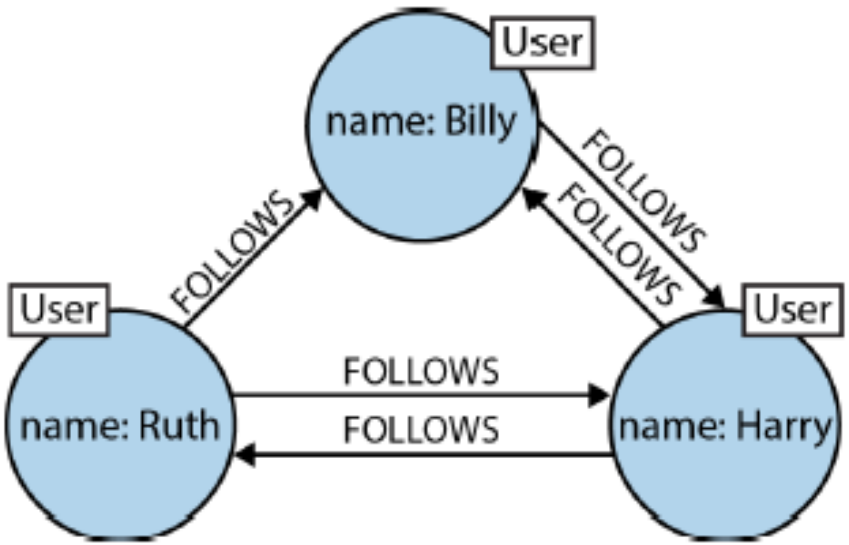
# Graph Databases

# Graph Databases

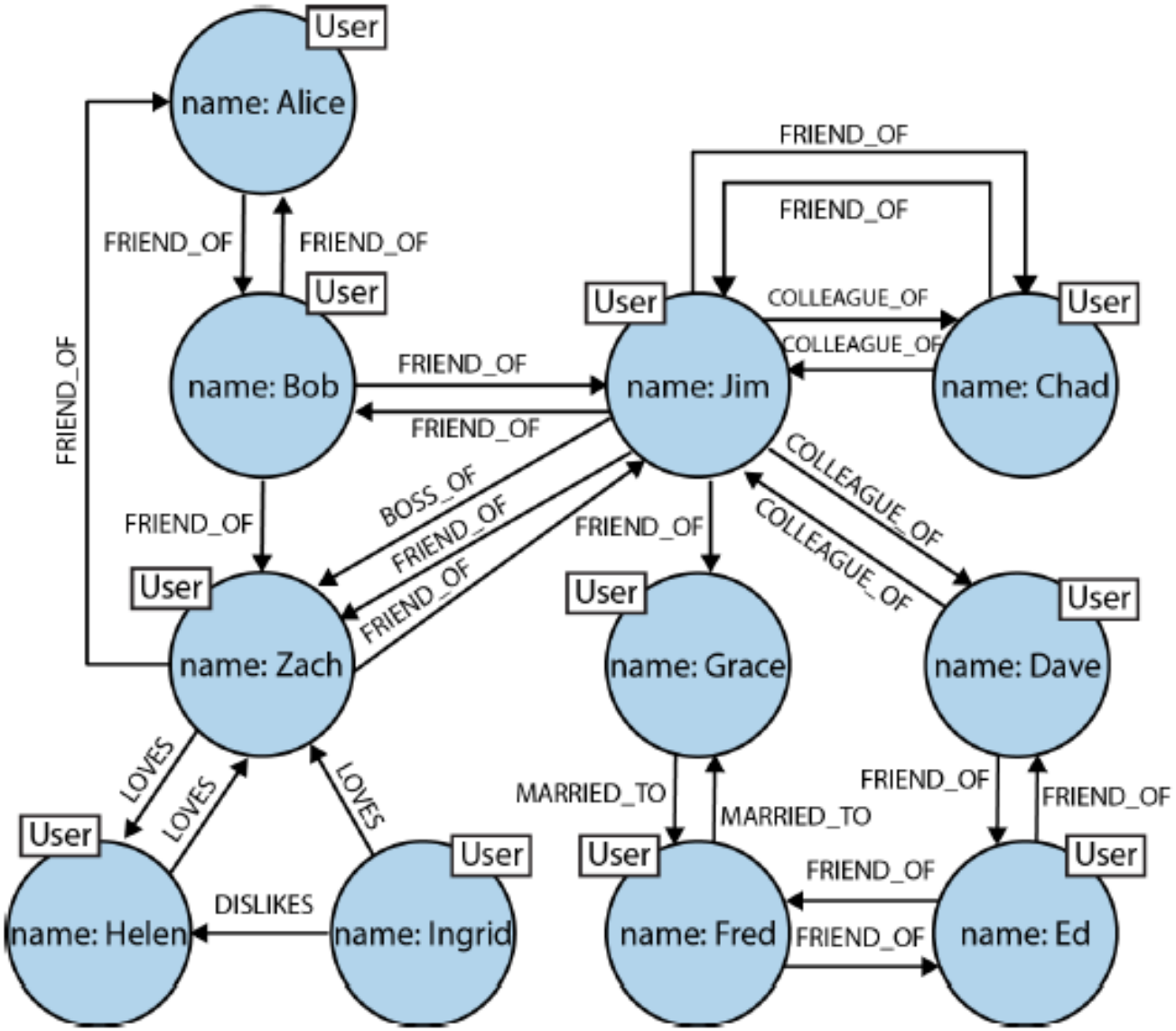
A graph database is a database that uses graph structures for semantic queries with *nodes* (possibly of different kinds), *edges*, and *properties* to represent and store data.

Graph databases can naturally represent certain kinds of semi-structured, **highly interconnected data**, such as those present in social networks, or in geospatial and biotech applications.

# Graph Example – 1



# Graph Example – 2



# Graph Databases – Why

Of course, the previous graphs could be modeled using other kinds of NoSQL approaches, as well as RDBMSs.

Nevertheless, the resulting databases would be very difficult to query, update, and populate.

On the contrary, graph databases can easily answer to queries such as:

- what is the shortest path connecting node $X$ with node $Y$?
- what are the friends of *Billy*?
- what are the friends of friends of *Billy*?

# Graph Databases – Key Properties

There are two important properties that characterize graph database technologies:

- **The underlying storage**: some graph databases use *native graph storage*, while others serialize data into a relational database, or rely on other NoSQL databases
- **The processing engine**: some graph databases are capable of native graph processing by means of *index-free adjacency*, meaning that connected nodes are **physically** "pointing" to each other

# Index-free Adjacency

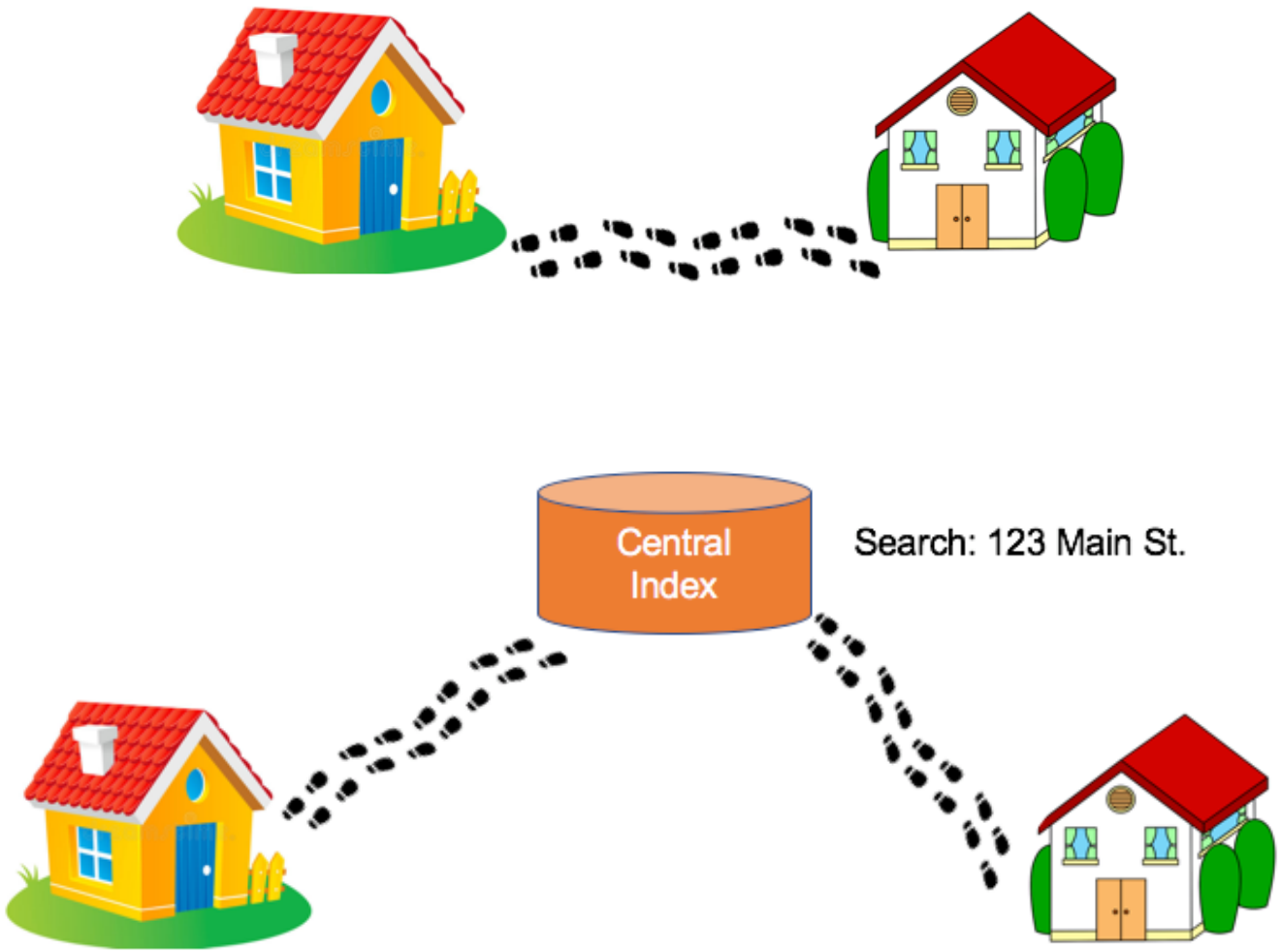
Native graph databases do not depend heavily on indexes because the graph itself provides a natural adjacency index.

Relationships in a native graph database provide **direct**, physical connection to directly related nodes, by means of pointers.

Such pointers allow traversing million of nodes in seconds, in contrast with the need for joining data which is several orders of magnitude slower.

As the graph grows in size, the cost of a local step remains the same.

# Native vs. Non-Native Graph Databases



# Graph Databases – Data Models

In addition to a specific storage model, graph databases may adopt a specific data model, such as:

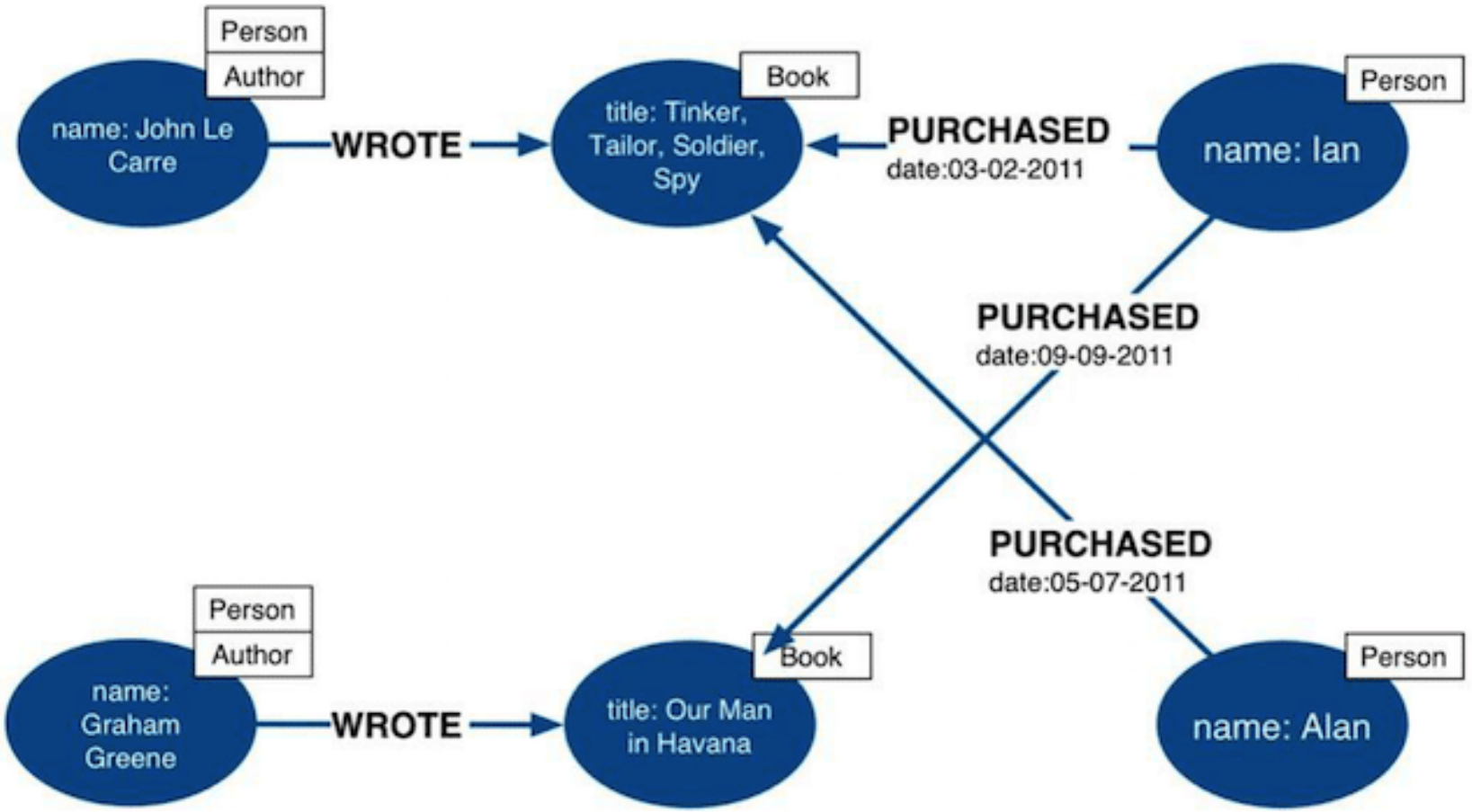
- Property graphs
- Hypergraphs
- Triples

# Graph Databases – Property Graphs

A property graph has the following characteristics:

- It contains nodes and relationships
- Nodes contain properties (key-value pairs)
- Nodes can be labeled with one or more labels (e.g., to encode specialization hierarchies)
- Relationships are named and directed, and always have a start and an end node
- Relationships can also contain properties

# Property Graph – Example

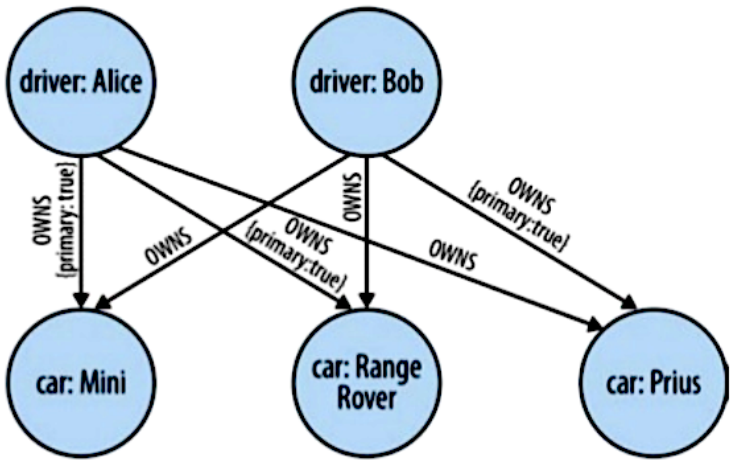


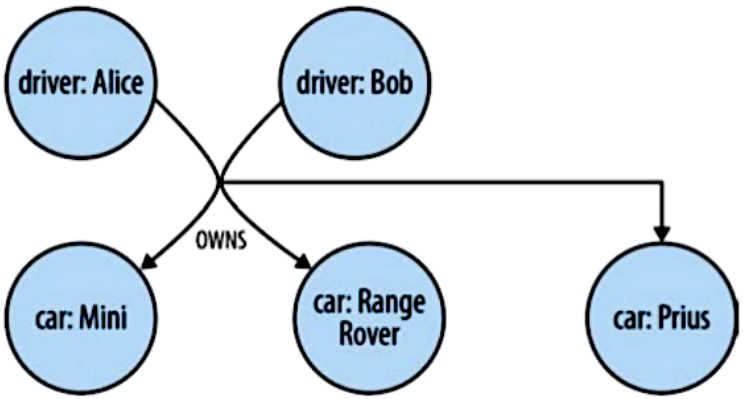
# Graph Databases – Hypergraphs

An hypergraph is a generalized model in which a relationship (called an hyperedge) can connect any number of nodes.

Hypergraphs can be particularly useful to deal with scenarios involving many-to-many relationships.

It is always possible to represent the information in a hypergraph by means of a classical property graph.



# Graph Databases – Triples

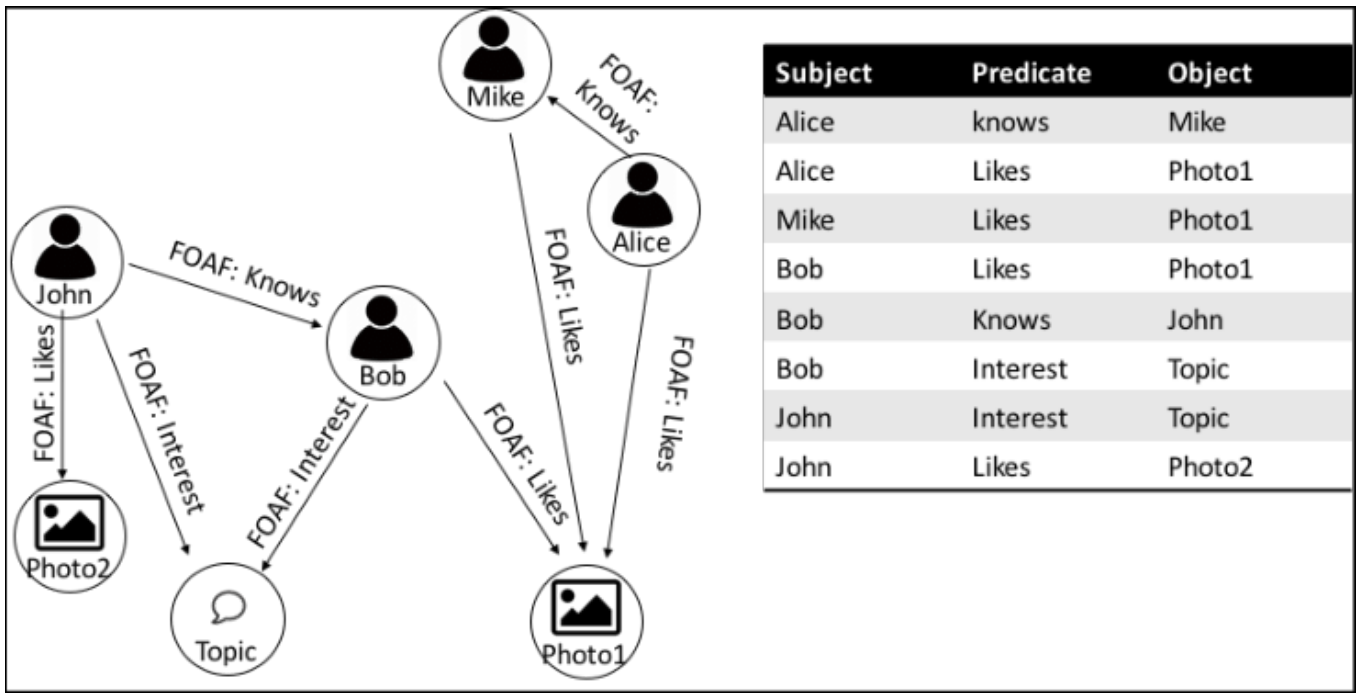
Triple stores are related to the Semantic Web movement, whose goal is to make Internet data machine-readable.

Semantic Web is focused on harvesting useful data and relationship information from the Web and storing it in triples for querying.

A triple is a subject-predicate-object data structure. By means of triples, it is possible to capture facts, such as “Ginger dances with Fred” and “Fred likes ice cream”.

Triple stores typically provide SPARQL capabilities to retrieve and manipulate RDF (Resource Description Framework) data, so data format and query language are standardized.

# Triple store – Example




| Subject | Predicate | Object |
|---|---|---|
| Alice | knows | Mike |
| Alice | Likes | Photo1 |
| Mike | Likes | Photo1 |
| Bob | Likes | Photo1 |
| Bob | Knows | John |
| Bob | Interest | Topic |
| John | Interest | Topic |
| John | Likes | Photo2 |

# Graph Databases: Giraph

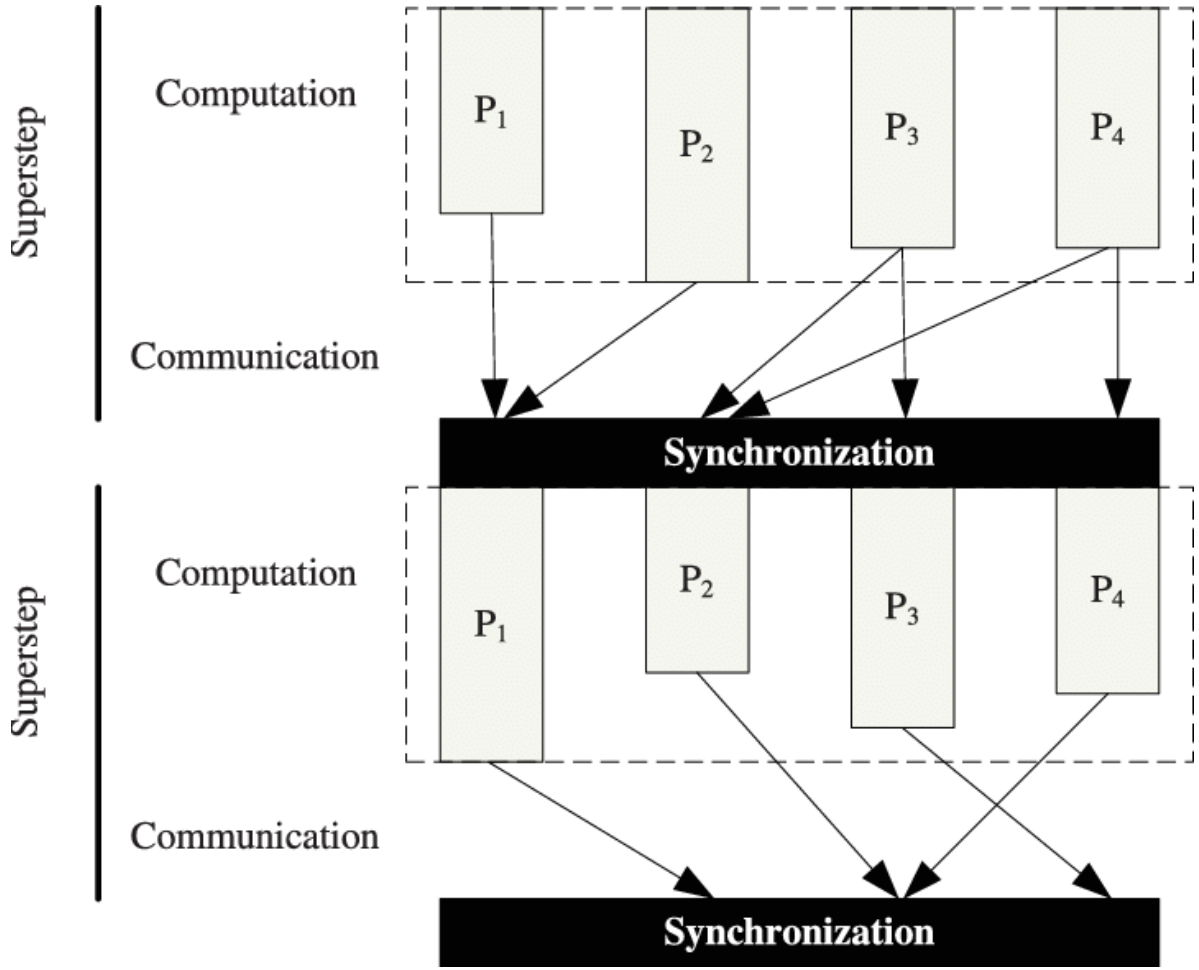
Apache Giraph was developed by Yahoo and later donated to the Apache Foundation.

Giraph relies on Apache Hadoop's MapReduce implementation to process graphs.

It is based on the Bulk Synchronization Parallel model (1980). Every computation is based on three supersteps:

- *parallel computing*: participating processors may perform local computations (may overlap with communication)
- *communication*: processes exchange data between themselves
- *barrier synchronization*: when a process reaches the barrier, it waits until all other processes have reached the same barrier

# Bulk Synchronization Parallel Model



# Other Examples of Graph Databases

**Neo4J** is a very performing, native, open source property graph database that guarantees ACID properties offering scalability up to billions of nodes.

- Online sandbox: `https://neo4j.com/sandbox/`

**AllegroGraph** is a closed source triplestore, designed to store RDF triples; it supports ACID properties.

**ArangoDB** is a multi-model, open source, database system that supports three data models (key/value, documents, graphs).

# Bibliography

Stavros Harizopoulos, Daniel Abadi, Peter Boncz (2009), Column-Oriented Database Systems, VLDB 2009 Tutorial

Ian Robinson, Jim Webber & Emil Eifrem, Graph Databases Second Edition, O'Reilly Media, Inc.

Big Data Management and NoSQL Databases Lecture 7. Column-family stores, Doc. RNDr. Irena Holubova, Ph.D.