

# 1 Understanding NoSQL [1]

NoSQL refers to a database that is not based on SQL (Structured Query Language), which is the language most commonly associated with relational databases. Essentially, NoSQL data isn't relational, NoSQL databases usually do not have schema, and they come with looser consistency models than traditional relational databases do.

The term "NoSQL" refers to the fact that traditional relational databases are not adequate for all solutions, particularly ones involving large volumes of data. But the term has been extended to also mean "Not only SQL", indicating support for potential SQL-based interfaces even if the core database isn't relational. Software developers that use NoSQL solutions don't necessarily advocate dismissing relational databases, but instead see value in using the right data store for the job.

## 1.1 Use of NoSQL

NoSQL data stores respond to key data needs that are not met by relational databases.

## 1.2 Key-value stores

Some NoSQL databases store key-value pairs to allow quick lookups, for example, in the case of question/answer access. Relational databases are more oriented towards storing complex data and various relationships between different types of data. This methodology is overcomplicated when a developer wants to implement a quick way to store and access Q&A data.

## 1.3 Document stores

Other types of data are more document oriented and have variations. For example, forms of data can have many optional fields. Relational databases with their rigid schemas require all of these fields to be defined for every row of stored data. Document-based NoSQL stores are more flexible and efficient in handling this.

## 1.4 NoSQL what does it mean? [2]

What does NoSQL mean and how do you categorize these databases? NoSQL means Not Only SQL, implying that when designing a software solution or product, there is more than one storage mechanism that could be used based on the needs. The term NoSQL was coined by Carlo Strozzi in the year 1998. He used this term to name his Open Source, Light Weight database which did not have an SQL interface. In the early 2009, when last.fm wanted to organize an event on open-source distributed databases, Eric Evans, a Rackspace employee, reused the term to refer databases which are non-relational, distributed, and do not conform to atomicity, consistency, isolation, durability - four obvious features of traditional relational database systems. NoSQL was a hashtag (#nosql) chosen for a conference held in Atlanta, USA, to discuss these new databases. The most important result of the rise of NoSQL is Polyglot Persistence. NoSQL does not have a prescriptive definition but we can make a set of common observations, such as:

- Not using the relational model
- Running well on clusters
- Mostly open-source
- Built for the 21st century web estates
- Schema-less

## **1.5 Faster access to bigger sets of data**

Relational databases sacrifice performance when searching large volumes of data. Historically, developers have built systems in which writing SQL queries to find a few rows of data involved thinning out data sets in the most efficient way. The bigger the result set, though, the more expensive the queries become. Large volumes of data or queries that involve aggregating large amounts of data are referred to as "data warehousing."

NoSQL data stores are becoming widely adopted and are being tested in many situations. These situations involve large volumes of data as well as large rates of data growth in many consumer systems.

## **1.6 Less rigid consistency requirements**

NoSQL is also considered an alternative to traditional relational databases because certain consistency requirements that are inherently part of relational databases are very different in modern enterprises.

Developers are discovering that certain data requirements don't demand the rigid ACID model of relational databases that usually comes with worse performance. Instead they can meet their needs using eventual consistency that tends to come with better performance. Some NoSQL data stores even allow the developer to pick how loose or rigid the consistency must be.

## **1.7 Limitations of NoSQL**

SQL is a powerful, 40-year-old standard that has been possible because all relational databases have the same concept of storing data in tables and relating data through foreign keys. Although switching from one relational database to another isn't 100% transparent, it is much easier than switching between two different NoSQL data stores. Developers that have learned SQL have little challenge switching between vendors.

Because each NoSQL data store has unique aspects in both how its data is stored as well as how different bits of data relate to each other, no single API manages them all. When embracing a new NoSQL data store, the developer must invest time and effort to learn the new query language as well as the consistency semantics.

## 2 Classical relational databases

### 2.1 Introduction

In the computing system (web and business applications), there are enormous data that comes out every day from the web. A large section of these data is handled by Relational Database Management Systems (RDBMS). The idea of relational model came with E.F. Codd's 1970 paper "A relational model of data for large shared data banks" which made data modeling and application programming much easier. Beyond the intended benefits, the relational model is well-suited to client-server programming and today it is predominant technology for storing structured data in web and business applications.

Over the last few years however we have seen the rise of a new type of databases, known as NoSQL databases, that are challenging the dominance of relational databases. Relational databases have dominated the software industry for a long time providing mechanisms to store data persistently, concurrency control, transactions, mostly standard interfaces and mechanisms to integrate application data, reporting. The dominance of relational databases, however, is cracking.

### 2.2 Classical relational database follow the ACID Rules [3]

A database transaction, must be atomic, consistent, isolated and durable.

- **Atomic** : A transaction is a logical unit of work which must be either completed with all of its data modifications, or none of them is performed.
- **Consistent** : At the end of the transaction, all data must be left in a consistent state.
- **Isolated** : Modifications of data performed by a transaction must be independent of another transaction. Unless this happens, the outcome of a transaction may be erroneous.
- **Durable** : When the transaction is completed, effects of the modifications performed by the transaction must be permanent in the system.

Often these four properties of a transaction is acronymed as ACID.

### 2.3 Distributed Systems [3]

A distributed system consists of multiple computers and software components that communicate through a computer network (a local network or by a wide area network). A distributed system can consist of any number of possible configurations, such as mainframes, workstations, personal computers, and so on. The computers interact with each other and share the resources of the system to achieve a common goal.

#### 2.3.1 Advantages of Distributed Computing

- **Reliability (fault tolerance)** :The important advantage of distributed computing system is reliability. If some of the machines within the system crash, the rest of the computers remain unaffected and work does not stop.
- **Scalability** : In distributed computing the system can easily be expanded by adding more machines as needed.
- **Sharing of Resources** : Shared data is essential to many applications such as banking, reservation system. As data or resources are shared in distributed system, other resources can be also shared (e.g. expensive printers).

- **Flexibility** : As the system is very flexible, it is very easy to install, implement and debug new services.
- **Speed** : A distributed computing system can have more computing power and it's speed makes it different than other systems.
- **Open system** : As it is open system, every service is equally accessible to every client i.e. local or remote.
- **Performance** : The collection of processors in the system can provide higher performance (and better price/performance ratio) than a centralized computer.

### 2.3.2 Disadvantages of Distributed Computing

- **Troubleshooting** : Troubleshooting and diagnosing problems.
- **Software** : Less software support is the main disadvantage of distributed computing system.
- **Networking** : The network infrastructure can create several problems such as transmission problem, overloading, loss of messages.
- **Security** : Easy access in distributed computing system increases the risk of security and sharing of data generates the problem of data security

## 2.4 Scalability [3]

In electronics (including hardware, communication and software), scalability is the ability of a system to expand to meet your business needs. For example scaling a web application is all about allowing more people to use your application. You scale a system by upgrading the existing hardware without changing much of the application or by adding extra hardware.

There are two ways of scaling horizontal and vertical scaling :

### 2.4.1 Vertical scaling

To scale vertically (or scale up) means to add resources within the same logical unit to increase capacity. For example to add CPUs to an existing server, increase memory in the system or expanding storage by adding hard drive.

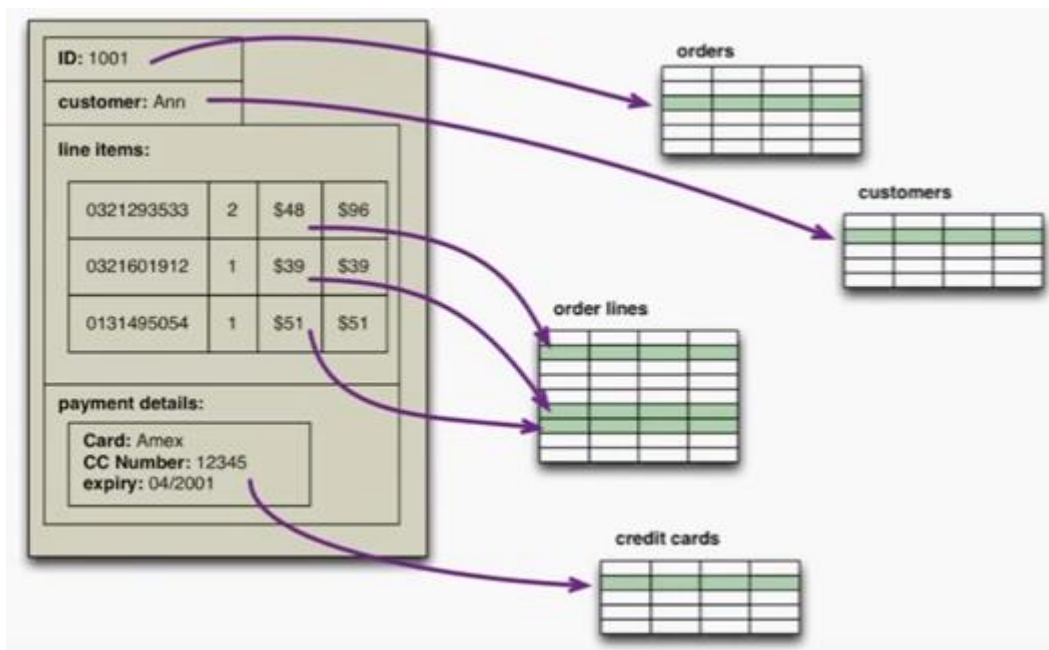
### 2.4.2 Horizontal scaling

To scale horizontally (or scale out) means to add more nodes to a system, such as adding a new computer to a distributed software application. In NoSQL system, data store can be much faster as it takes advantage of “scaling out” which means to add more nodes to a system and distribute the load over those nodes.

## 3 Why NoSQL databases? [2] [4]

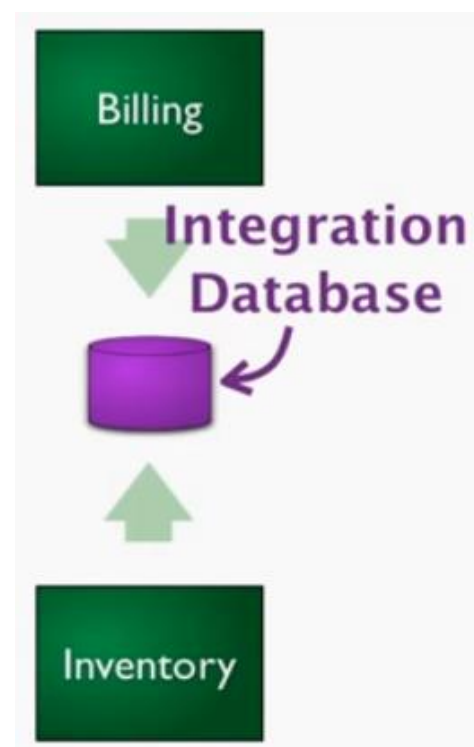
### 3.1 Impedance mismatch

Most application developers run into the problem that in their software they are working with cohesive structures of objects in memory, and then in order to save it off to the database they have to stripe the object over multiple tables, so the object ends up being splattered across lots and lots of tables. This is what is called the **impedance mismatch**. Application developers have been frustrated with the impedance mismatch between the relational data structures and the in-memory data structures of the application. Using NoSQL databases allows developers to develop without having to convert in-memory structures to relational structures.

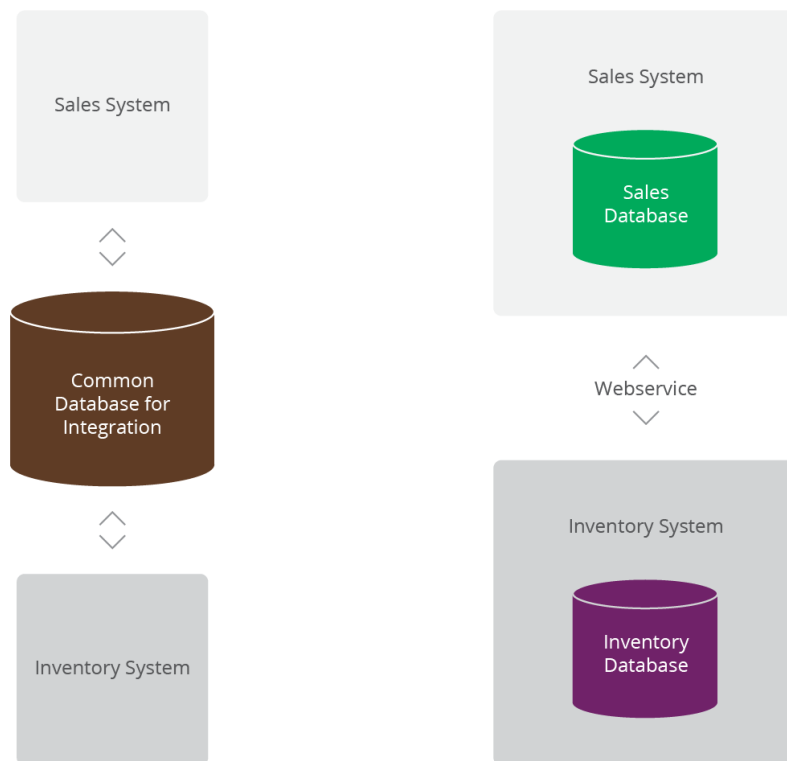


The fact that there are two different models of how to look at things and the fact that both models have to match, causes difficulties. This is what leads to object relation mapping frameworks. This impedance mismatch problem led to the fact that in the mid-nineties people said: "We think relational databases are going to go away and object databases will be replacing them. In that way we can take care of memory structures and save them directly to disk without any of this mapping between the two." But this didn't happen.

Why didn't object databases actually fulfill that potential? At the heart of it, it is the fact that SQL databases had become an integration mechanism through which many people integrated different applications. As a result that really made it very hard for any kind of technology to come in. So relational databases continued to keep being dominant right through the years 2000. As a result relational databases had 20 years of complete dominance of the enterprise databases.

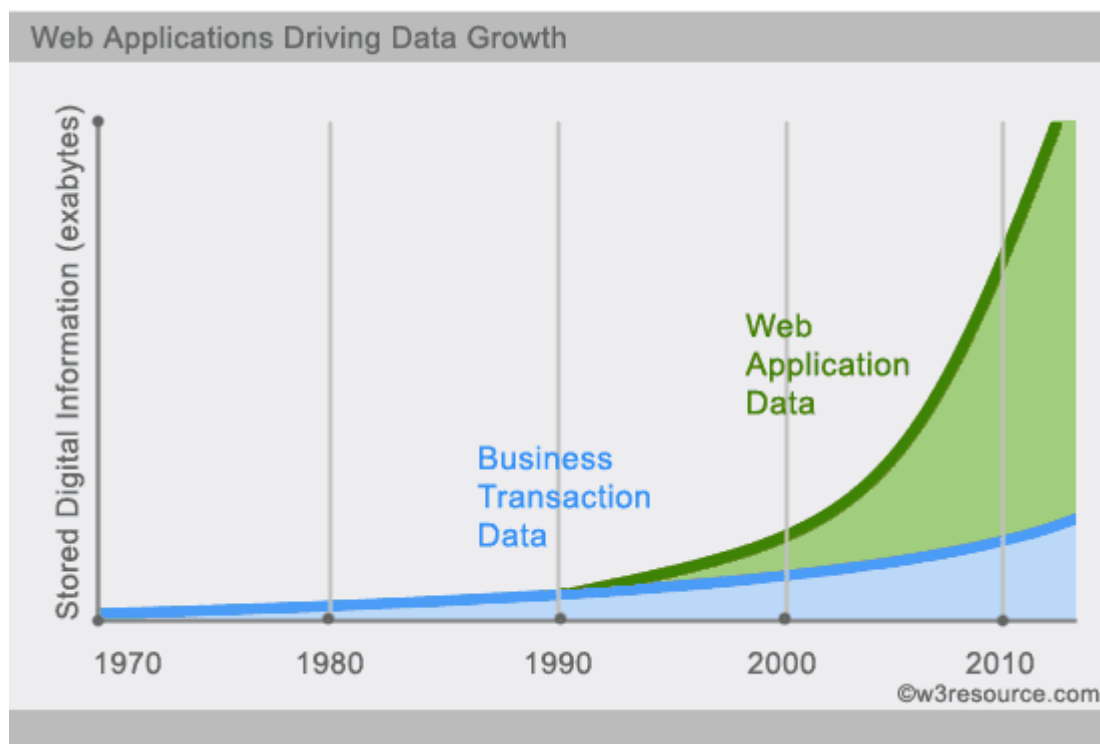


Nowadays though, there is a movement away from using databases as integration points in favor of encapsulating databases using services.



### 3.2 What did change in favor of NoSQL?

The rise of the web as a platform created a vital change in data storage as the need to support large volumes of data became more important. Especially some particular websites have lots and lots of traffic such as Amazon or Google or ...



The data storage needs of an ERP application are lot more different than the data storage needs of a Facebook or an Etsy, for example. Personal user information, social graphs, geo location data, user-generated content and machine logging data are just a few examples where the data has been increasing exponentially. Databases are required to process huge amount of data. Which SQL databases were never designed to. The evolution of NoSQL databases is to handle these huge data properly.

As you get large amounts of traffic coming into your data, you need to scale things. You can do this in two different ways. You got one obvious route: you can scale things up by using bigger boxes. This is being illustrated by the figure below. This approach has problems: it costs a lot and there and there are real limits as to how far you can go.

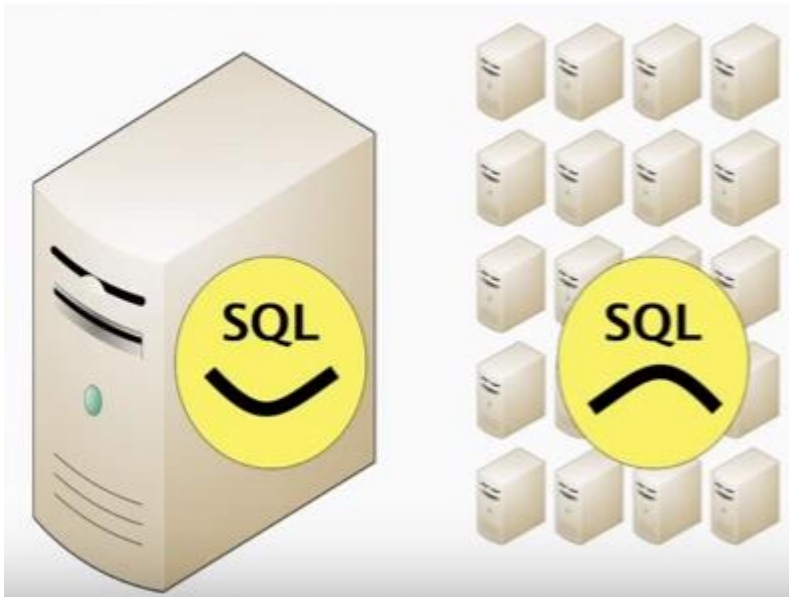


So lots of organizations, most famously Google use a completely different approach: lots and lots of little boxes, just commodity hardware all thrown into these massive grids.



But this approach involves some new problems. Relational databases were not designed to run efficiently on clusters. It does not work very well with large clusters of little boxes and several of the Big Data players understood this. They attempted to spread relational databases and run them across clusters, but it's very

hard to do so. A couple of organizations said "we've had enough of this, we need to do something different" and they developed their own data storage systems that were really quite different from relational databases. They started talking a little bit about them, published papers about it, .... It is this that really inspired a whole new movement of databases which is the NoSQL movement.





## 4 Common characteristics of NoSQL databases [4]

The most common characteristics of NoSQL databases:

- **not – relational:** NoSQL databases are more about non – relational than it is about no – sql
- **open – source:** Most of the NoSQL databases are open source, although there are some commercial tools that like to call themselves NoSQL databases. Maybe over time that will be no longer a common characteristic but it is still a common characteristic at the moment
- **cluster – friendly:** this is the ability to run on large clusters. This is one of the main concepts that drove companies like Google and Amazon towards NoSQL, but it's not an absolute characteristic. There are some NoSQL databases that aren't really focused around running on clusters.
- **21st Century web:** NoSQL databases come out of the 21st century website culture
- **schema – less:** NoSQL databases use different data models to the relational model

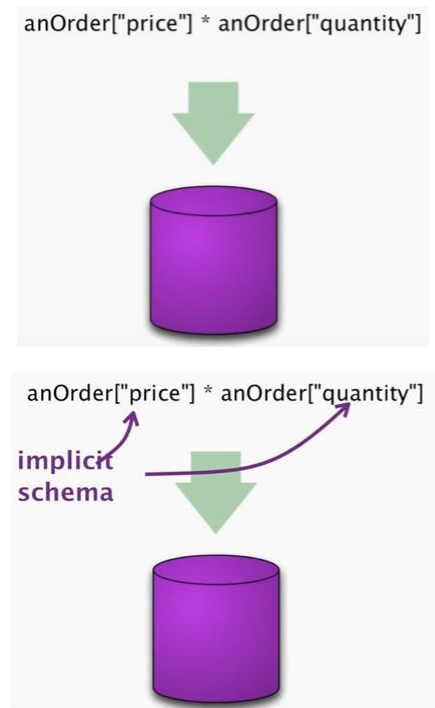
### 4.1 Schema-less ramifications

Almost all NoSQL databases don't tend to have a set schema. With a relational database you can only put the data into the database as long as it fits in the schema of what you've defined for that database. This increases your flexibility. The NoSQL people will talk endlessly about how it makes it easier to migrate data over time. But that's not really the entire truth.

Usually when you're talking to a database you want to get some specific pieces of data out of it, you're going to say: "I would like the price or I would like the quantity or I would like the customer." As soon as you're doing that what you're doing is you're setting up an implicit schema. You are assuming that an order has a price field. You are assuming that the order has a quantity field. You are assuming that the field is called price and not cost or price\_to\_customer or whatever other thing you could think of what it would be. So there is an implicit schema and you've got to manage this implicit schema in many ways in a similar approach to the way that you manage the relational more strict schema. So schema-less is really a bit of a loosely used term here.

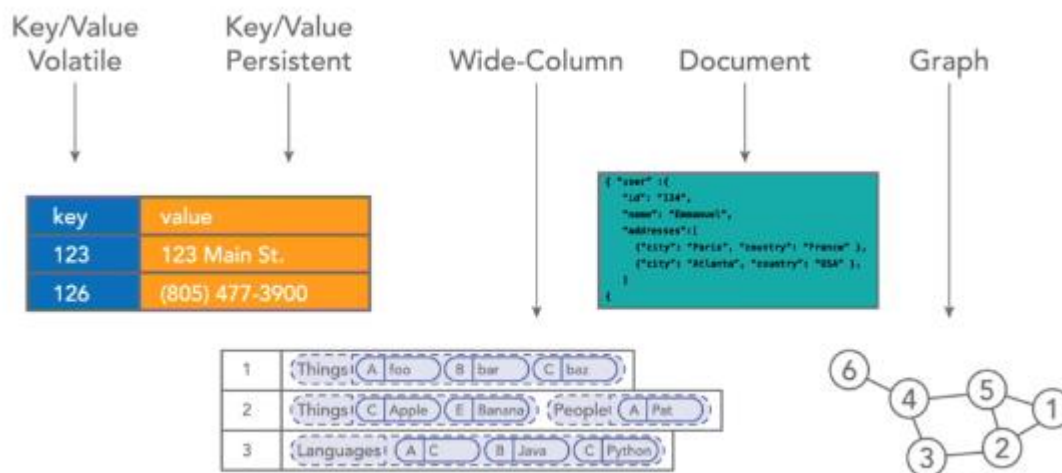
By having no fixed storage schema does give you some options that you don't get with relational databases and there is a difference and there are advantages in terms of flexibility as well but you can't ignore the fact that you are always dealing with an implicit schema.

The only time you don't have to worry about an implicit schema is if you do something like: "Give me all the fields in this record and throw them up on the screen." Occasionally you want to do that but most of the time you actually want to do something more interesting.



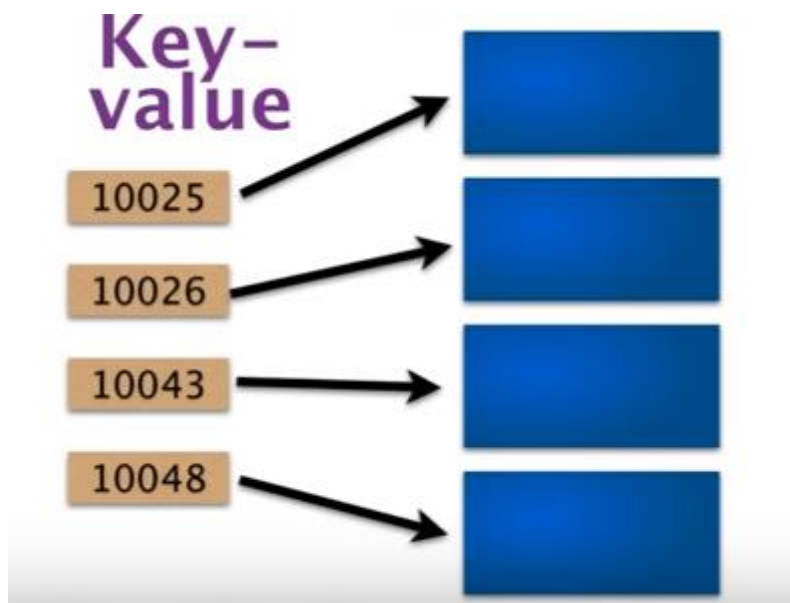
## 5 Types of NoSQL databases [4]

NoSQL databases can broadly be categorized in four types.



### 5.1 Type 1: Key-Value databases

Key-value stores are the simplest NoSQL data stores to use from an API perspective. The basic idea is you have a key. You go to the database and tell: "Grab me the value of the key in the database." The database knows absolutely nothing about what's in that value. It could be a single number, it could be some complex document, it could be an image. The database doesn't know and doesn't care. Basically it's just a hashmap but most of the time persistent on the disk. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.

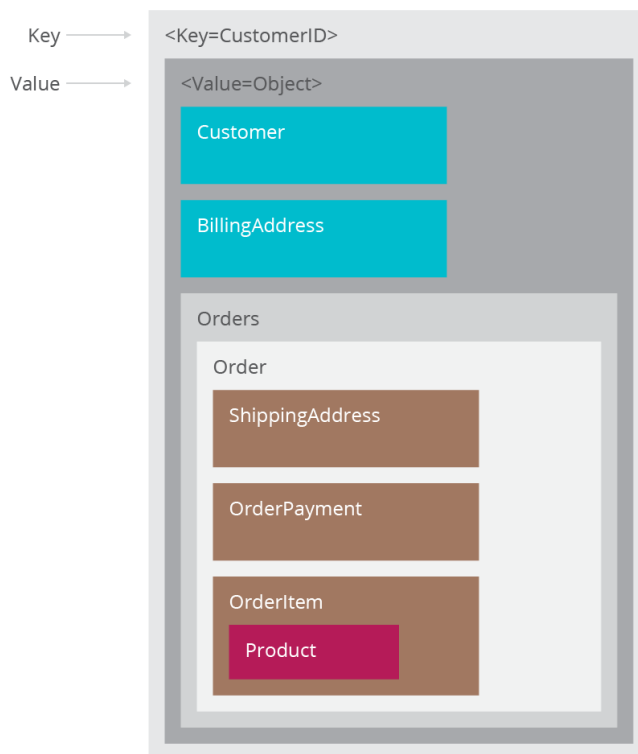


Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.

Some of the popular key-value databases are Riak, Redis (often referred to as Data Structure server), Memcached and its flavors, Berkeley DB, HamsterDB (especially suited for embedded use), Amazon DynamoDB (not open-source), Project Voldemort and Couchbase.

All key-value databases are not the same, there are major differences between these products, for example: Memcached data is not persistent while in Riak it is. These features are important when implementing certain solutions. Let's consider we need to implement caching of user preferences, implementing them in Memcached means when the node goes down all the data is lost and needs to be refreshed from source system, if we store the same data in Riak we may not need to worry about losing

data but we must also consider how to update stale data. It's important to not only choose a key-value database based on your requirements, it's also important to choose which key-value database.



Example worth checking out: <http://try.redis.io/>

In summary

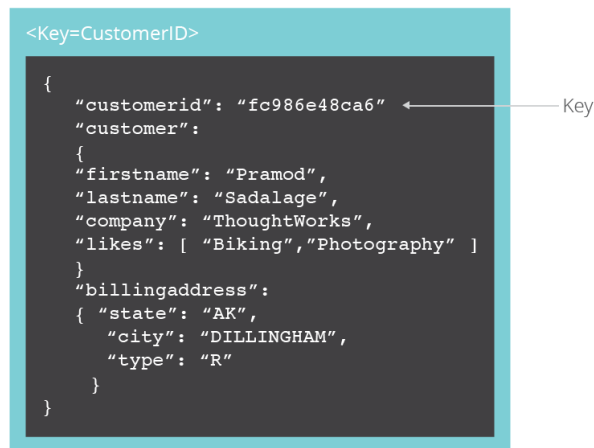
- Key-value stores are most basic types of NoSQL databases. They can be considered as huge lookup tables
- Designed to handle huge amounts of data.
- Based on Amazon's Dynamo paper.
- Key value stores allow developer to store schema-less data.
- In the key-value storage, database stores data as hash table where each key is unique and the value can be string, JSON, BLOB (basic large object) etc.
- A key may be strings, hashes, lists, sets, sorted sets and values are stored against these keys.
- For example a key-value pair might consist of a key like "Name" that is associated with a value like "Robin".
- Key-Value stores can be used as collections, dictionaries, associative arrays etc.
- Key-Values stores would work well for shopping cart contents, or individual values like color schemes, a landing page URI, or a default account number.

## 5.2 Type 2: Document databases

Another data model that is very common is the document data model. A document data model thinks of the database as a storage of a whole mass of different documents where each document is some complex data structure. Usually that data structure is represented in forms of JSON, but XML, BSON, ... is also possible. These documents are self-describing, hierarchical tree data structures which can consist of maps, collections, and scalar values. The documents stored are similar to each other but do not have to be exactly

the same. Document stores don't tend to have a set schema. With a relational database you can only put the data into the database as long as it fits in the schema of what you've defined for that database.

The big difference between a key – value database and a document database is that you can query into the document structure and you can usually retrieve portions of the document or update portions of a document. So in a document database you have these different documents that all flash around and the usual document databases will allow you to say: "Give me a document that has these fields with these values." Document databases such as MongoDB provide a rich query language and constructs such as database, indexes, etc allowing for easier transition from relational databases.



```
<Key=CustomerID>
{
  "customerid": "fc986e48ca6"
  "customer":
  {
    "firstname": "Pramod",
    "lastname": "Sadalage",
    "company": "ThoughtWorks",
    "likes": [ "Biking", "Photography" ]
  }
  "billingaddress":
  {
    "state": "AK",
    "city": "DILLINGHAM",
    "type": "R"
  }
}
```

In terms of uptake and adoption, the document databases have been adopted more widely than any other type of NoSQL database.

Some of the popular document databases we have seen are MongoDB, CouchDB , Terrastore, OrientDB, RavenDB, and of course the well-known and often reviled Lotus Notes that uses document storage.

MongoDB is the number one NoSQL database and it is so because of a number of things. Their tools, and support, and community looks very much like that of a traditional enterprise company. They have user groups, they have trainings, they have online trainings, they have all kinds of a partner ecosystem, they have a maturity around their community that is very important in terms of their adoption.

The boundary between a key – value database and a document database is somewhat blurry. Key – value stores and document databases have this common notion of you taking some complex structure and saving it as a single unit into the database whether it can be in a relatively transparent document (document database) or a completely opaque value (key – value database).

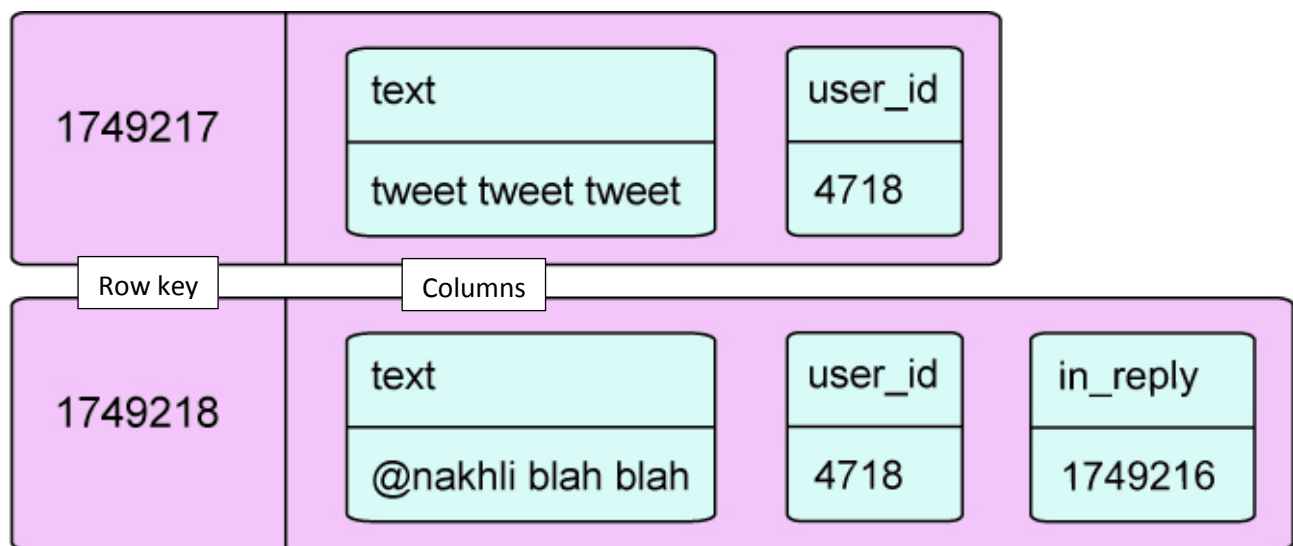
In summary

- A collection of documents
- Data in this model is stored inside documents.
- A document is a key value collection where the key allows access to its value.
- Documents are not typically forced to have a schema and therefore are flexible and easy to change.
- Documents are stored into collections in order to group different kinds of data.
- Documents can contain many different key-value pairs, or key-array pairs, or even nested documents.

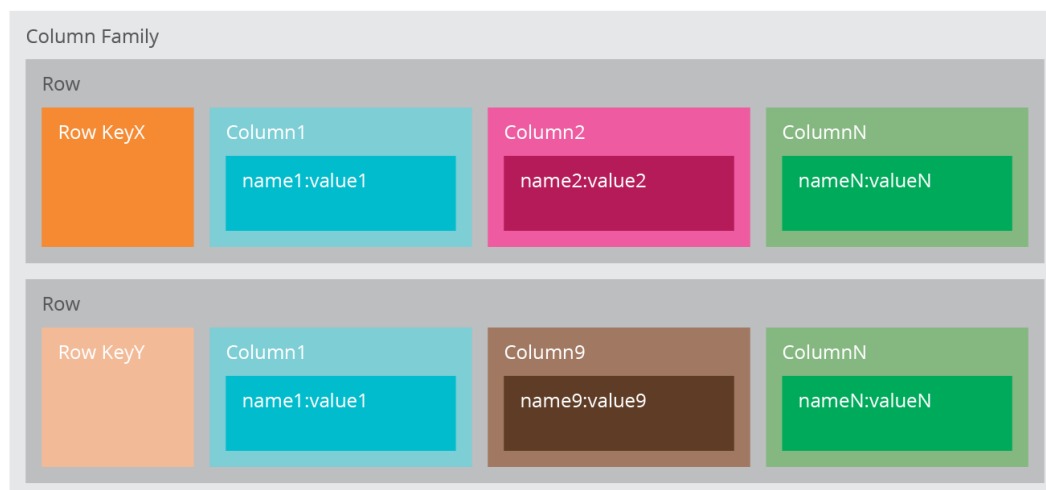
## 5.3 Type 3: Column family databases

Column-family databases store data in column families as rows that have many columns associated with a row key. Column families are groups of related data that is often accessed together. For a Customer, we would often access their Profile information at the same time, but not their Orders.

Each column family can be compared to a container of rows in an RDBMS table where the key identifies the row and the row consists of multiple columns. The difference is that various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows.



When a column consists of a map of columns, then we have a super column. A super column consists of a name and a value which is a map of columns. Think of a super column as a container of columns.



Cassandra is one of the popular column-family databases; there are others, such as HBase, Hypertable, and Amazon DynamoDB. Cassandra can be described as fast and easily scalable with write operations spread across the cluster. The cluster does not have a master node, so any read and write can be handled by any node in the cluster.

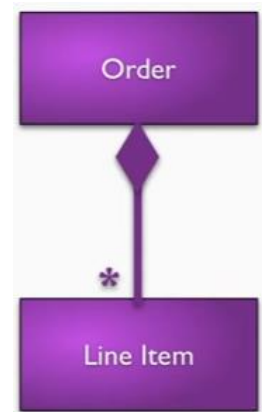
In summary

- Column-oriented databases primarily work on columns and every column is treated individually.
- Values of a single column family are stored contiguously.
- All data within each column family have the same type which makes it ideal for compression.

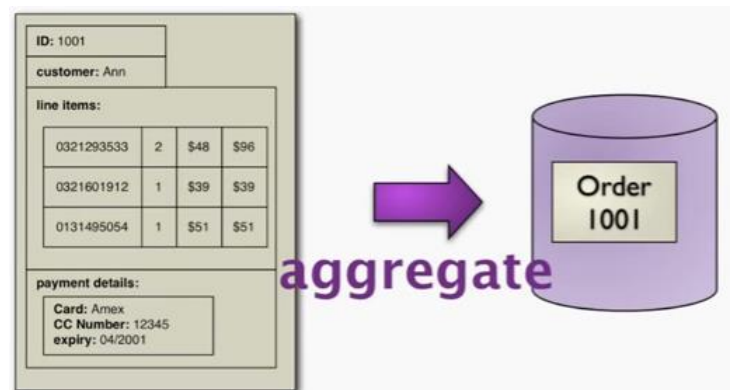
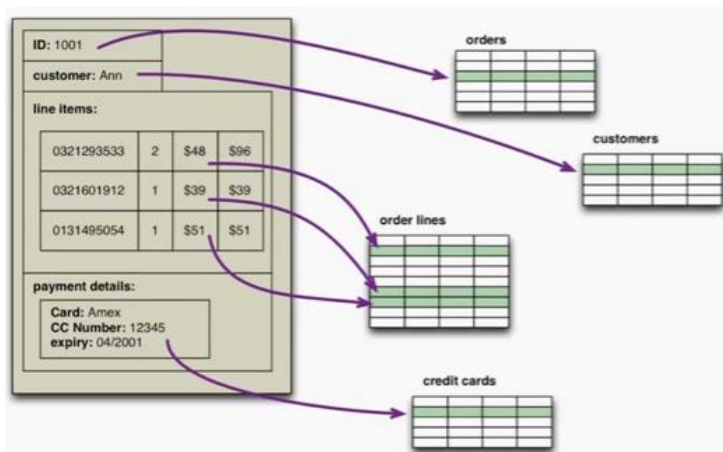
- Column family stores can improve the performance of queries as it can access specific column data.
- High performance on aggregation queries (e.g. COUNT, SUM, AVG, MIN, MAX).
- Works on data warehouses and business intelligence, customer relationship management (CRM), Library card catalogs etc.

## 5.4 Aggregate – Oriented databases

You could describe key – value databases, document databases and column – family databases as Aggregate-Oriented databases that allow you to store these big complex structures. Often when we want to model things we have to group things together into natural aggregates because when we're talking to a database, even a relational database, it makes sense to think of those aggregates when storing and retrieving data. If we're modeling orders for instance, usually we'll have separate classes: the orders and the line items. We think of the order as one thing, a single unit. In a relational database we have to splatter this unit over a whole bunch of tables. In an aggregate – oriented database, we can save that aggregate as a single unit in terms of the database itself.

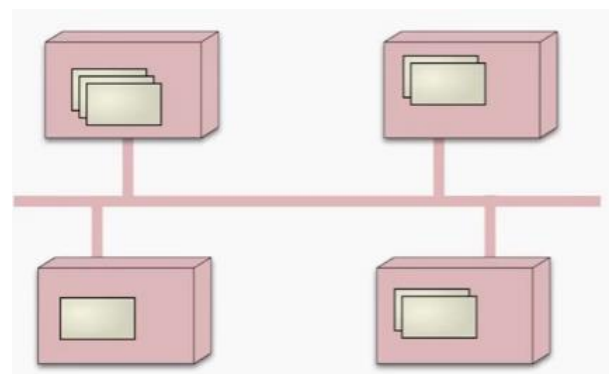


So the great thing about this is, that now when you're taking your aggregate in memory instead of spreading it across lots of individual records you get to store the whole thing in the database in one go and the database knows what your aggregate boundaries are.



Where this becomes really useful is when talking about running the system across clusters because you want to distribute the data. If you distribute data, you want to distribute data that tends to be accessed together and so the aggregate tells you what data is going to be accessed together.

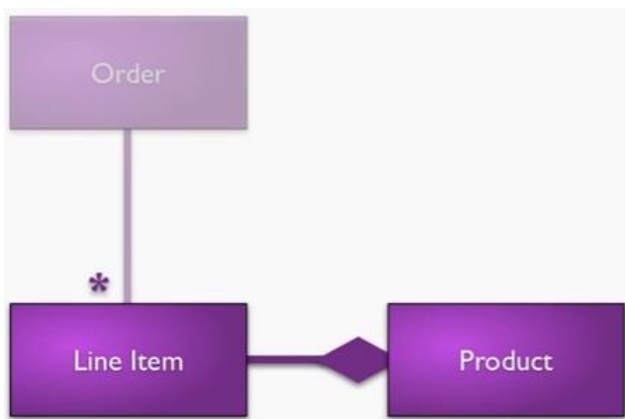
So by placing different aggregates on different nodes across your cluster, you know that when somebody says: "Give me all the details about this particular order", you're only going to one node on the cluster instead of shooting around to pick up different rows from different tables.



So aggregate orientation naturally fits in very nicely with storing data on large clusters.

However, nothing is perfect and aggregate orientation isn't always a good thing. Let's imagine you've got an order system and you want to look at the data and you want to say: "Tell me the revenue or prior revenue of a certain product." In this case you don't care about orders at all. You only care about what's going on with individual line items of many orders grouping them together by product. You want to change the aggregation structure from a structure where orders aggregate line items to a structure where products aggregate line items. The product now becomes the root of the aggregate.

Product	revenue	prior revenue
321293533	3083	7043
321601912	5032	4782
131495054	2198	3187
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...
...	...	...



In a relational database, this is straightforward you just query data differently. It's very straightforward to rearrange the data in the structures you might want. It's more complicated if you use NoSQL.

So aggregates make it easier for the database to manage data storage over clusters, since the unit of data now could reside on any machine and when retrieved from the database gets all the related data along with it. Being aggregate oriented is an advantage if most of the time you're using the same aggregate to push back and forth into persistence. It is a disadvantage if you want to slice and dice your data in different ways.

## 5.5 Distribution Models

Aggregate oriented databases make distribution of data easier, since the distribution mechanism has to move the aggregate and not have to worry about related data, as all the related data is contained in the aggregate. There are two styles of distributing data:

- Sharding: Sharding distributes different data across multiple servers, so each server acts as the single source for a subset of data.
- Replication: Replication copies data across multiple servers, so each bit of data can be found in multiple places. Replication comes in two forms:
  - Master-slave replication makes one node the authoritative copy that handles writes while slaves synchronize with the master and may handle reads.
  - Peer-to-peer replication allows writes to any node; the nodes coordinate to synchronize their copies of the data.



Master-slave replication reduces the chance of update conflicts but peer-to-peer replication avoids loading all writes onto a single server creating a single point of failure. A system may use either or both techniques. Like Riak database shards the data and also replicates it based on the replication factor.

## 5.6 Type 4: Graph databases

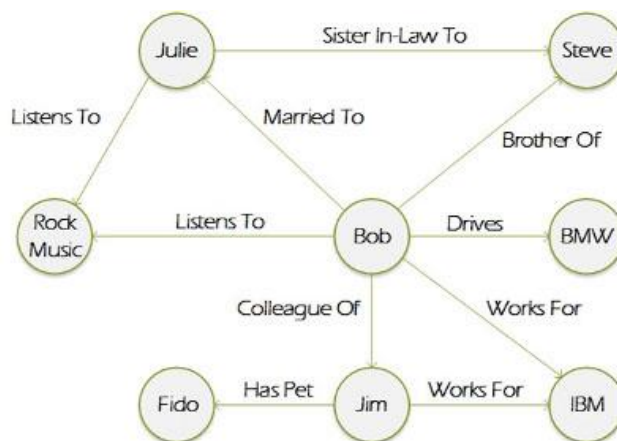
Graph databases are not aggregate oriented at all. This is a completely different data model.

A graph database data model is basically that of a node and arc graph structure, not bar chart or anything like that, but just nodes and arcs. The nice thing about storing a graph database, is that it's very good at handling and moving across relationships between things.

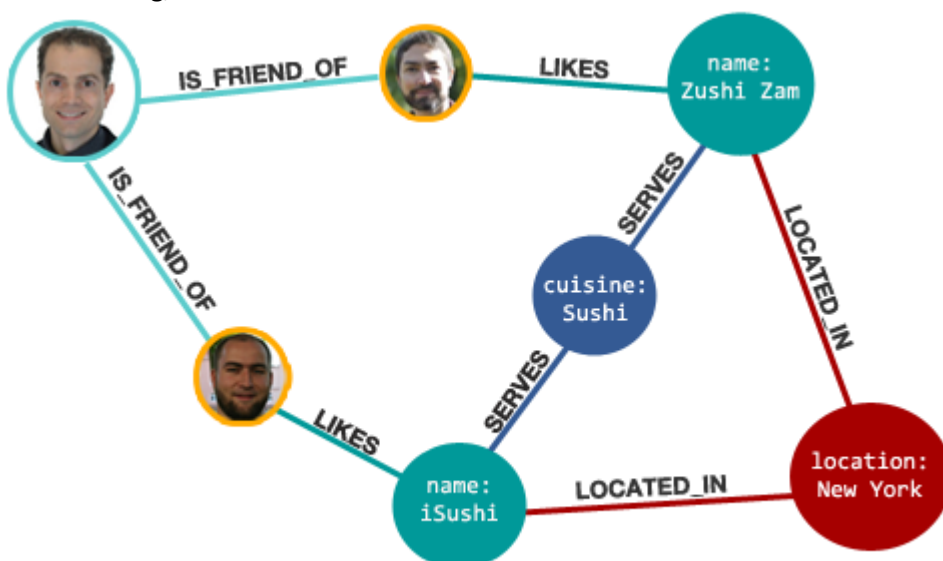
Relational databases – you might think with the word relation in there – that they're good at handling relationships, but relation doesn't mean relationship, it means something in set theory and actually relational database are not good at jumping across relationships. You have to set up foreign keys and you have to do joins, ... If you do too many joins, you can get in a mess.

Graph databases are good in relationships. You can jump around relationships, left, right and center and they are optimized to do it fast. Furthermore they can come up with an interesting query language that is designed around, allowing you to query graph structures. You can do some very interesting graph oriented queries in graph databases. Things that would be very, very difficult to write in terms of SQL and that would be very hard in terms of performance.

In graph databases, traversing the joins or relationships is very fast. The relationship between nodes is not calculated at query time but is actually persisted as a relationship. Traversing persisted relationships is faster than calculating them for every query.



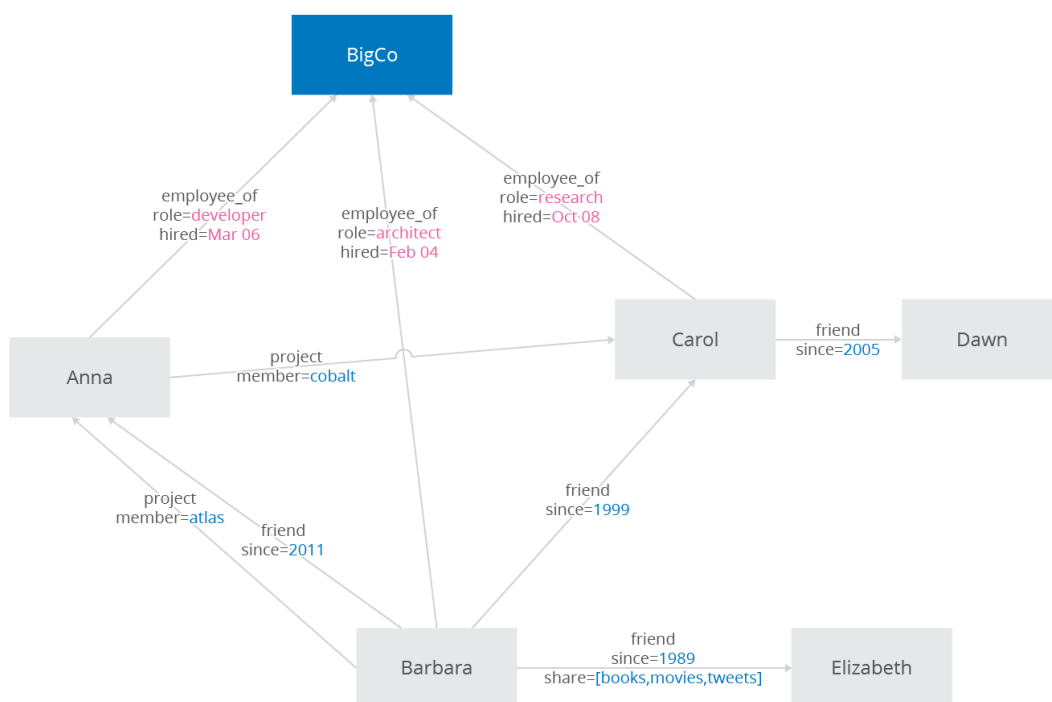
Nodes can have different types of relationships between them, allowing you to both represent relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access. Since there is no limit to the number and kind of relationships a node can have, they all can be represented in the same graph database.





Relationships are first-class citizens in graph databases; most of the value of graph databases is derived from the relationships. Relationships don't only have a type, a start node, and an end node, but can have properties of their own. Using these properties on the relationships, we can add intelligence to the relationship—for example, since when did they become friends, what is the distance between the nodes, or what aspects are shared between the nodes. These properties on the relationships can be used to query the graph.

Since most of the power from the graph databases comes from the relationships and their properties, a lot of thought and design work is needed to model the relationships in the domain that we are trying to work with. Adding new relationship types is easy; changing existing nodes and their relationships is similar to data migration, because these changes will have to be done on each node and each relationship in the existing data.



So graph databases are used for data that has a lot of many-to-many relationships, where the relationships are the key you need to understand: when not only the data in the nodes is important, but the relationships between the nodes are also important and when your primary objective is quickly finding connections, patterns, and relationships between the objects.

There are many graph databases available, such as Neo4J, Infinite Graph, OrientDB, or FlockDB (which is a special case: a graph database that only supports single-depth relationships or adjacency lists, where you cannot traverse more than one level deep for relationships).

Example worth checking out: <http://console.neo4j.org/>

In summary

- A graph database stores data in a graph.
- It is capable of elegantly representing any kind of data in a highly accessible way.
- A graph database is a collection of nodes and edges
- Each node represents an entity (such as a student or business) and each edge represents a connection or relationship between two nodes.

- Every node and edge is defined by a unique identifier.
- Each node knows its adjacent nodes.
- As the number of nodes increases, the cost of a local step (or hop) remains the same.

## 5.7 Overview



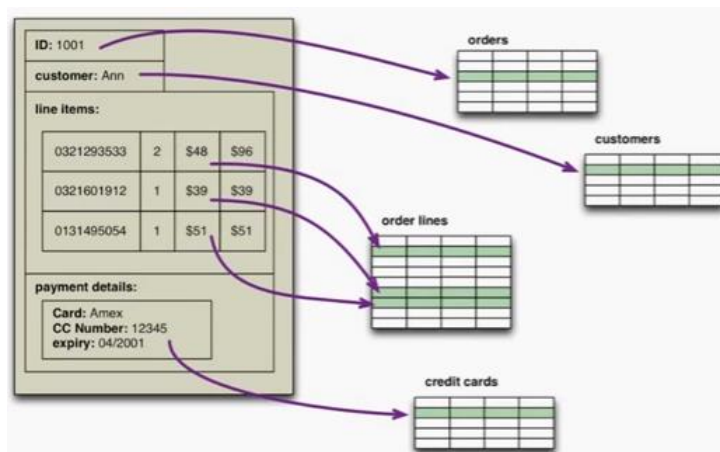
## 6 NoSQL and consistency [4]

### 6.1 Introduction

The previous chapter talked about the data model. This chapter is about another issue: consistency and effectively dealing with lots of people trying to modify the same data at the same time.

Everyone knows that relational databases are ACID. They do the familiar acid transactions that everyone knows: atomic, consistent, isolated, durable. NoSQL doesn't do any of that kind of things.

So basically what it boils down to is, if you've got a single unit of information and you want to split it across several tables, you don't want to be caught in a position where you only get to write half of the data and somebody else reads it. Or you get to write half of the data and somebody else takes the same order and writes a different half of the data and things get really messy. In that kind of situation, you need to have this mechanism to control to effectively give you atomic updates and that's really what



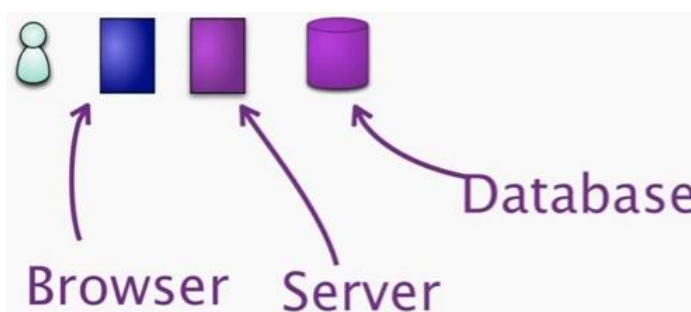
transactions are all about. Atomic updates so that you either succeed or fail and nobody comes in the middle and messes things up. When it comes to NoSQL databases. The first thing to point out is graph databases do tend to follow ACID updates, which makes sense. They decompose the data even more than relational databases do, so they've got even more need to make sure they use transactions to wrap things together. So if anybody says that NoSQL databases don't do ACID, that's not entirely true.

Aggregate oriented databases actually don't need transactions as much, because the aggregate is a kind of bigger, more richer structure. In fact the domain driven design community said from the beginning – even before NoSQL – "Keep your transactions within a single aggregate". That's effectively what you do in the world of aggregate oriented databases.

Any aggregates update is going to be atomic, it's going to be isolated, it's going to be consistent within itself. It's only when you update multiple documents, that you have to worry about the fact that you haven't got ACID transactions, but that problem occurs much more rarely than you would expect.

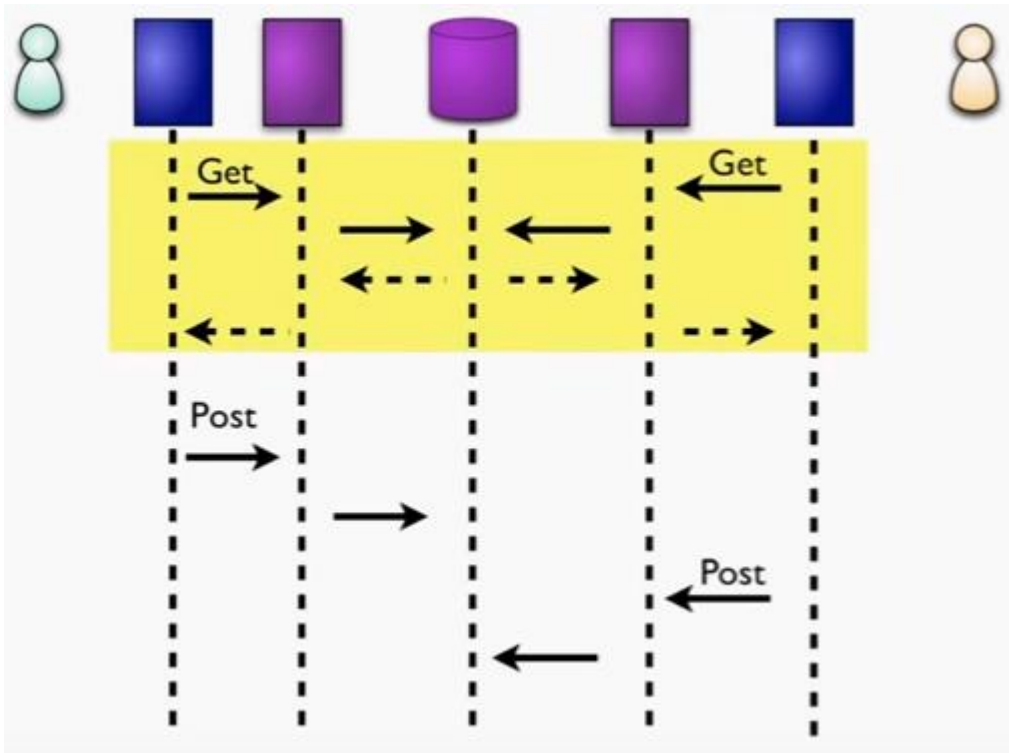
Even in a relational world ACID transactions don't mean we get to be completely consistent and don't have to worry about update anomalies.

Imagine we have some typical multi-layered system: we've got a person talking to a browser. The browser talks to the server, the server talks to a single database. There are 2 people talking to the same database at the same time, although through different browsers and servers.

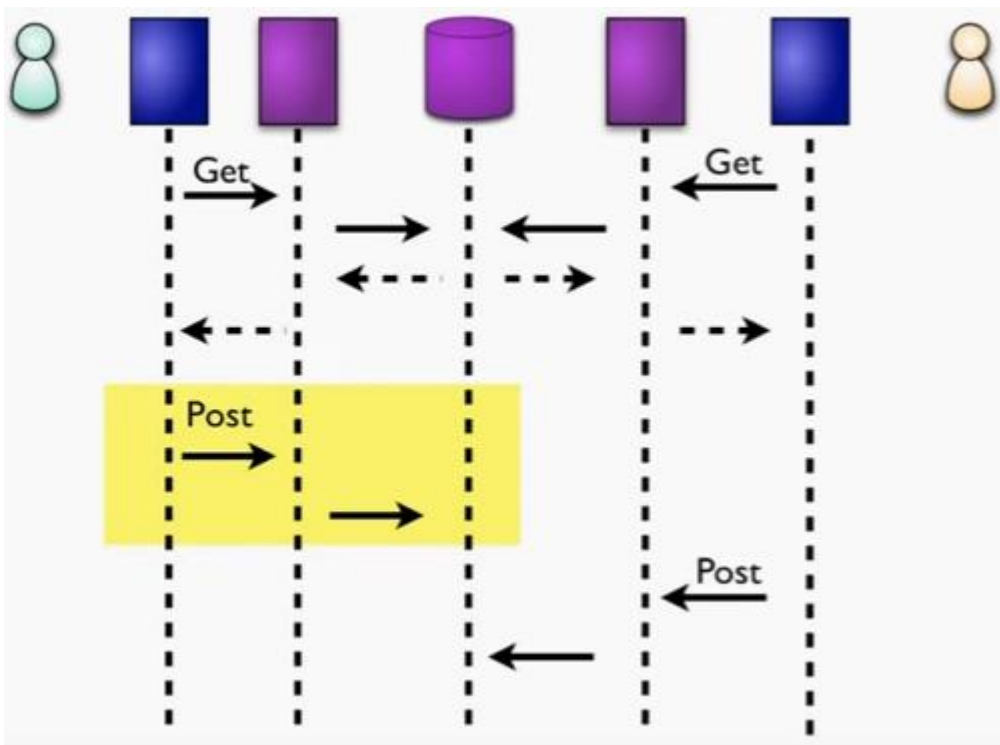




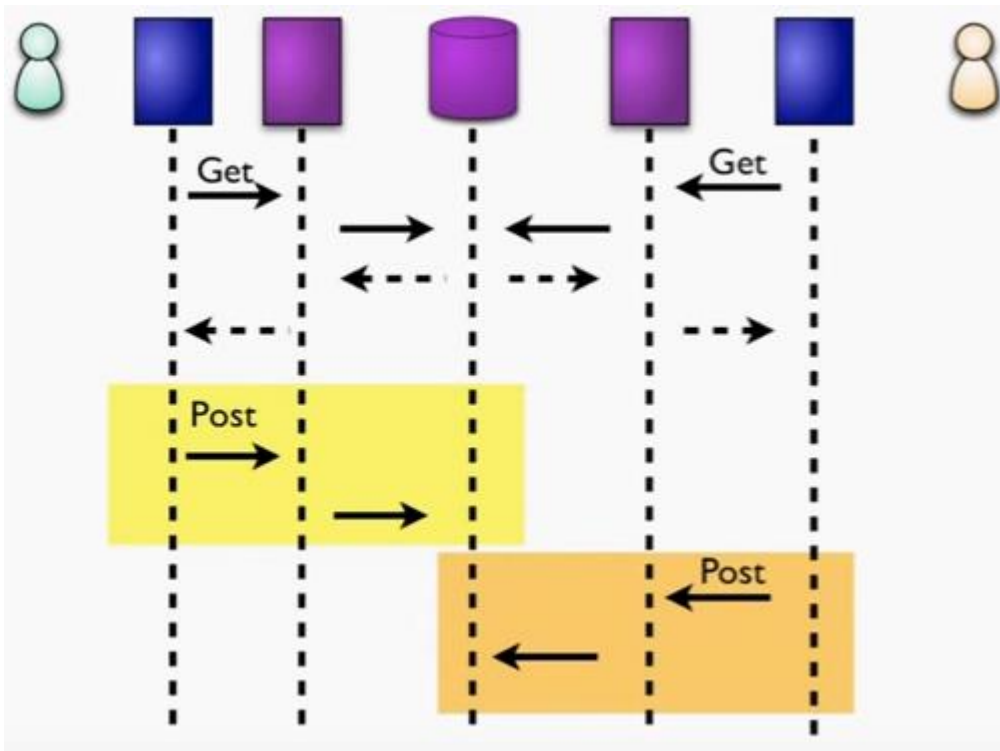
Both people left and right take the same piece of data with a GET request. Essentially they bring it up onto their browser screen and both human beings think "I need to make some changes to this".



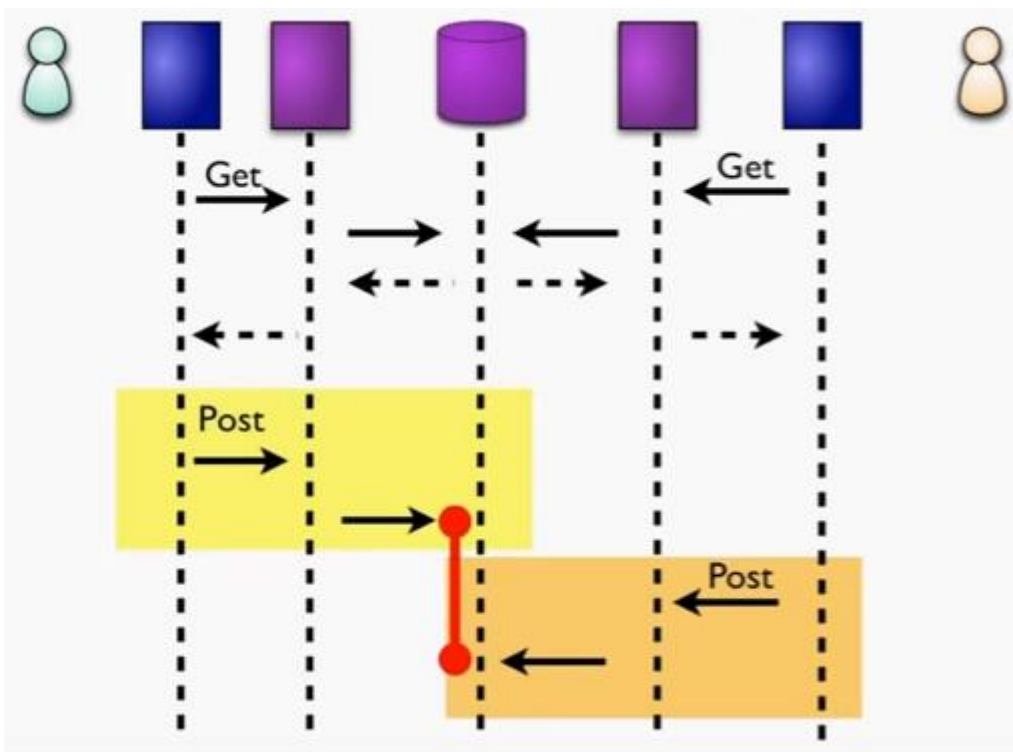
Eventually the guy on the left says: "Okay I've got my updated data, let's post some changes."



Then shortly afterwards the guy on the right says: "I've updated my data now. Let's post some changes now."



Of course if you let this happen just like that, this is a write write conflict: two people have updated the same piece of information. They weren't aware of each other's update and they've got themselves in trouble.



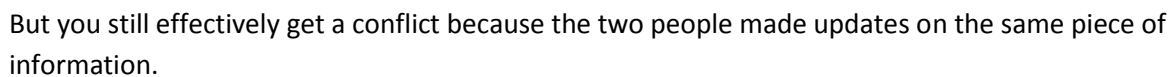
The diagram illustrates a distributed transaction protocol between two nodes (represented by blue squares) and a central database (represented by a purple cylinder). The nodes are connected to the database via dashed lines. The sequence of events is as follows:

- Node 1 (Left):** Initiates a transaction by sending a **Get** message to the database.
- Database:** Responds to Node 1 with a **Post** message.
- Node 2 (Right):** Initiates a transaction by sending a **Get** message to the database.
- Database:** Responds to Node 2 with a **Post** message.
- Node 1 (Left):** Sends a **Post** message to the database.
- Database:** Responds to Node 1 with a **Get** message.
- Node 2 (Right):** Sends a **Get** message to the database.
- Database:** Responds to Node 2 with a **Post** message.

The diagram also includes a green bracket on the left side, indicating a time period, and a clock icon, suggesting a timeline or duration of the transaction.

22

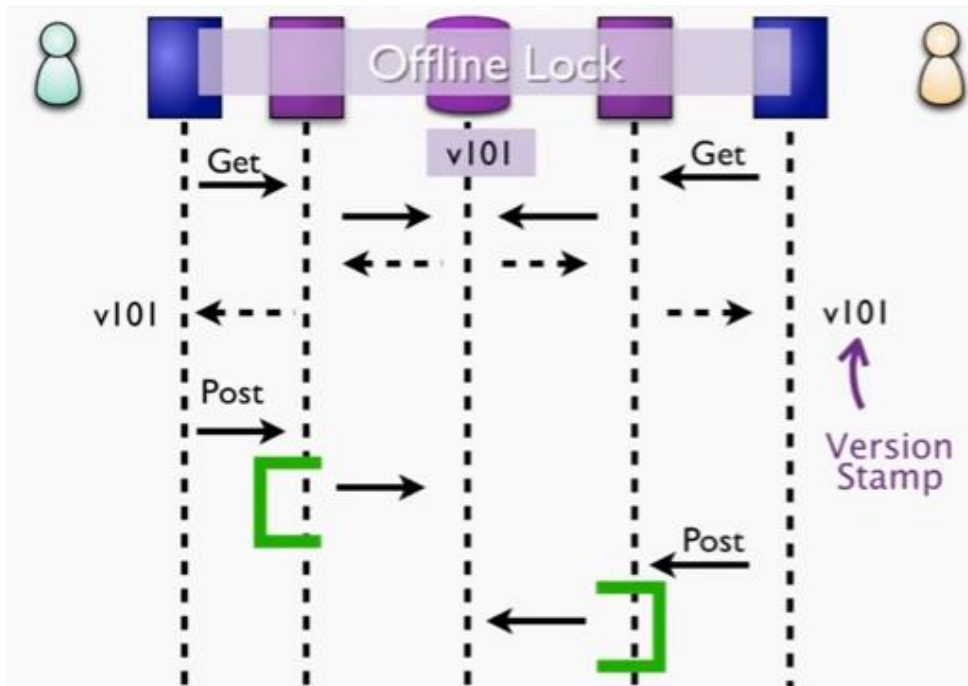
Instead just wrap the transaction around the update. This stops a collision in which case you would get some tables updated over here and some different tables updated differently over there and the result would be an inconsistent mess.



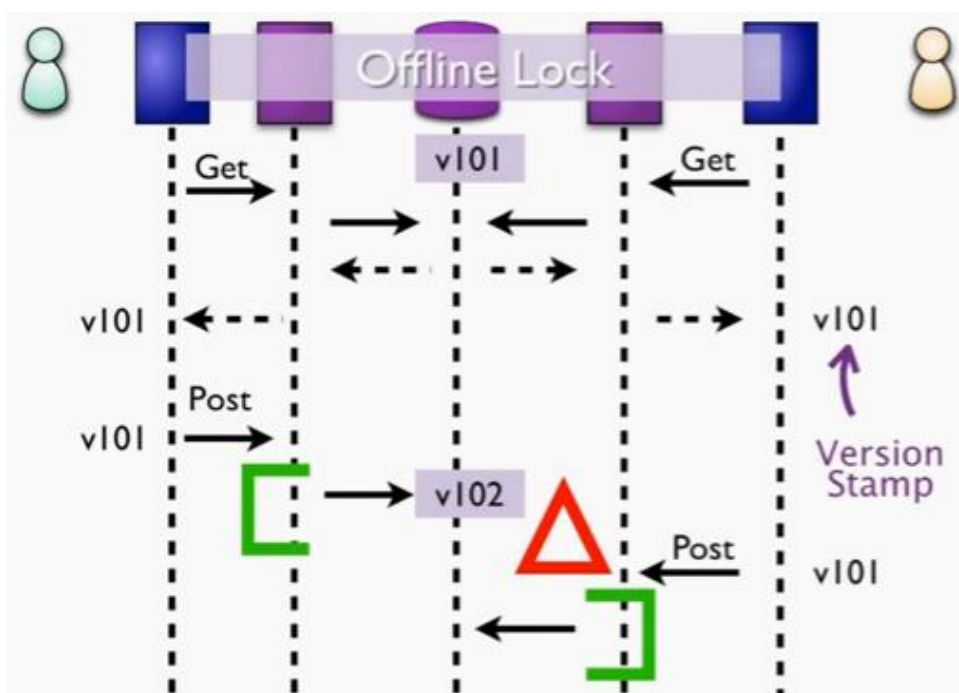


This is what typically might happen even in aggregate oriented databases if you have to modify more than one aggregate. Because you might find one person modifies the first one and then he goes over the second one and the other person does it the other way around and as a result this could lead into an inconsistency between aggregates.

If you come across this, you can solve this and basically use a technique which is referred to as an **offline lock**. Basically what that means: you give each data record or each aggregate a version stamp and when you retrieve it, you retrieve the version stamp with the aggregate data.



When you post, you provide the version stamp of where you read from and then for the first guy everything works out okay and the version stamp gets incremented. When the second person tries to post and he still got the old version stamp, then you know something's up and you can do whatever conflict resolution approach that you take.





You can use the same basic techniques with NoSQL databases.

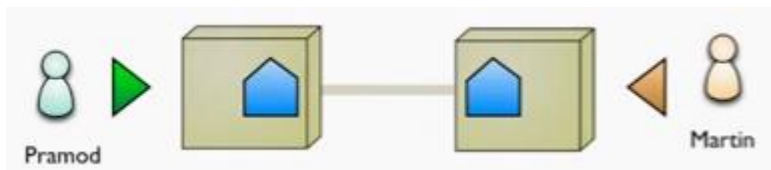
The nice thing is, you don't have to worry about transactions because the aggregate gives you that natural unit of update: it is your transaction boundary. But once you cross aggregates, then you've got to think about juggling version stamps and doing something of that kind. But it's not really very different to what you have to do with a relational database because offline locks force you to do this juggling with version stamps anyway.

## 6.2 Consistency and availability

When we talk about consistency, it is useful to think about two kinds of consistency: the consistency described in the previous paragraphs is called **logical consistency**: these consistency issues occur whether you're running on a cluster of machines or whether you're running on one single machine. You always have to worry about these kinds of consistency issues. Now when you start spreading data across multiple machines, this can introduce more problems. When it comes to distributing data, you can talk about it in two different ways: one is **sharding** data, this means taking one copy of the data and putting it on different machines so that each piece of data lives in only one place but you're using lots of machines. Sharding doesn't really change the picture very much. You still get the same logical consistency problems that you do with a single machine. They are exacerbated to some degree but the basic problems are still the same.

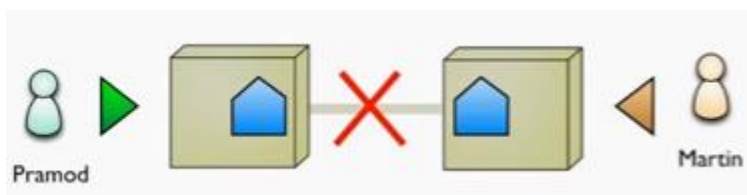
Another thing however that's common to do with clusters of machines is to **replicate** data, this means to put the same piece of data in lots of places. This can be advantageous in terms of performance because now you've got more nodes handling the same set of requests. It can also be very valuable in terms of resilience: if one of your nodes goes down the other replicas can still keep going. So these cluster oriented approaches are good for availability and resilience. However as soon as you replicate data a new class of consistency problems starts coming in. This will be illustrated with a simple example.

There are two people: Pramod and Martin and they happen to be on different continents. Pramod is in India and Martin is in the US. They both want to book the same hotel room and so they send out a booking request. They send their requests to local processing nodes.

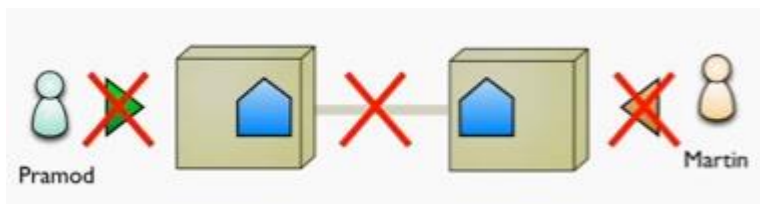


The processing nodes at this point need to communicate. The system as a whole needs to come up with some kind of decision, essentially ensuring that only one of both can book the hotel room.

Let's take a kind of variation on this example. Again they both want to book a hotel room but now the communication line has gone down: the two nodes cannot communicate. They send out their requests.



There's two 2 alternatives of what's going to happen. Alternative one is the system tells the users that the communication lines have gone down, so they can't take the hotel bookings at the moment, "Please try again later".



The alternative is the system accepts both bookings and the hotel room is double booked.



What's been illustrated here, is a choice between **consistency**, which means "No I'm not going to do anything while the communication lines are down" and **availability**, which says "Yes I'm going to keep on going but at the risk of introducing an inconsistent behavior". A vital thing here to realize is that this is a choice and it's a choice that can only be made by knowing about the business rules. It is the business people who will have to decide what's more important: the risk of double booking the last room in the hotel or the fact that you have to bring down the site and say "sorry we can't accept any orders at the moment" which is kind of bad for business.

## 6.3 CAP theorem [3] [2]

You must understand the CAP theorem when you talk about NoSQL databases or in fact when designing any distributed system. CAP theorem states that there are three basic requirements which exist in a special relation when designing applications for a distributed architecture.

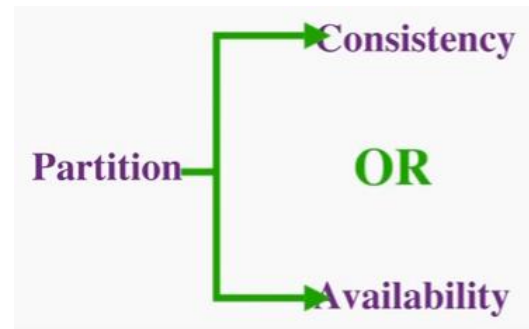
**Consistency:** This means that the data in the database remains consistent after the execution of an operation. For example after an update operation all clients see the same data.

**Availability:** This means that the system is always on (service guarantee availability), no downtime.

**Partition Tolerance:** This means that the system continues to function even when the communication among the servers is unreliable, i.e. the servers may be partitioned into multiple groups that cannot communicate with one another.

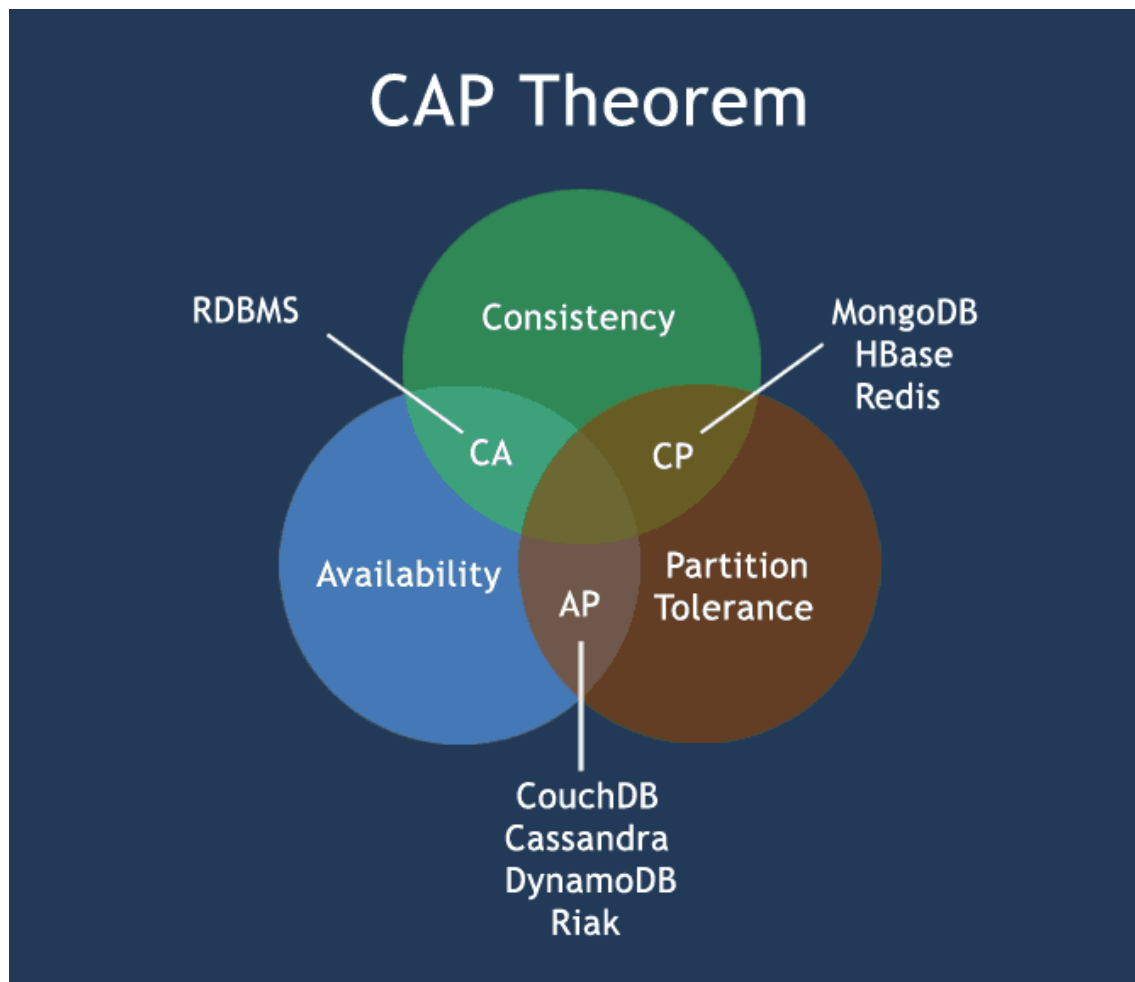
Eric Brewer put forth the CAP theorem which states that in any distributed system you can choose only two of consistency, availability or partition tolerance. So, essentially you have three concepts: Consistency, Availability and Partition Tolerance and you get to pick any two out of three.

It's a bit clearer if you say: If you've got a network partition (e.g. in a distributed system) you have a choice: do you want to be consistent or do you want to be available. That's really what the CAP theorem boils down to. If you have a single database running on a single server, then it's not going to be partitioned. In this case you don't have to worry: the database will be available and consistent.



As soon as you have a distributed system you have to make a choice. Although that isn't a single binary choice. You actually have a spectrum, you can actually trade off levels of consistency and availability.

A lot of the time what you are actually doing is you're trading off consistency versus response time because the more consistency you want to have across a cluster of nodes, the more nodes that have to get involved in the conversation. Think of the hotel case: the 2 had to communicate, that's going to slow down the response time. Even with the network up, you still can decide to do all the communication regarding the consistency afterwards, because you want a faster response time. That's a business decision.



Many NoSQL databases try to provide options where the developer has choices by which they can tune the database as per their needs. For example if you consider Riak which is a distributed key-value database. There are essentially three variables  $r$ ,  $w$ ,  $n$  where

- $r$ =number of nodes that should respond to a read request before its considered successful.

- $w$ =number of nodes that should respond to a write request before its considered successful.
- $n$ =number of nodes where the data is replicated aka replication factor.

In a Riak cluster with 5 nodes, we can tweak the  $r$ ,  $w$ ,  $n$  values to make the system very consistent by setting  $r=5$  and  $w=5$  but now we have made the cluster susceptible to network partitions since any write will not be considered successful when any node is not responding. We can make the same cluster highly available for writes or reads by setting  $r=1$  and  $w=1$  but now consistency can be compromised since some nodes may not have the latest copy of the data. The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.

NoSQL databases provide developers with lots of options to choose from and fine tune the system to their specific requirements. Understanding the requirements of how the data is going to be consumed by the system, questions such as is it read heavy versus write heavy, is there a need to query data with random query parameters, will the system be able to handle inconsistent data.

Understanding these requirements becomes much more important, for long we have been used to the default of RDBMS which comes with a standard set of features no matter which product is chosen and there is no possibility of choosing some features over others. The availability of choice in NoSQL databases, is both good and bad at the same time. Good because now you have a choice to design the system according to the requirements. Bad because now you have a choice and you have to make a good choice based on requirements and there is a chance where the same database product may be used properly or not used properly.

An example of features provided by default in RDBMS is transactions. We are so used to this feature that we have stopped thinking about what would happen when the database does not provide transactions. Most NoSQL databases do not provide transaction support by default, which means the developers have to think how to implement transactions: "Does every write have to have the safety of transactions or can the writes be segregated into "critical that they succeed" and "it's okay if I lose this write" categories." Sometimes deploying external transaction managers like ZooKeeper can also be a possibility.

## 7 Summary

### 7.1 Polyglot persistence [2]

All the choice provided by the rise of NoSQL databases does not mean the demise of RDBMS databases. We are entering an era of polyglot persistence, a technique that uses different data storage technologies to handle varying data storage needs. Polyglot persistence can apply across an enterprise or within a single application.

### 7.2 Why you might want to use a NoSQL database? [4]

There are 2 drivers that push us towards NoSQL databases

#### 7.2.1 Large scale data

You've got more data than you can comfortably or economically fit into a single database server. You can try to run a relational database across the cluster or you can go into NoSQL. Running relational databases across clusters is still somewhat the black card.

Big amounts of data is a big issue. One thing heard is: "Only very few organizations have to worry about this stuff. If you are Google and Amazon, in that case, yes, but pretty much everybody else no." Reality is there is tons of data coming at us and every organization is going to be capturing and processing more and more data. So this large-scale data problem, it's only going to grow.

#### 7.2.2 Easier development

People want to develop more easily. For example, someone who works for a newspaper and website. They're dealing with articles. They're saving articles, updating articles, push articles back and forth. The article to them is a natural aggregate. Spreading that article's data and metadata across the relational databases is awkward, but taking it as a single article and pushing it into the database, that's much more straightforward. The impedance mismatch problem is drastically reduced if you've got a natural aggregate. Because a data model doesn't really fit very well with relational databases, a NoSQL database is better because of the aggregated oriented nature except for the graph databases.

## 8 Exercises

### 8.1 What type of database would you use? [5]

- (1) If your application needs complex transactions because you can't afford to lose data or if you would like a simple transaction programming model.
- (2) If your application needs to handle lots of small continuous reads and writes, that may be volatile
- (3) If your application needs to implement social network operations
- (4) If your application needs to operate over a wide variety of access patterns and data types
- (5) If your application needs to operate on data structures like lists, sets, queues
- (6) If your application needs programmer friendliness in the form of programmer friendly data types like JSON, HTTP, REST, JavaScript
- (7) If your application needs to dynamically build relationships between objects that have dynamic properties
- (8) If your application needs to cache or store BLOB data
- (9) If your application needs to log continuous streams of data that may have no consistency guarantees necessary at all
- (10) If your application needs powerful offline reporting with large datasets

### 8.2 Case studies [6]

#### 8.2.1 Is my hardware under-utilized?

##### Problem

- The source data is in Log files.
- All data is kept and archived
- Variety: The format of the data is CSV. The data is also structured: an identifier and an event. So no variety in the data at all
- Volume: 1 TB per year (in terms of big data this is small big data)
- Velocity: It is produced each minute. It's not being analyzed at all other than on an exception basis.
- The data can be stored on the public cloud
- You only need to have simple aggregate queries going against this log data: you need min, max, count in order to know how the hardware is being utilized because you can figure out by aggregates against those logs when hardware is not utilized. You don't need to do any predictive queries or anything around machine learning.

##### Solution

A NoSQL Key-value or Wide column store, and the reason for this is the storage of larger amounts of data is much, much cheaper than storing it on a relational system, and even though the customer only has 1 TB of data right now, the idea is that you want to build a system that can grow over time.

As they start to make use of this data, and query it more frequently, they're going to need a system that's performant as data volumes increase, and as query intensity increases. Selection of a Key-value or a Wide column depends on query complexity. An important point is to understand if simple aggregation can answer that business question or if you need complex grouping and sub-queries and complexity around the queries. In the case of more complex query types, a Wide column store would be preferred.

## 8.2.2 Storing customer or donor information

### Problem

Companies want to understand what the characteristics are of their top customers, donors, suppliers, so on and so forth.

- The type of data is customer data.
- Variety: It's currently stored in a MySQL database. It's structured, it's relational, it's in tables. So basically, no variety in data: it's relational data.
- Volume: 5 gigabytes per year. Which is a small amount of data for a big data project. The data growth is five percent year over year. So very slow growing. There is a planned acquisition which is going to really increase the amount of data worked with. It's going to more than triple it, and possibly even more.
- The data can be stored on the public cloud.
- Real-time queries are important : to have real-time information when the customer calls up or when they get in touch

### Solution

A document database, or we could potentially look at Hadoop with libraries.

Just looking at the five gigabytes of MySQL data, it would be very tempting just to say "Stay with MySQL." However, because of the potential for high growth because of acquisitions, there could be a more cost-effective solution by looking at NoSQL document database, or Hadoop depending on the volume.

## 8.2.3 Linking museum data

### Problem

This question is specific to the domain of art museums. Where can patrons view works by families of painters?

- The source data is art museum data. It's captured in a variety of locations, specifically in the different art museums around the world.
- This data has a particular format. It's semi-structured and it's what's called linked data. It's stored in a format of a triple store, which is a subject, predicate, object.
- Volume: 2 TB world-wide
- The data growth is really small, because art museums don't acquire a huge amount of paintings all at once. It's a little bit at a time.
- The data can be on the public cloud.
- So the data is a predictable volume, and it's really not very big at all. It's a small volume, small velocity, and no variety.

### Solution

This is a classic scenario for using a NoSQL graph database. The only thing that you need to think about in this case, is whether the graph databases should be local in each museum location or owned by each museum, or if there should be a set of hosted graph databases that are hosted on a public cloud with some sort of management to them.

if you think about the resources in an art museum, they probably don't have a lot of IT people and it's highly unlikely they have somebody who's an expert in graph databases. So if you can offload some of the management and scaling to a cloud vendor that's going to provide those types of services, and also monitoring, that's going to really help them in terms of the usability of the data.



## 9 Sources

- [1] „<https://spring.io/understanding/NoSQL>,“ 2016. [Online].
- [2] <https://www.thoughtworks.com/insights/blog/nosql-databases-overview>. [Online]. [Geopend 2016].
- [3] „<http://www.w3resource.com/mongodb/nosql.php>,“ [Online]. [Geopend 2016].
- [4] M. Fowler, „[https://www.youtube.com/watch?v=ql\\_g07C\\_Q5I](https://www.youtube.com/watch?v=ql_g07C_Q5I),“ [Online]. [Geopend 2016].
- [5] „<http://highscalability.com/blog/2011/6/20/35-use-cases-for-choosing-your-next-nosql-database.html>,“ [Online]. [Geopend 2016].
- [6] „<http://www.lynda.com>,“ [Online]. [Geopend 2016].