DMIF, University of Udine

Data Management for Big Data

Introduction To NoSQL

Andrea Brunello
andrea.brunello@uniud.it

May, 2021

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

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*).

Benefits of NoSQL

Elastic Scaling:

- RDBMS scale up: bigger load ⇒ 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 simpler data models

Big Data:

- Huge increase in data, RDBMS: capacity and constraints of data volumes at their limits
- NoSQL are specifically designed for big data

Drawbacks of NoSQL

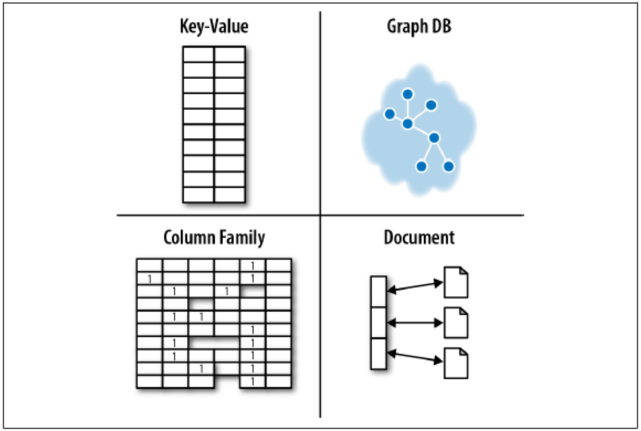
Several default constraints of RDBMS are not supported at the database level (e.g., foreign keys).

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).

The NoSQL Quadrant



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.

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

Key/Value Store - Focus

Key/values stores offer just the add/remove/modify/find operations on 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** is a high performance key/value database from Oracle Inc., 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 and fault protection

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, nonetheless they 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 JSON and XML.

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

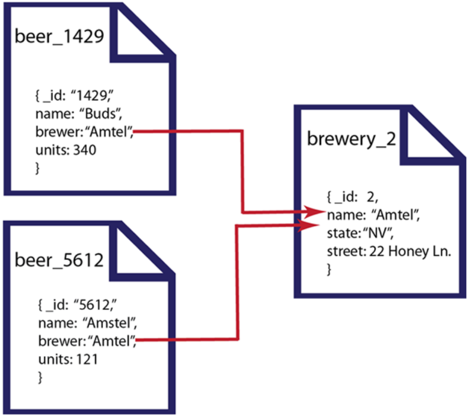
Document-oriented Databases – Operations

The core operations that a document-oriented database supports for documents are similar to other databases, 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

$\rightarrow$ **MongoDB** is an example of a document-oriented database.

Document example



Column-oriented Stores

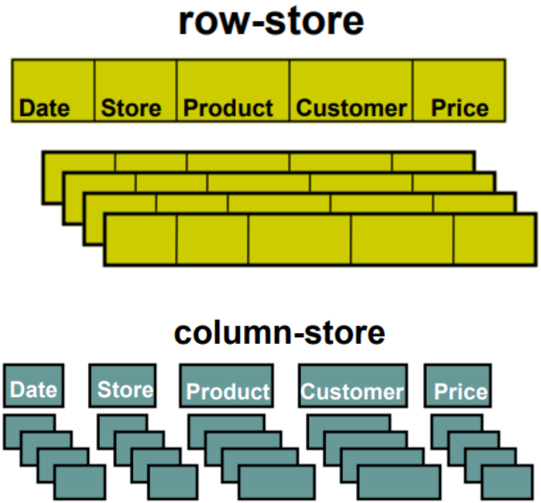
The roots of colum-store DBMSs can be traced in the 1970s.

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.

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



Column Family - Blocks On Disk

In a classical 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).

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 construction.

Newly inserted tuples have to be broken up into their component attributes, and each attribute bust be written separately.

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

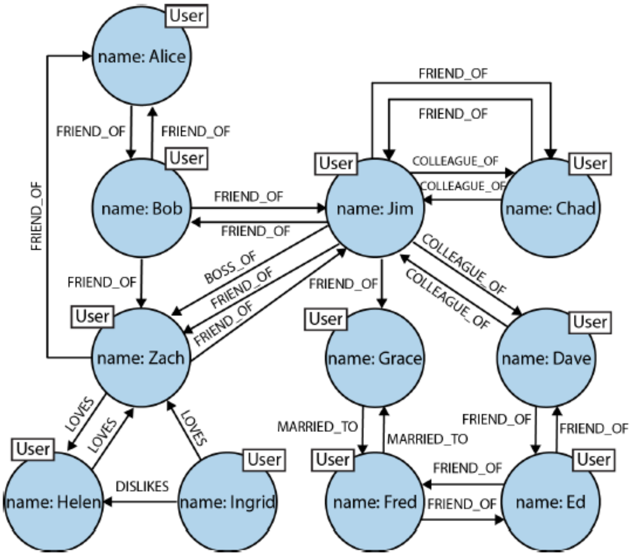
→ **Cassandra** is an example of a column store database.

Graph Databases

A graph database is a database that uses graph structures for semantic queries with nodes, 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



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*?

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.

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).

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

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.