HoGent

BEDRIJF
EN
ORGANISATIE

NoSQL

Chapter 1

HoGent

Understanding NoSQL

HoGent

HoGent

Understanding NoSQL

- NoSQL = Not Only SQL
  - NoSQL data stores respond to key data needs that are not met by relational databases
  - Relational databases are not dismissed
- NoSQL databases
  - Usually do not have a schema
  - They come with looser consistency models
  - 4 categories
    - Key-value stores: allow quick lookups (~hashmap in persistency)
    - Document stores: allow to store JSON objects, …
      - Relational databases have rigid schemas
      - Document stores are more flexible
    - Column family databases
    - Graph databases

NoSQL what does it mean

- NoSQL = Not Only SQL = There is more than one storage mechanism that could be used based when designing a software solution
- 1998: Carlo Strozzi used the term to name his Open Source, Light Weight database which did not have an SQL interface
- 2009: Eric Evans reused the term as a twitter hashtag (#nosql) for a conference in Atlanta about databases which are non-relational, distributed, and do not conform to atomicity, consistency, isolation, durability

NoSQL what does it mean

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

Understanding NoSQL

- Faster access to bigger sets of data
  - Relational databases sacrifice performance when searching large volumes of data.
  - The bigger the result set, the more expensive the queries become
  - NoSQL databases
    - Large volumes
    - Large rates of data growth
- Less rigid consistency requirements
  - Certain data requirements don't demand the rigid ACID model
  - Use of ACID => usually worse performance
  - Eventual consistency => usually better performance

Understanding NoSQL

- SQL
  - 40 years old => very mature
  - Switching from 1 relational database to another is much easier than switching between 2 NoSQL databases
- Each NoSQL database has unique aspects
  - The developer must invest time and effort to learn the new query language and the consistency semantics

Classical relational databases

HoGent

Introduction

- Web applications generate enormous amounts of data every day
- These data are handled by relational database management systems
- 1970: E.F. Codd: “A relational model of data for large shared data banks”
- The relational model is well-suited to client server programming
- Today it is the predominant technology for storing structured data in web and business applications

Introduction

- Last few years: rise of NoSQL databases
- These NoSQL databases are challenging the dominance of relational databaes.
- Relational databases have dominated the software industry for long time
- They provide mechanisms
  - To store data persistently
  - Concurrency control
  - Transactions
  - Reporting

Classical relational database follow the ACID Rules

- A database transaction must be
  - Atomic: A transaction is a logical unit of work which must be either completed with all of its data modifications or nothing at all
  - 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. Otherwise 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

Distributed systems

- A distributed system consists of multiple computers and software components that communicate through a computer network
- A distributed system can consist of any number of possible configurations, such as mainframes, workstations, …
- The computers interact with each other and share the resources of the system

Advantages of Distributed Computing

- Reliability (fault tolerance):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. Other resources can also be shared, e.g. printers
- Flexibility: The system is very flexible, it is very easy to install, implement and debug new services.

Advantages of Distributed Computing

- 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
- Performance: The collection of processors in the system can provide higher performance (and better price/performance ratio) than a centralized computer.

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: Increased risk of security and sharing of data generates the problem of data security

Scalability

- Scalability is the ability of a system to expand to meet the business needs
- E.g. 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.

Vertical scaling

- Vertical scaling = Scale up = To add resources within the same logical unit to increase capacity
- E.g.
  - To add CPUs to an existing server
  - To increase memory in the system
  - To expand storage by adding hard drive

Horizontal scaling

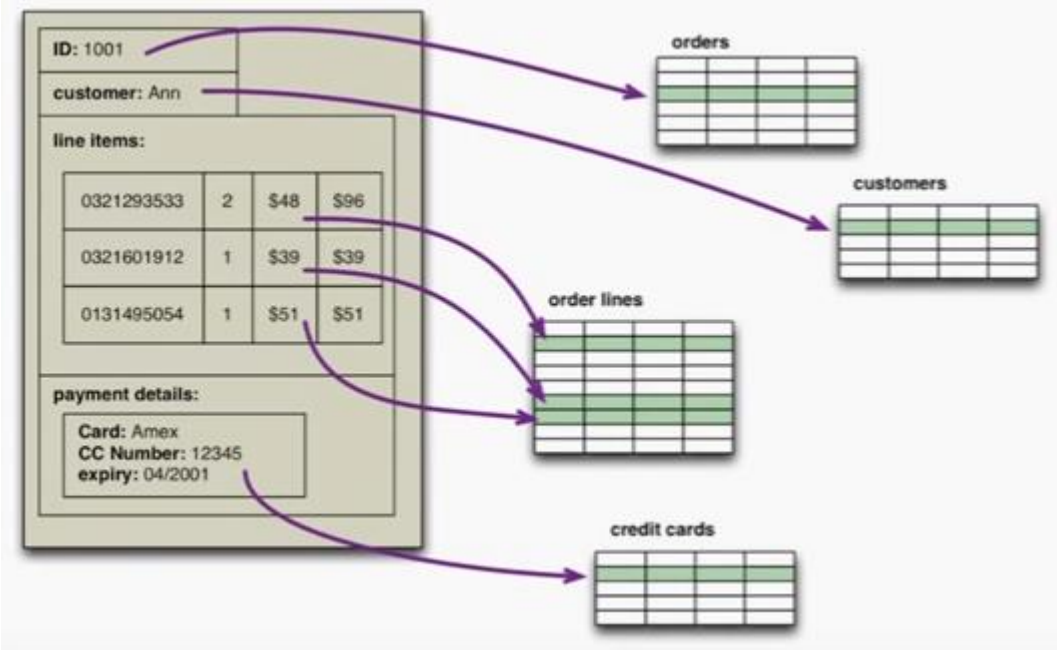
- Horizontal scaling = Scale out = To add more nodes to a system (= adding a new computer to a distributed software application)
- In NoSQL system, adding more nodes to system =>
  - Data store can be much faster
  - The load over those nodes is distributed

Why NoSQL databases?

HoGent

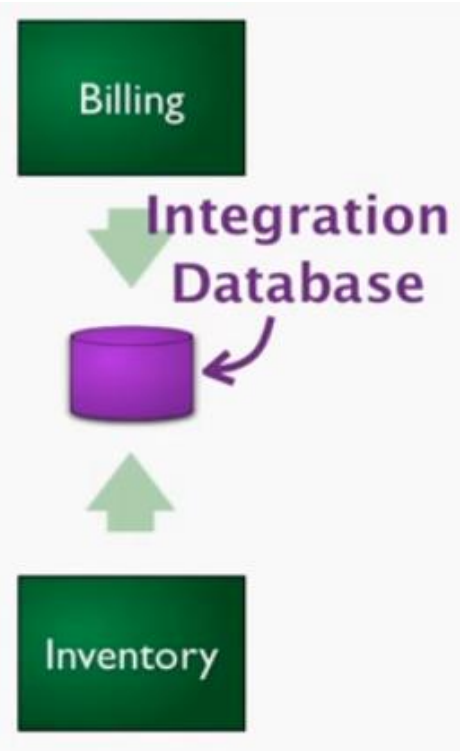
Impedance mismatch

- Impedance mismatch
  - In software: cohesive structures of objects in memory
  - In databases: you have to stripe the object over multiple tables
- NoSQL databases allows developers to develop without having to convert in-memory structures to relational structures



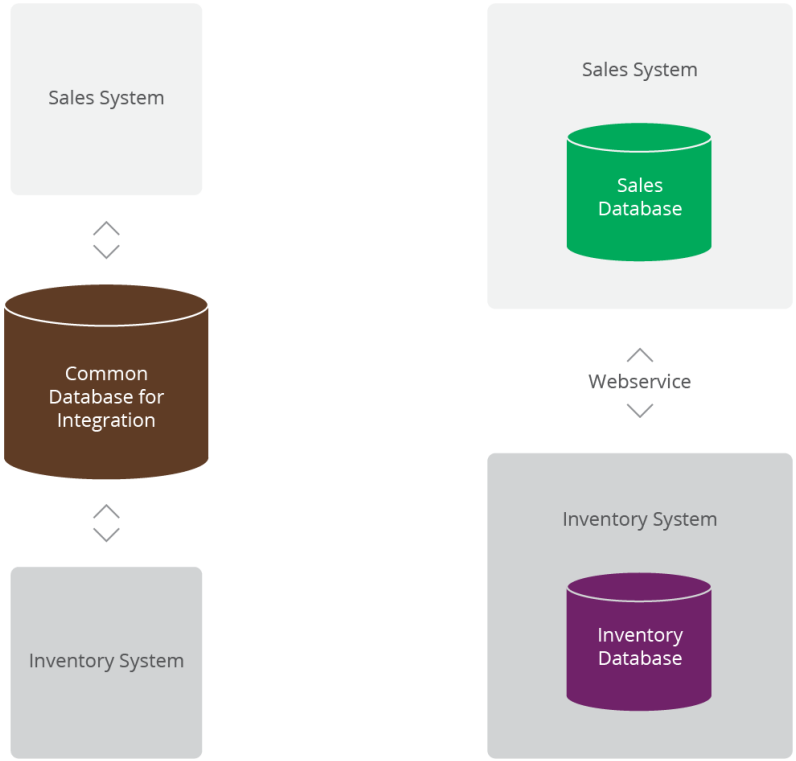
Impedance mismatch

- 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 not?
- SQL databases had become an integration mechanism through which people integrated different applications



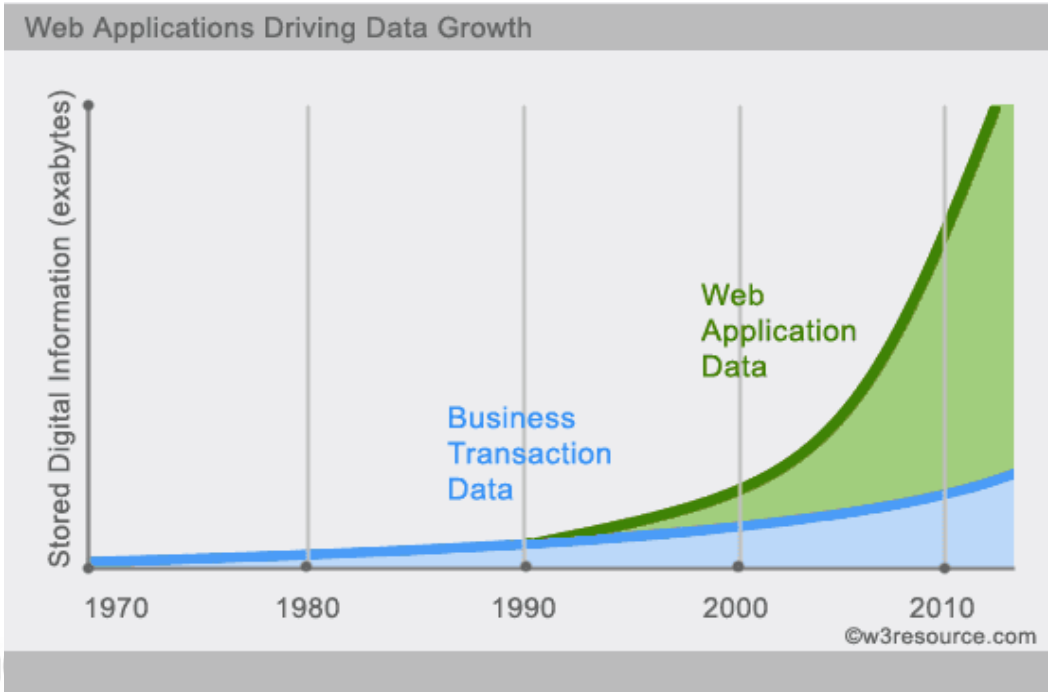
Impedance mismatch

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



What dit change in favor of NoSQL?

- The rise of the web as platform => large volumes of data running on clusters
- Relational databases were not designed
  - To run efficiently on clusters
  - To handle enormous amounts of data



What dit change in favor of NoSQL?

- The data storage needs of an ERP application are different from the data storage needs of a Facebook, …
- Personal user information, social graphs, geolocation data, user-generated content, machine logging data, …
  => data has been increasing exponentially
  => databases are required to process huge amount of data

What dit change in favor of NoSQL?

- As you get large amounts, you need to scale things.
  - (1) You can scale things up using bigger boxes
    - It costs a lot and there are real limits as to how far you can go

  - (2) You can use lots and lots of little boxes, just commodity hardware, all thrown into massive grids
    - Relational databases were not designed to run efficiently on clusters. It's very hard to spread relational databases and run them on clusters

Common characteristics of NoSQL databases

HoGent

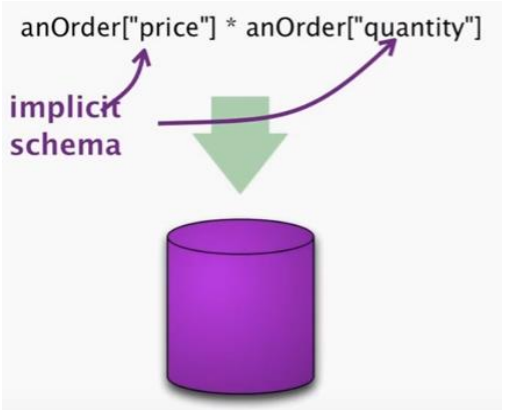
HoGent

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
- 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

Schema-less ramifications

- Almost all NoSQL databases don't tend to have a set schema
- In relational databases you can only put the data into the database as long as it fits in the schema
- No schema => it's easier to migrate data over time
- Implicit schema
  - When you're talking to a database, you're going to say: "I would like the price or I would like the quantity of a specific order" => you are assuming the field is called price (and not cost or whatever else)
  - There is no strict schema but an implicit schema

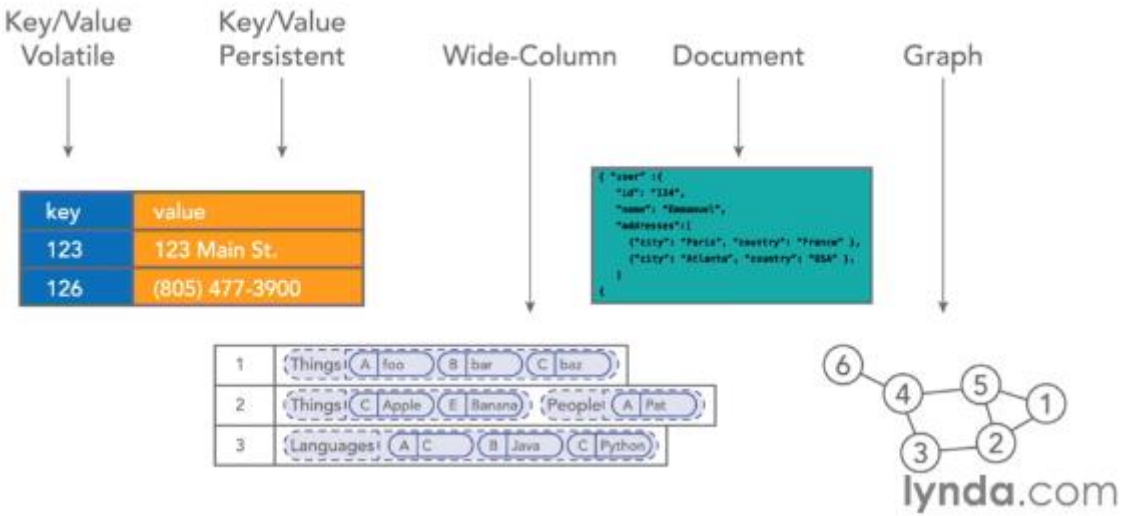


Types of NoSQL databases

HoGent

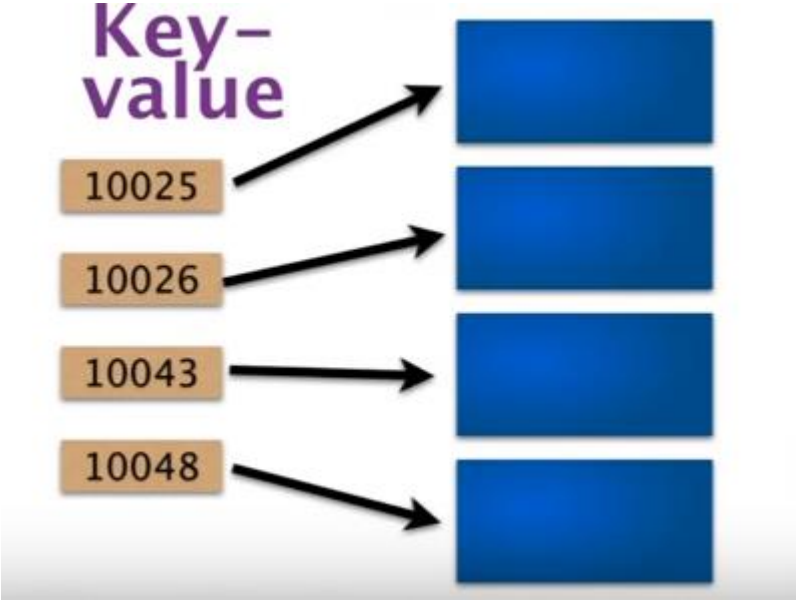
Overview

- NoSQL databases can broadly be categorized in four types.
  - Key – Value databases
  - Document databases
  - Column family databases
  - Graph databases

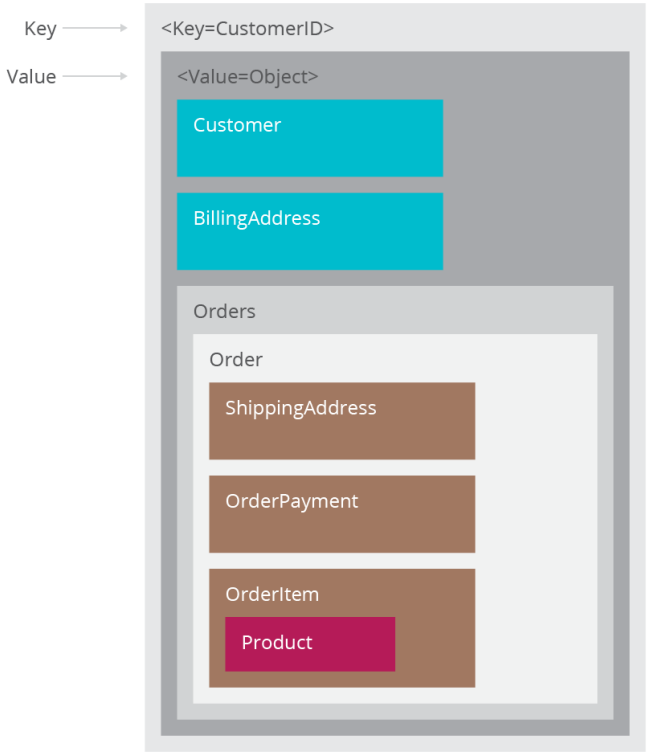


Type 1: Key-Value databases

- Key-Value databases are the simplest NoSQL data stores
- It's like a hashmap but most of the time persistent on the disk
- The database knows nothing about the value: it could be a single number, a complex document, an image, …
  The value is a blob that the data store just stores, whithout caring or knowing what's inside
- They always use primary-key access => great performance



Type 1: Key-Value datab



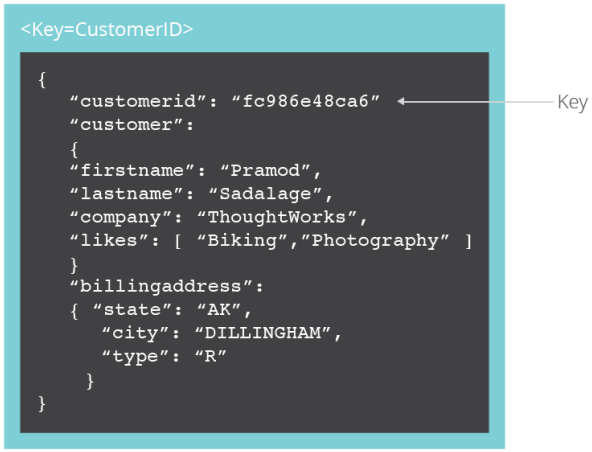
- Some popular key-value stores
  - Riak
  - Redis
  - Memcached
  - DynamoDB
  - …
- Some are persistent on disk (e.g. Riak)
- Some are not persistent on disk (e.g. Memcached) => if a node goes down all the data is lost and needs to be refreshed from the source system
- http://try.redis.io/

Type 2: Document databases

- A document database thinks of a database as a storage of a whole mass of different documents
- Each document is some complex data structure, like JSON, XML, BSON, …
- The documents stored are similar to each other but do have to be exactly the same
- Document stores don't tend to have a set schema
- 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

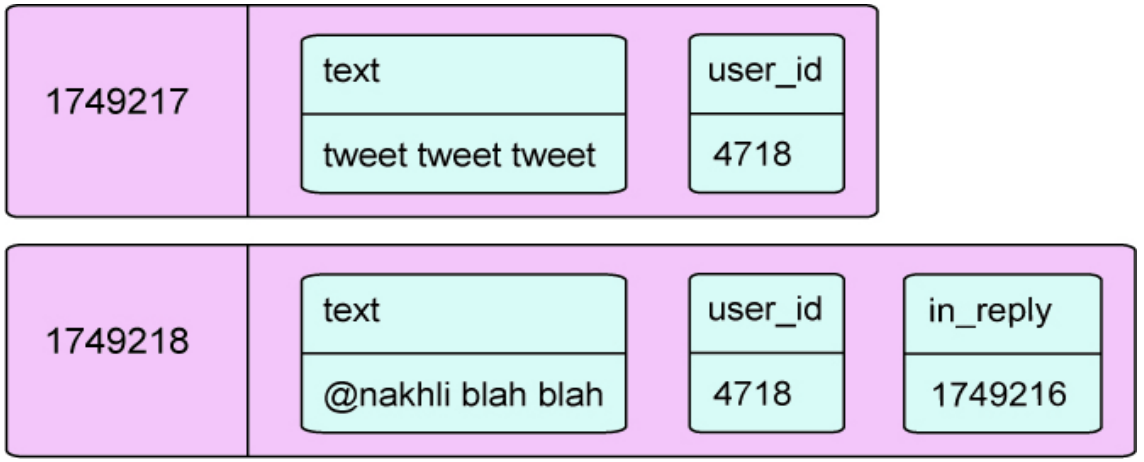
Type 2: Document databases

- Document databases have been adopted more widely than any other type of NoSQL database
- Some popular document databases
  - MongoDB
  - CouchDB
  - …



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. E.g. for a Customer, we would often access their Profile information at the same time, but not their Orders
- Some popular column – family databases:
  - Cassandra
  - Hbase
  - ...

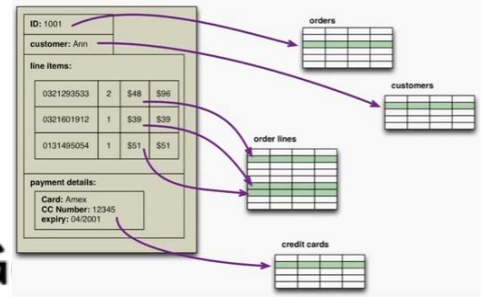


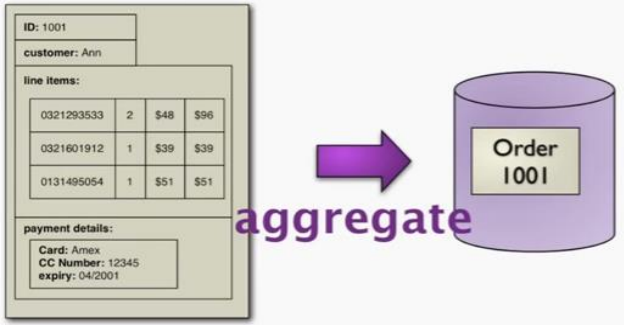
Aggregate – Oriented database

- Key – value databases
- Document databases
- Column – family databases

Aggregate – Oriented databases

- Aggregate – Oriented databases: Allow you to store big complex structures
- When we have to model things, we have to group things together into natural aggregates
- When we want to store this in a relational database, we have to splatter this unit over a whole bunch of tables.

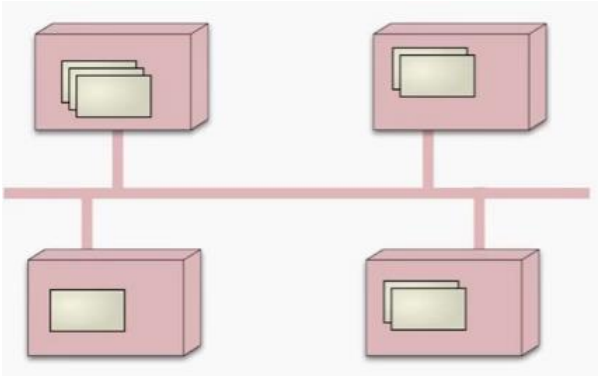




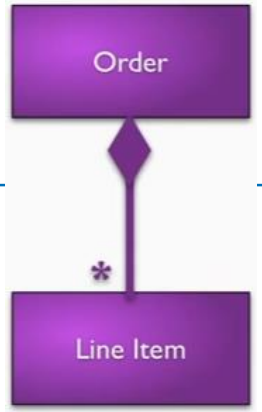
HoG

Aggregate – Oriented database

- This becomes really useful 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 the aggregate tells you what data is going to be accessed together.
- So when you need the data, you go to one node on the cluster instead of picking up different rows from different tables.

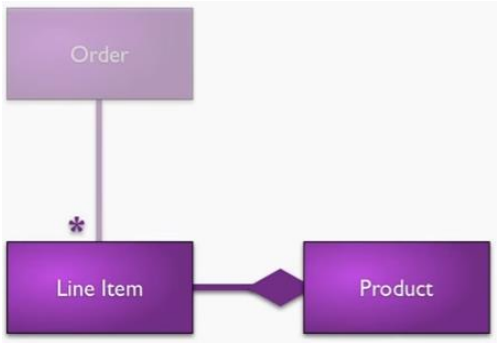


Aggregate – Oriented database



- E.g.: orders and line items. We think of the order as 1 thing, a single unit.
- But if you want to know the revenue of a certain product, you don't care about the orders at all.
  You only care about what's going on with individual line items of many orders grouping together by product. You want to change the aggregation structure from a structure where orders aggregate line items to a structure where products aggegrate line items.
- In a relational database, this is straightforward. It's more complicated if you use NoSQL.

Product	revenue
321293533	3083
321601912	5032
131495054	2198
...	...
...	...
...	...
...	...
...	...
...	...

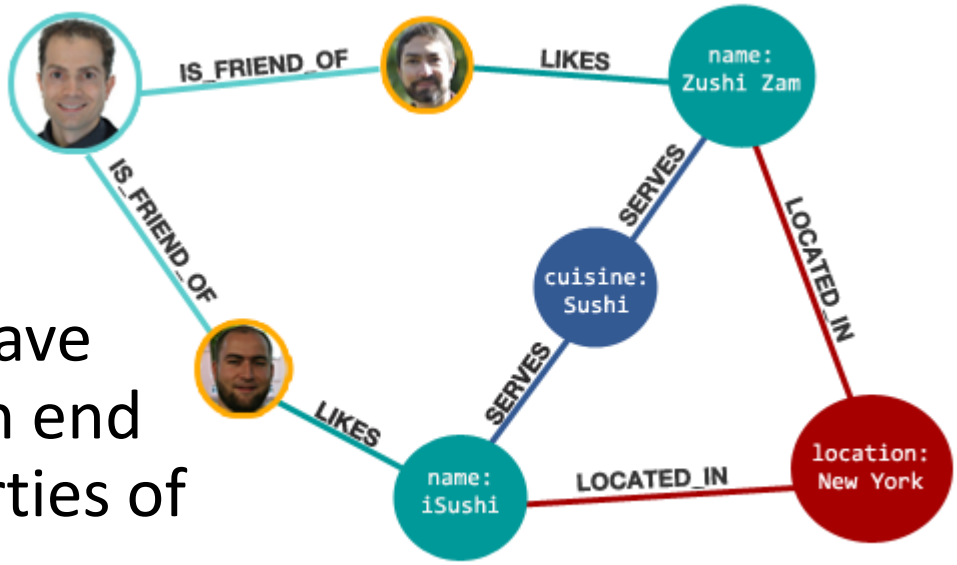


Distribution models

- Aggregate oriented databases make the 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.
- 2 styles of distributing data
  - Sharding: distributes different data across multiple servers, so each server acts as a single source for a subset of data
  - Replication: copies data across multiple servers, so each bit of data can be found in multiple places
    - Master – slave replication
    - Peer – to – peer replication

Type 4: Graph databases

- Graph databases are not aggregate oriented at all.
- A node and arc graph structure
- Not only the data in the nodes is important, but also the relationships between the nodes, You can jump around the relationships.
- Nodes can have different types of relationships between them
- Relationships don't only have a type, a start node and an end node, but can have properties of their own.



- Graph databases are used for data that has a lot of many – to – many relationships

Type 4: Graph databases

- Some graph databases
  - Neo4J
  - Infinite Graph
  - …
- http://console.neo4j.org/

Overview


Document
mongoDB
CouchDB
RAVEN DB
Colum
n-
family
Cassandra
APACHE
HBASE
Neo4j
NOSQL for the Enterprise
Project Voldemort
riak
Graph
redis
Key-value

NoSQL and consistency

HoGent

Introduction

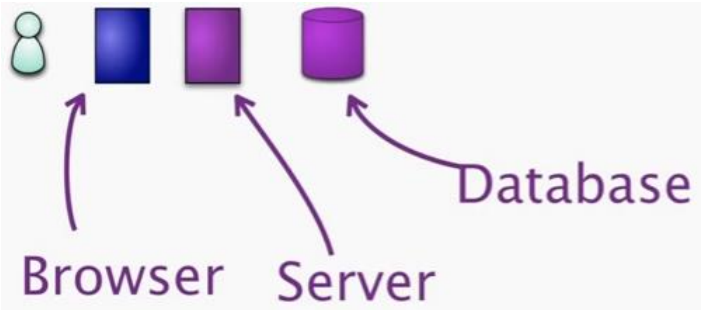
- Relational databases are ACID
- If you've got a single unit of information and you want to split it across several tables, you don't want to write half of the data and someone else writes a different half of the data
- You want an atomic update so that you either succeed or fail and nobody comes in the middle and messes things up.
- Graph databases tend to follow ACID updates
  - They compose the data even more than relational databases

Introduction

- Aggregate oriented databases don't need transactions as much, because the aggregate is a bigger structure.
- Any aggregates update is going to be atomic, isolated and 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. This problem occurs much more rarely than you would expect.

Example

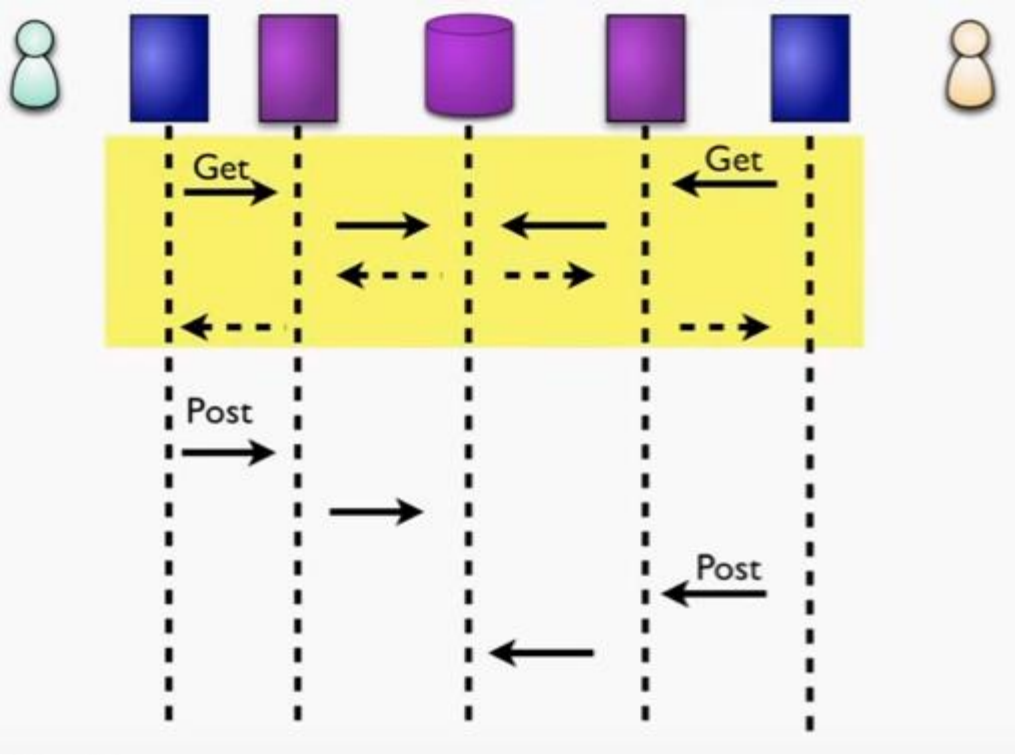
- A person is talking to a browser.
- The browser talks to the server
- The server talks to a single database



- There are 2 people talking to the same database at the same time, although through different browsers and servers.

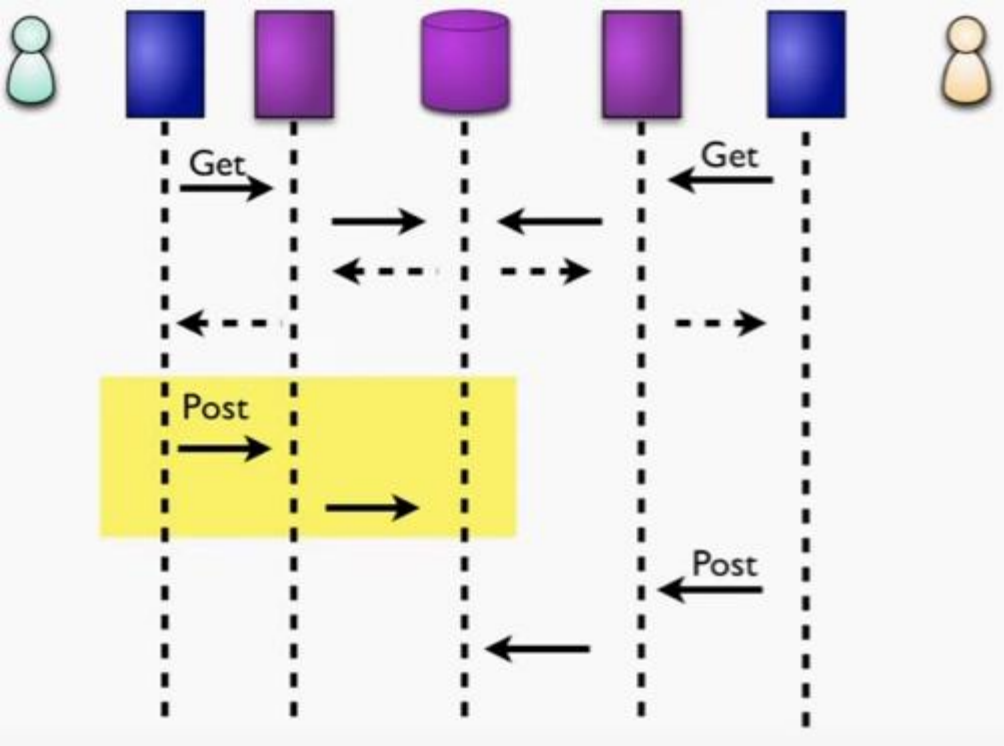
Example

- 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".



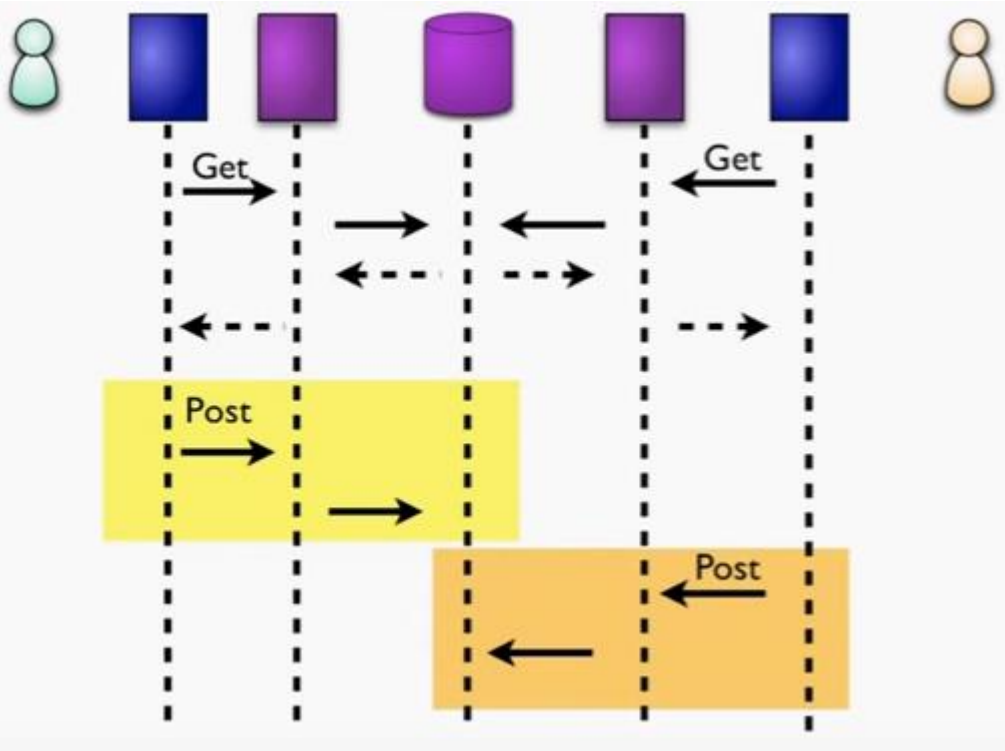
Example

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



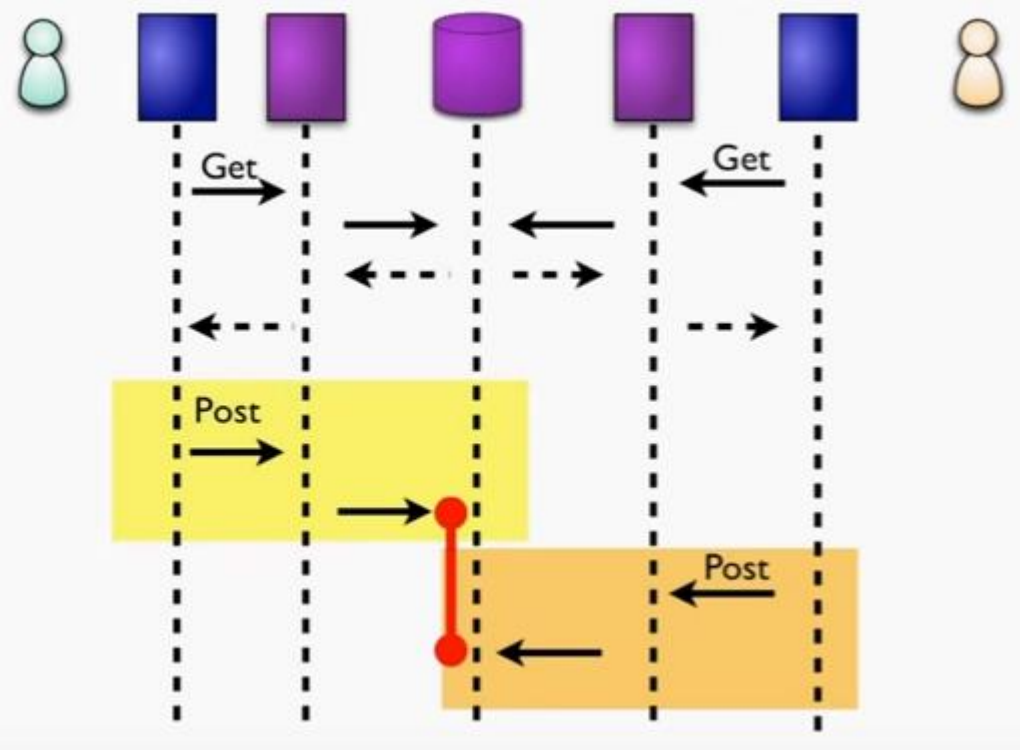
Example

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



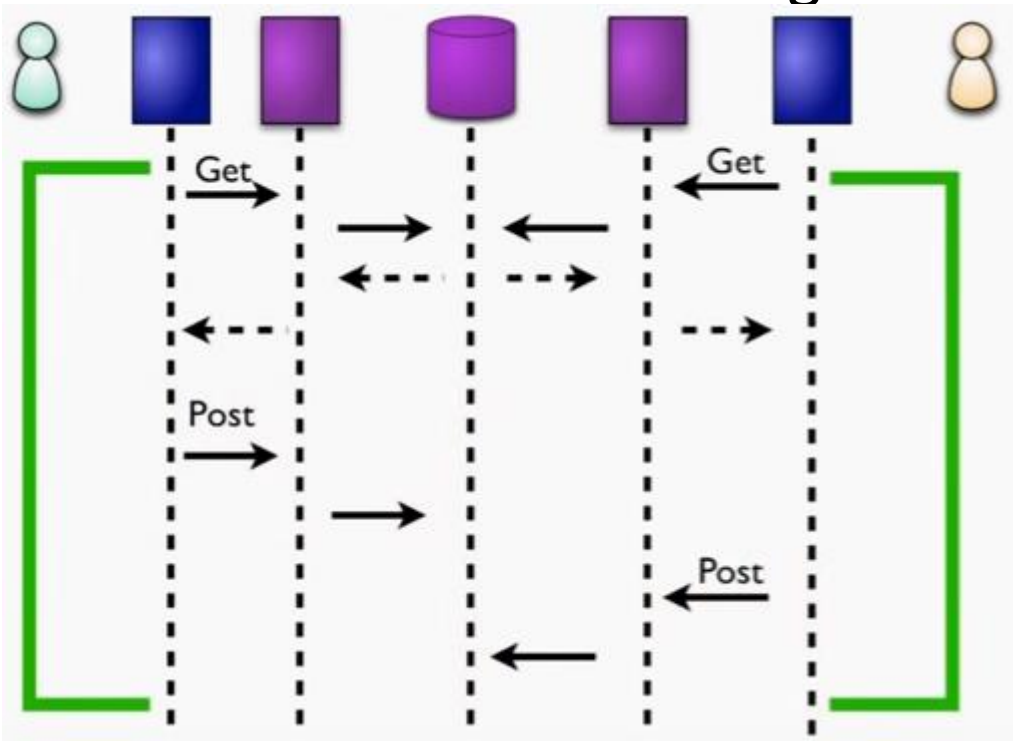
Example

- 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.



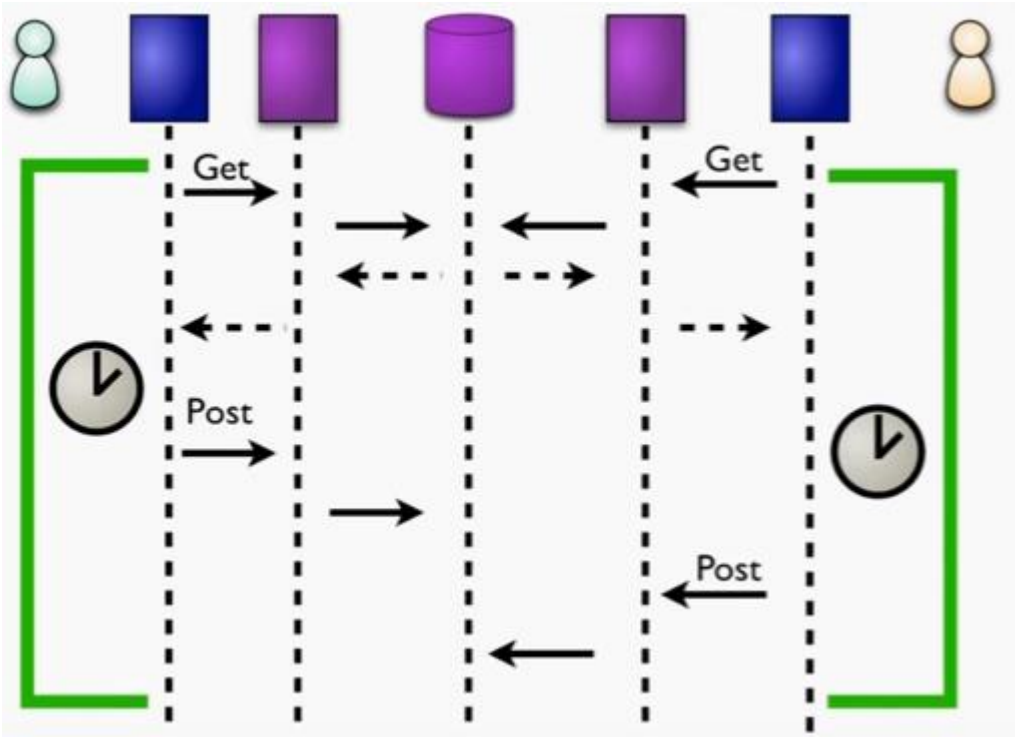
Example

- ACID to the rescue.
- To prevent this conflict, you can wrap the entire interaction from getting the data on the screen and posting it back again in a transaction. That way, one of both will be told: "You have to do this again."



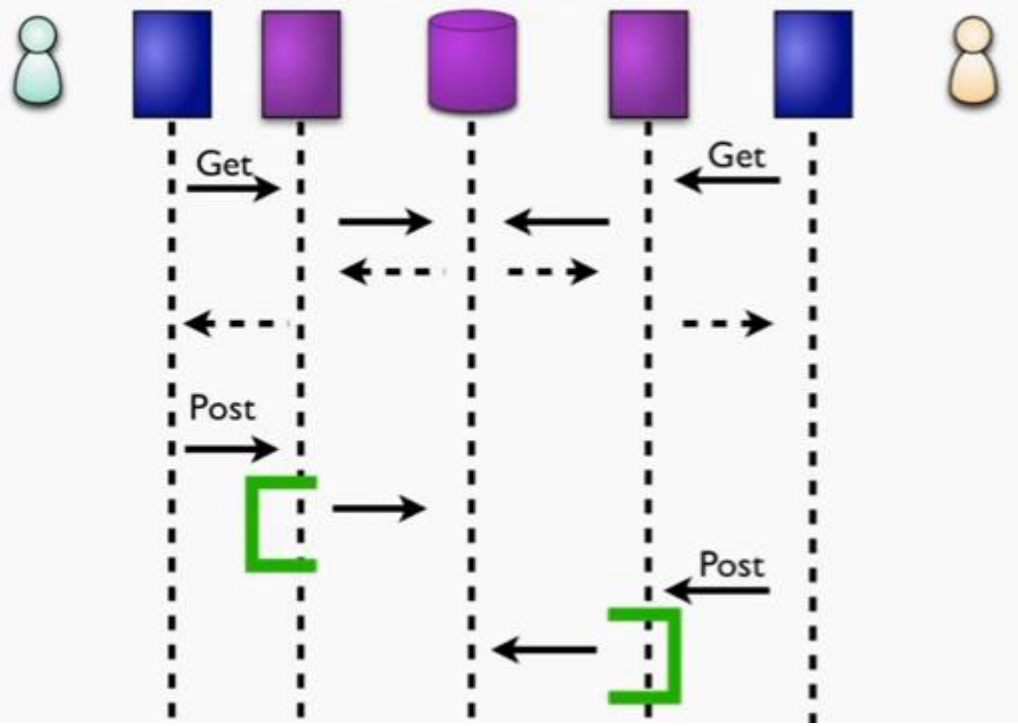
Example

- Holding a transaction open for that length of time, while you've got a user looking and updating the data through the UI, that's going to really suck your performance out of your system.



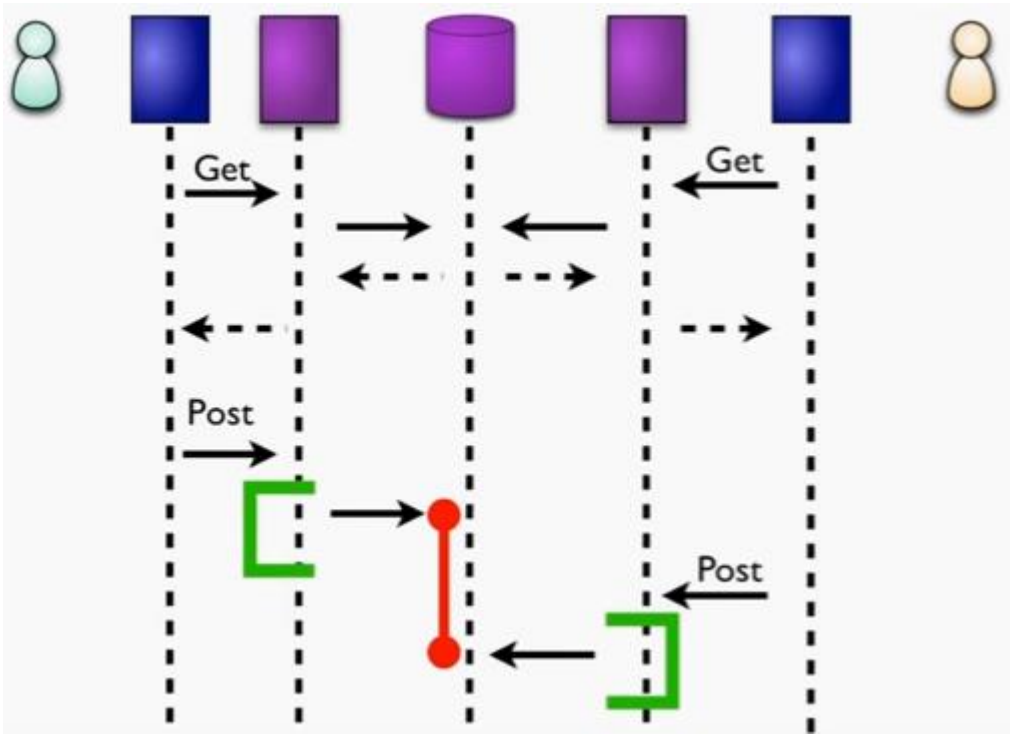
Example

- Just wrap the transaction around the update



Example

- But you still effectively get a conflict because the two people made updates on the same piece of information.

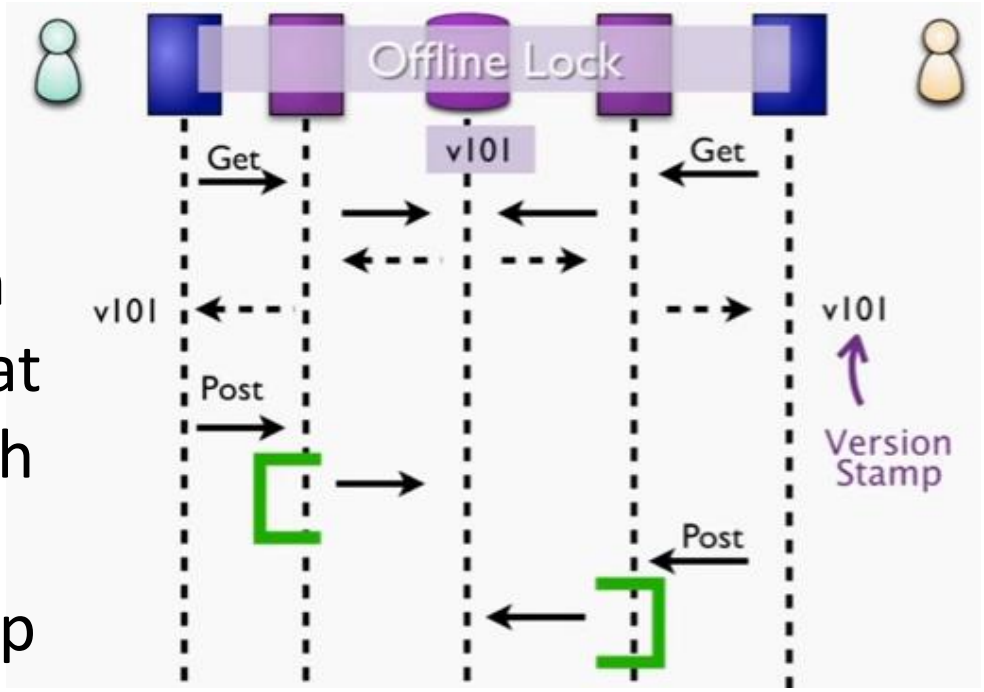


Example

- 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.

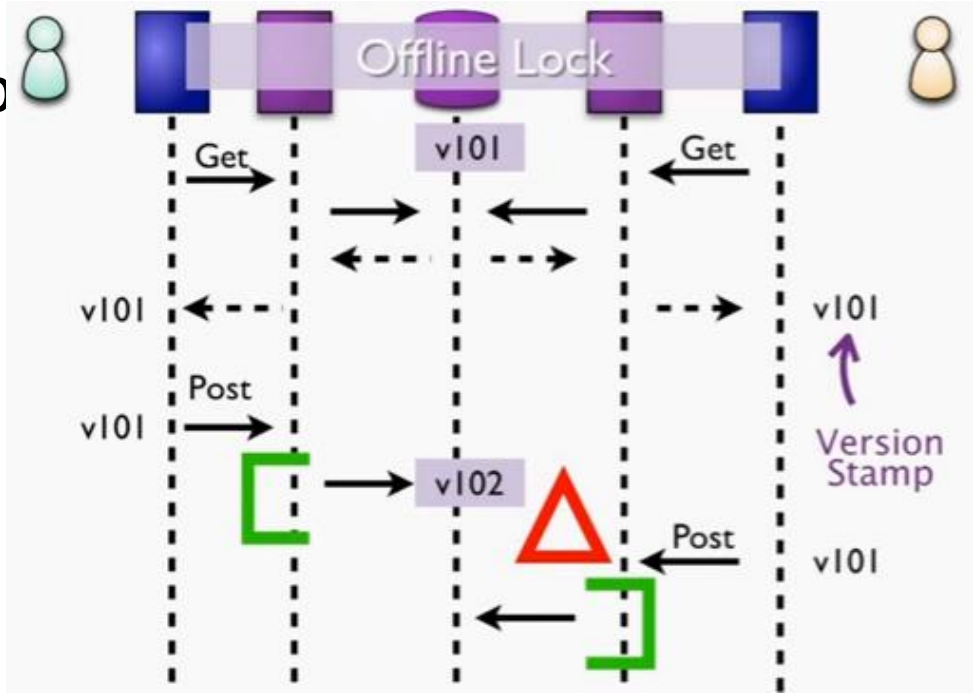
Example

- 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.



Example

- 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.



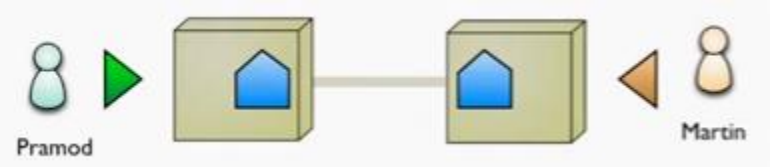
You can use the same basic techniques with NoSQL databases.

Consistency and availability

- Logical consistency: these consistency issues occur whether you're running on a single machine or on a cluster
- 2 styles of distributing data
  - Sharding: distributes different data across multiple servers, so each server acts as a single source for a subset of data
  - Replication: copies data across multiple servers, so each bit of data can be found in multiple places
    - +
      - More nodes handling the same set of requests
      - Resilience: if one of the nodes goes down, the other replicas can still keep going
    - -
      - Consistency problems

Consistency and availability

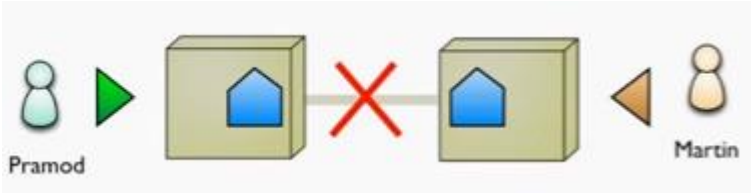
- Pramod (in India) and Martin (in the US) want to book the same hotel room
- They send out a booking request
- They send their requests to local processing nodes



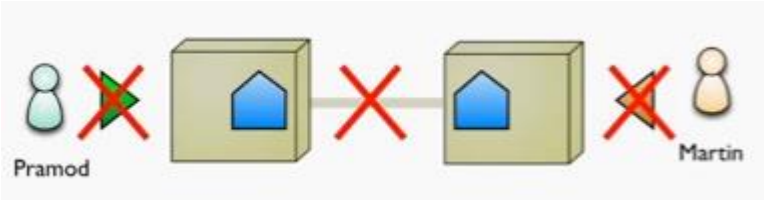
- The processing nodes need to communicate ensuring only one of both can book the hotel room

Consistency and availability

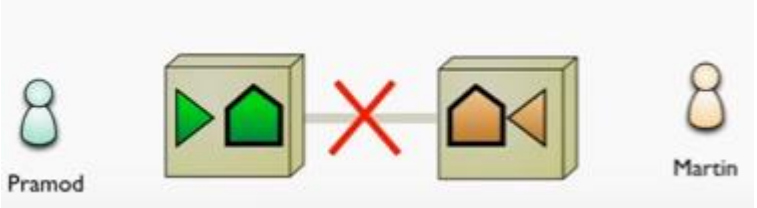
- The communication line has gone down. The two nodes cannot communicate.



- They send out their requests
- Alternative 1
  The system tells the users the communication lines have gone down, so they can't take the hotel bookings at the moment

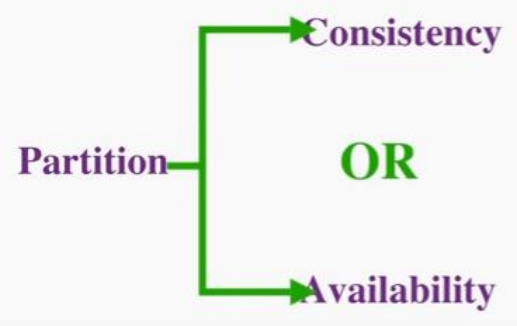


- Alternative 2
  The system accepts both bookings and the hotel room is double booked



Consistency and availability

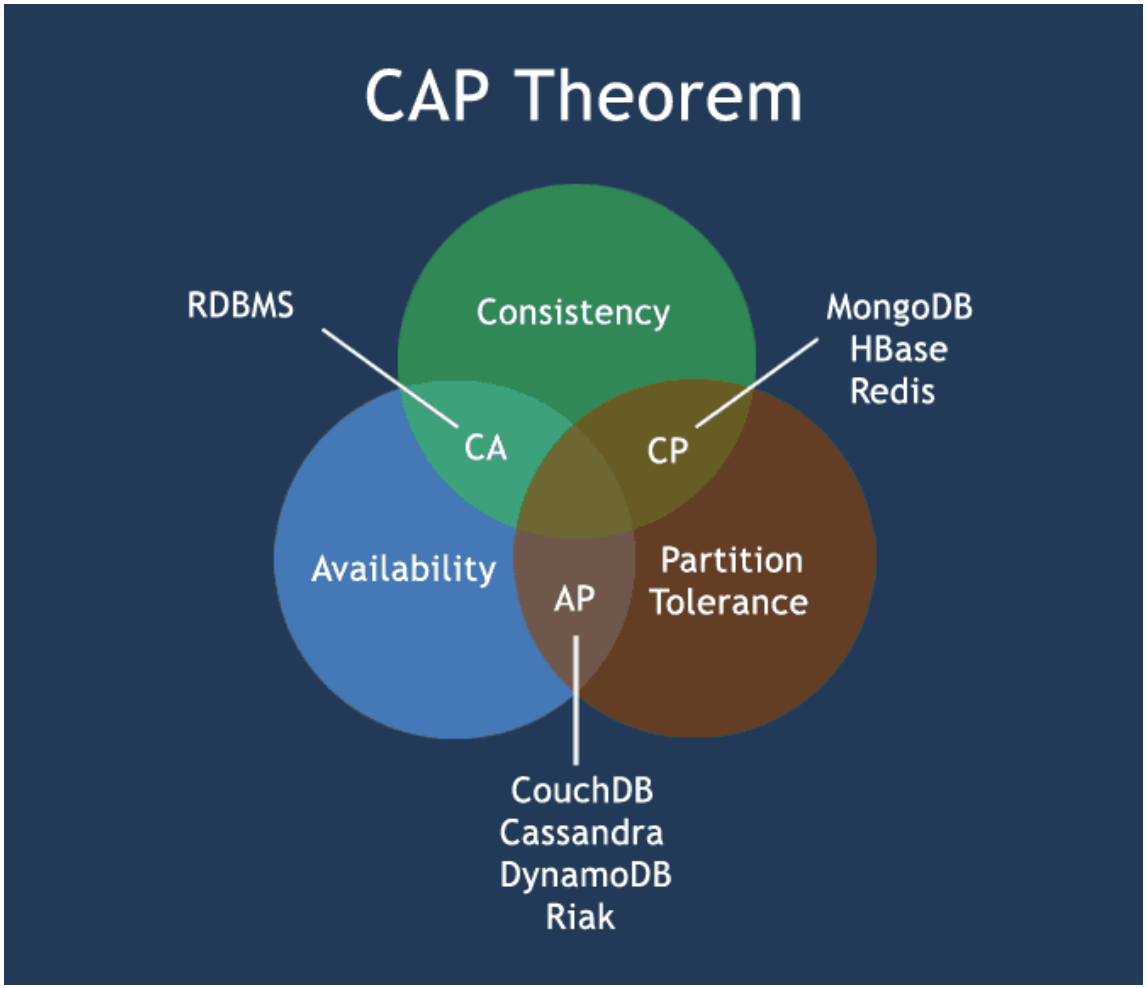
- This illustrates a choice between consistency (“I’m not going to do anything while the communication lines are down”) and availability (“I’m going to keep on going but at the ris of introducing an inconsistent behavior”)
- This is a business choice
- This example is a “simplification” of the CAP theorem



CAP theorem

- Consistency: The data in the database remains consistent after the execution of an operation.
- Availability: The system is always on (service guarantee availability), no downtime.
- Partition Tolerance: 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.
- As soon as you have a distributed system, you have to only can pick 2 out of 3
- This isn't a single binary choice. You can trade off levels of consistency and availability

CAP theorem


CAP Theorem
RDBMS
Consistency
MongoDB
HBase
Redis
CA
CP
Availability
Partition
Tolerance
AP
CouchDB
Cassandra
DynamoDB
Riak

CAP theorem

- Many NoSQL databases try to provide options where the developer has choices by which they can tune the database as per their needs.
- E.g. Riak
  - 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.
- Consider a Riak cluster with 5 nodes
  - How to make the system highly available?
  - How to make the system very consistent?

Summary

HoGent

Polyglot persistence

- The rise of NoSQL databases does not mean that RDBMS databases will disappear
- 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.

Why you might want to use NoSQL?

- Large scale data
  You've got more data than you can comfortably or economically fit into a single SQL database server. Running relational databases across clusters isn't easy. Big amounts of data is coming at us (not just a few companies, e.g. Google, …)
- Easier development
  Developers want to develop more easily, e.g. someone who works for a newspaper deals with articles and want to store and retrieve articles as a single unit = impedance mismatch

Exercises

HoGent

What type of database would you use?

- (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

What type of database would you use?

- (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

Is my hardware under-utilized?

- 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
- We really only need to have simple aggregate queries going against this log data: we need min, max, count in order to know how the hardware is being utilized because we can figure out by aggregates against those logs when hardware is not utilized. We don't need to do any predictive queries or anything around machine learning.

Storing customer or donor information

- Companies want to understand what are the characteristics 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 gonna 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

Linking museum data

- 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.