DMIF, University of Udine

Introduction To NoSQL

Andrea Brunello
andrea.brunello@uniud.it

May, 2020

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

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 data velocity.

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

Data is 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.

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).

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 it didn't happen).

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 successful, subsequent transaction.

BASE Properties

In order to better support the characteristics of big 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 don't have 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 recent write or an error
- *Availability*: every request receives a (non-error) response, with no guarantee that it contains the most recent write
- *Partition tolerance*: the system is allowed to lose arbitrarily many messages sent from one node to another (e.g., when a network partition failure happens)

The CAP Theorem - 2

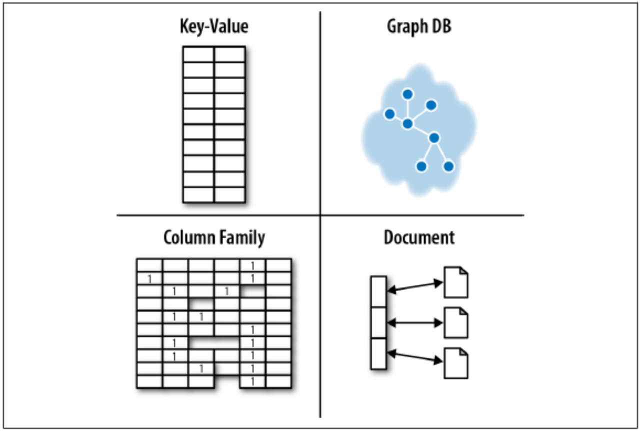
In fact, the choice is really between consistency and availability when a network partition failure happens; at all other times, no trade-off has to be made.

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 (e.g., Cassandra)
- *CP*: data is consistent, and maintains tolerance for network partitioning preventing data going out of sync (e.g., MongoDB, HBase)

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.

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

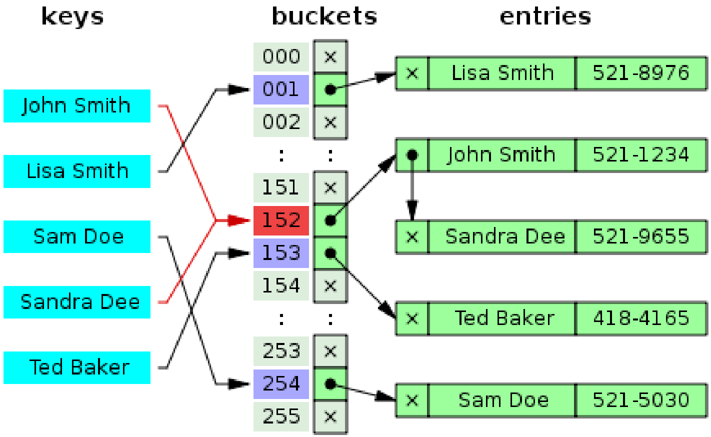
The hash table is a data structure that consists of an array of values and a hash function that maps any given key to a position inside the array.

The hash function transforms a string into an integer, 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



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 - Berkeley DB

Berkeley DB is a high performance key/value database from Oracle Inc., with implementations in C, Java and XML/C++.

It supports complex key-value types, ACID transactions, multi-processing and multi-threading for high concurrency, hot and cold backup, and distribution with *replicas*.

It can store <key, values> in four different data structures:

- *B-tree*: self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time
- *Hash*: good for large quantities of data
- *Queue*: a sequence of fixed-length records (key = index)
- *Recno*: basically a queue with variable-length records

Key/Value Store - Examples - Voldemort

Project Voldemort is a (LinkedIn-born) distributed key/value store based on Berkeley DB, designed for performance and fault protection.

It is flexible, having several software layers devoted to specific tasks, each exposing the put (add), get (find), delete (remove) methods.

Data are partitioned and replicated onto different servers, each storing only a subset of the entire data. This brings:

- horizontal scalability
- fault tolerance

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 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

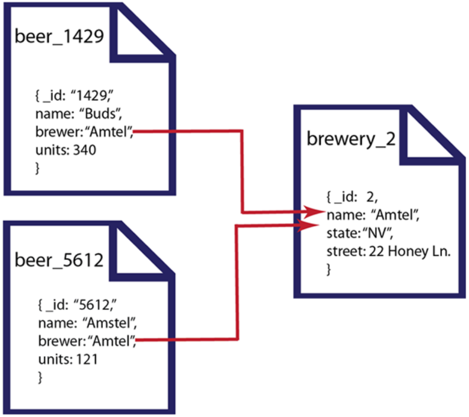
The core operations that a document-oriented database supports for documents are similar to other databases, and are named as CRUD:

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

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.

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

Document example



Column-oriented Stores

Column-oriented Stores

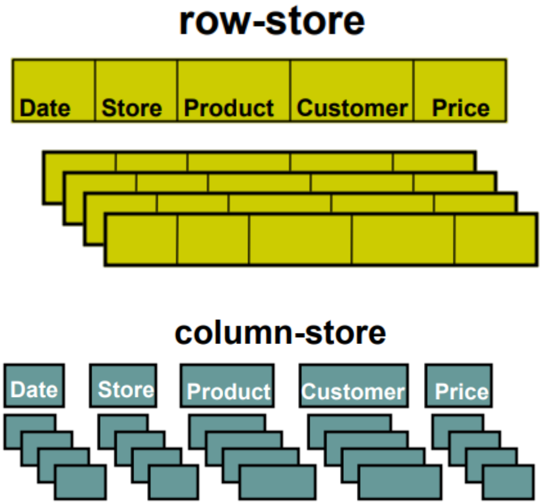
The roots of colum-store DBMSs can be traced in the 1970s.

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 an OLTP 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
```

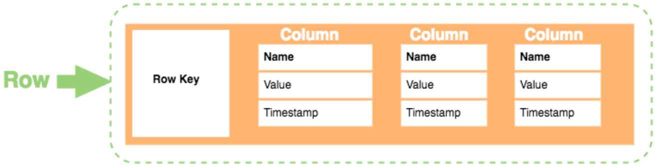
In a columnar database, blocks on the disk for the above data might look like this:

```
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 Data Model

Organizing data in columns it's not an exclusive of NoSQL realm but it's widely adopted in analytics application or Business Intelligence.

Data model is similar to key/value store; however, here each data unit (a row), identified by a key, includes several key-value tuples (one for each column value).



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 construction.

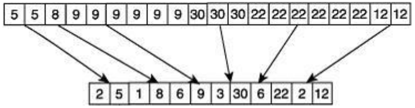
Newly inserted tuples have to be broken up into their component attributes, and each attribute bust be written separately.

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.

Compression with column stores

Column databases are exploited for 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).

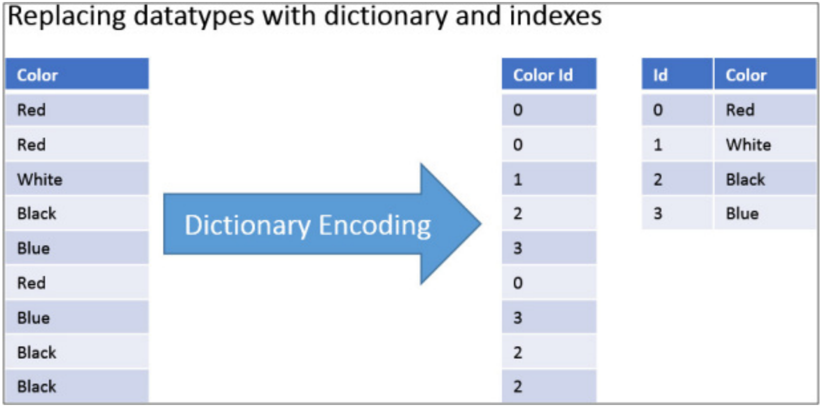


The first step performed by the compressing system is *encoding*, which converts data values to integers.

Two techniques are involved in this phase:

- *dictionary encoding*
- *value encoding*

Dictionary Encoding



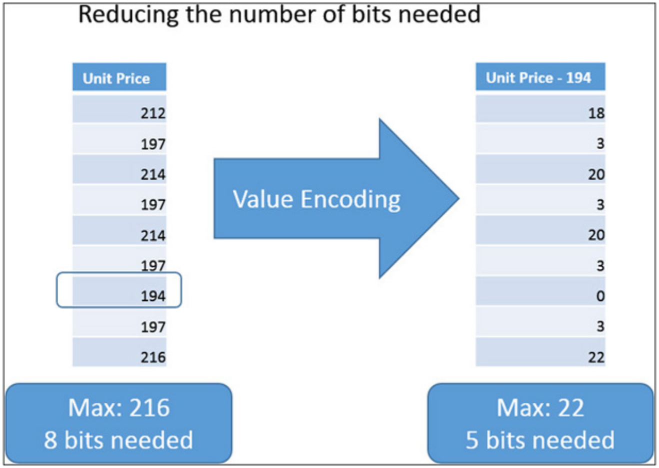
Replacing datatypes with dictionary and indexes

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



Reducing the number of bits needed

Unit Price
212
197
214
197
214
197
194
197
216

Max: 216
8 bits needed

Unit Price - 194
18
3
20
3
20
3
0
3
22

Max: 22
5 bits needed

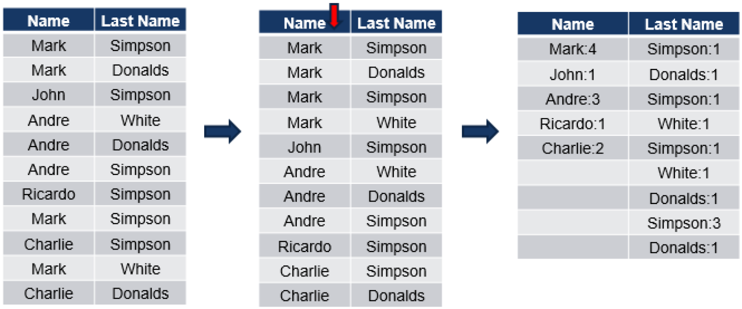
Compression Techniques

After the encoding phase there is the proper compression phase. Different techniques are available.

The two main algorithms for compressing data are:

- *Run-length encoding*, which is efficient in case of few unique values
- *Bit Packing*, that merges bits from different integers

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

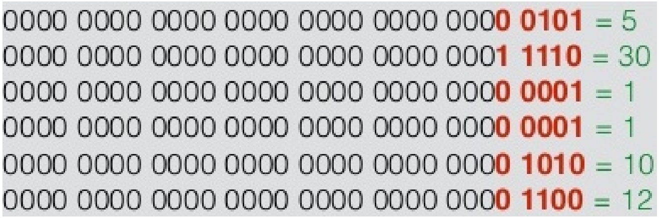
Efficiency strongly depends on the sort order of the table. So, in large tables it may be important to determine the best sorting of the data (obviously, all the columns in the same table are sorted in the same way).

Bit Packing

Efficient technique to store blocks of small integers, that supports random access.

Say you want to store:

- 5, 30, 1, 1, 10, 12 (32 bit integers)
- Raw data: $6 * 32 = 192$ bits
- Packed: $6 * 5 = 30$ bits (84% size reduction!)

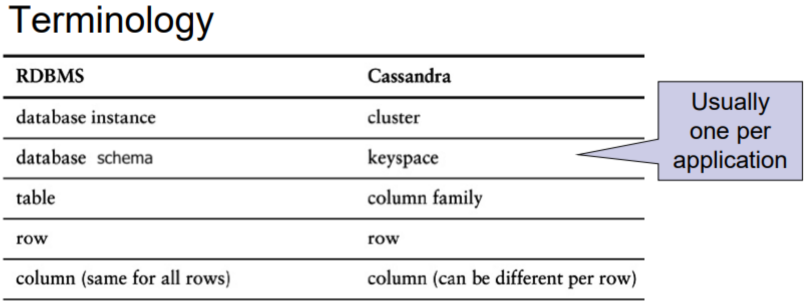

```
0000 0000 0000 0000 0000 0000 0000 0101 = 5
0000 0000 0000 0000 0000 0000 0001 1110 = 30
0000 0000 0000 0000 0000 0000 0000 0001 = 1
0000 0000 0000 0000 0000 0000 0000 0001 = 1
0000 0000 0000 0000 0000 0000 0000 1010 = 10
0000 0000 0000 0000 0000 0000 0000 1100 = 12
```

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

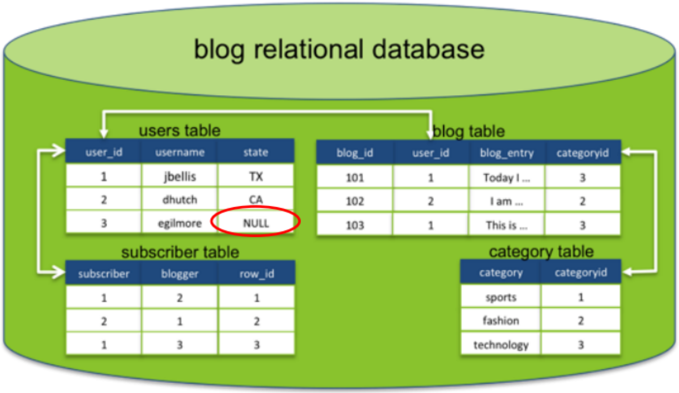


Terminology

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



users table

user_id	username	state
1	jbellis	TX
2	dhutch	CA
3	egilmore	NULL

blog table

blog_id	user_id	blog_entry	categoryid
101	1	Today I ...	3
102	2	I am ...	2
103	1	This is ...	3

subscriber table

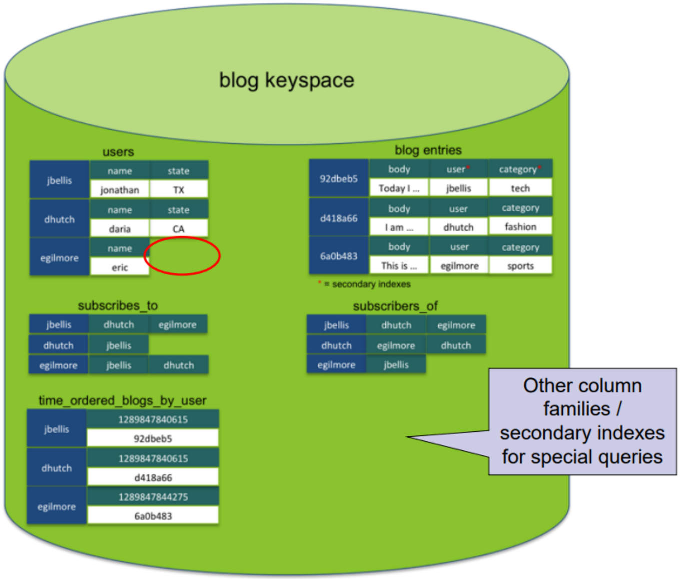
subscriber	blogger	row_id
1	2	1
2	1	2
1	3	3

category table

category	categoryid
sports	1
fashion	2
technology	3

NULL values still use some storage space.

Keyspace Blog Example



blog keyspace
Other column families / secondary indexes for special queries

users

jbellis	name	state
	jonathan	TX
dhutch	name	state
	daria	CA
egilmore	name	
	eric	

blog entries

92dbeb5	body	user*	category*
	Today I ...	jbellis	tech
d418a66	body	user	category
	I am ...	dhutch	fashion
6a0b483	body	user	category
	This is ...	egilmore	sports

* = secondary indexes

subscribes_to

jbellis	dhutch	egilmore
dhutch	jbellis	
egilmore	jbellis	dhutch

subscribers_of

jbellis	dhutch	egilmore
dhutch	egilmore	dhutch
egilmore	jbellis	

time_ordered_blogs_by_user

jbellis	1289847840615
	92dbeb5
dhutch	1289847840615
	d418a66
egilmore	1289847844275
	6a0b483

Cassandra - Data Models

The building blocks that define the Cassandra data model are the following (from higher to lower level):

- *Cluster*
- *Keyspace*
- *Column Family*
- *Column*
- *Super Column*
- *Row*

Cluster and Keyspace

The *cluster* is given by the set of servers that constitute the given Cassandra instance. A cluster can be considered as an RDBMS database instance, and includes one or more keyspaces.

The *keyspace* is a namespace for column families. It can be regarded as an RDBMS database schema.

Column Family

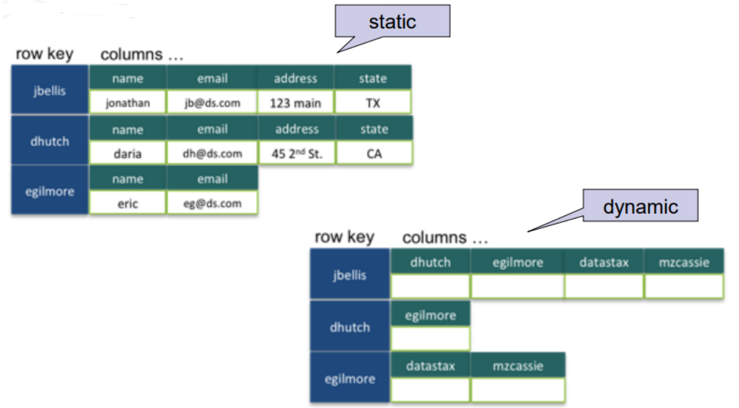
A *column family* is a column container (similar to the concept of RDBMS table).

Each column family is stored on a separate file, thus it is best practice to put together columns that are frequently accessed together.

A column family can be either static or dynamic:

- *static*: columns metadata (e.g., name and type) are defined a-priori, similarly to an RDBMS table. Rows have the same set of columns, though some columns may be undefined for some rows.
- *dynamic*: takes advantage of Cassandra's ability to use arbitrary, application-supplied columns for the single rows. Useful for handling unstructed data.

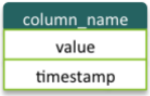
Column Family - Example



Users that subscribe to a particular user's blog

Column

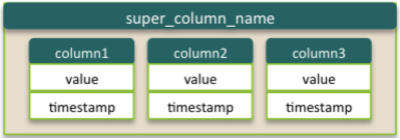
A *column* is the basic data structure of Cassandra with three values, namely *column name*, *value*, and *time stamp* (used for conflict resolution).



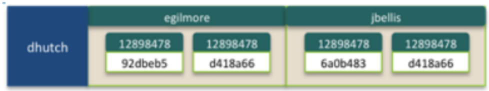
- Can be indexed on its name:
  - Using a secondary index
  - Primary index = row key, this ensures uniqueness, speeds up access, can influence storage order
- Several types:
  - *Expiring*, with an expiration (future delete) date called TTL
  - *Counter*, to store an incremental value (e.g., page views)
  - *Super*, to add another level of nesting

Super Column - Super Column Family

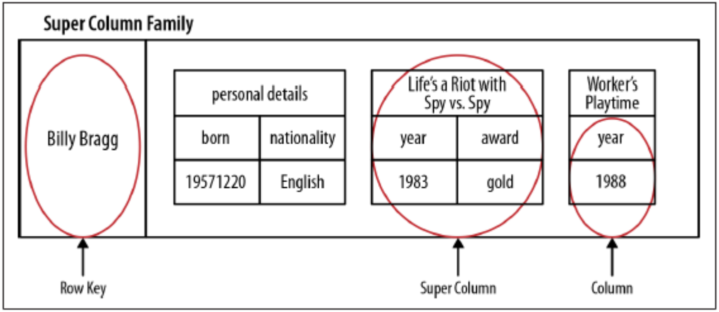
A *super column* is a column consisting of a map of columns. It has a name, and value involving the map of columns.



A *super column family* is a column family that contains super columns.



Super Column Family Example



Row

Each *row* is uniquely identified by its *key* (the equivalent of RDBMS primary key), which can be simple (single column) or compound (multiple columns).

The key determines on which server the row data will be stored.

Each row of a column family may contain all or only a subset of column values.

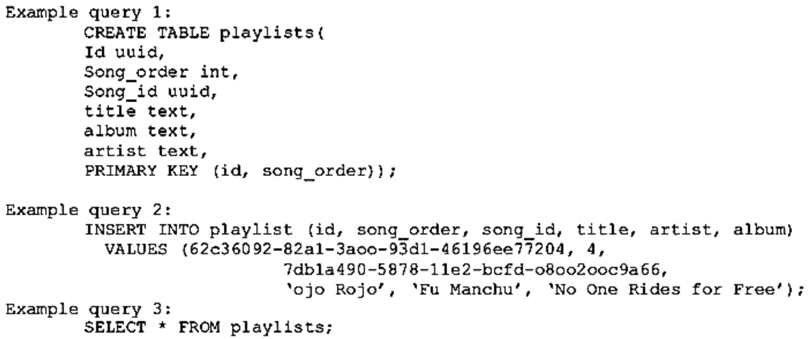
Query Language - CQL

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 Examples From Column Family

- *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

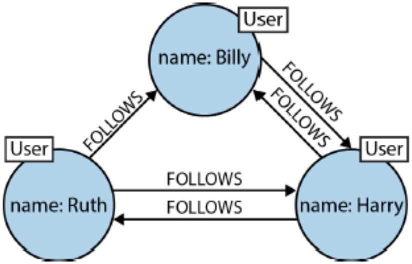
Graph Databases

A graph database is a database that uses graph structures for semantic queries with nodes, edges, and properties to represent and store data.

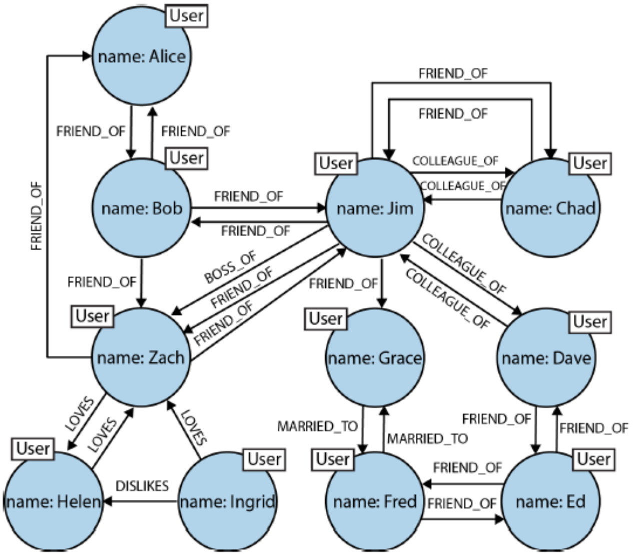
Graph databases can naturally represent certain kinds of semi-structured, highly interconnected data, such as those present in social networks, or in geospatial and biotech applications.

Graph databases are built for use with transactional (OLTP) systems and are engineered with ACID support (e.g., *Amazon Neptune*, *Neo4j*). Nevertheless, OLAP systems are also available (e.g., *Spark GraphX*, *Apache Giraph*).

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, use an object-oriented 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 in the database

Index-free Adjacency

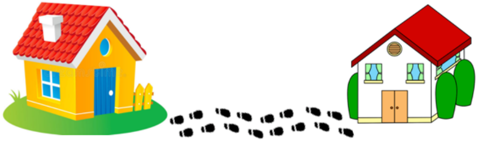
Native graph databases do not depend heavily on indexes because the graph itself provides a natural adjacency index.

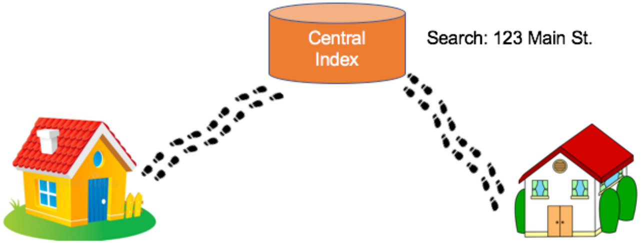
Relationships in a native graph database provide direct, physical connection to directly related nodes, by means of pointers.

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:

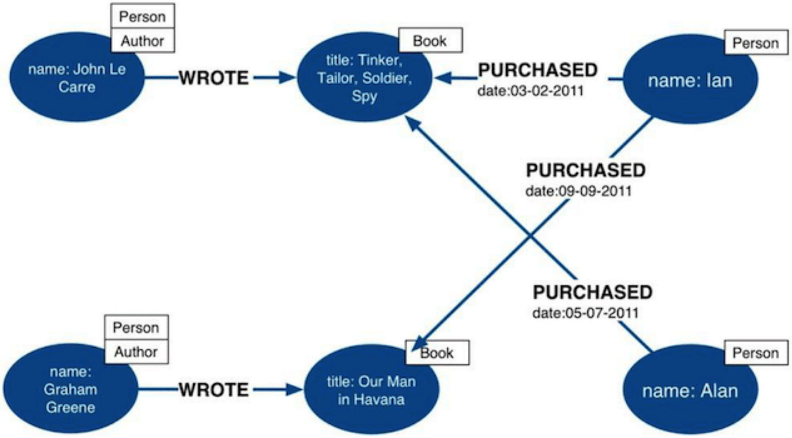
- Property graphs
- Hypergraphs
- Triples

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 end node
- Relationships can also contain properties

Property Graph – Example

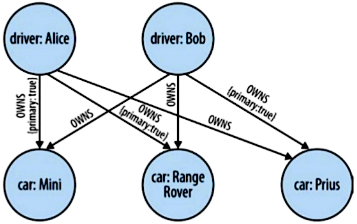


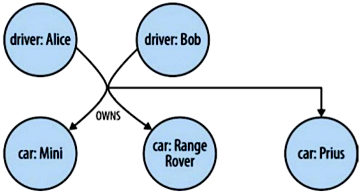
Graph Databases - Hypergraphs

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 as with 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.

Graph Databases: Giraph

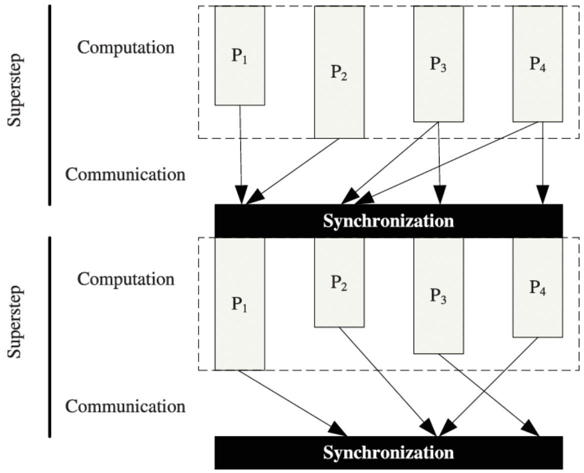
Apache Giraph was developed by Yahoo and later donated to the Apache Foundation.

Giraph utilizes 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.

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).