Introduction to NoSQL

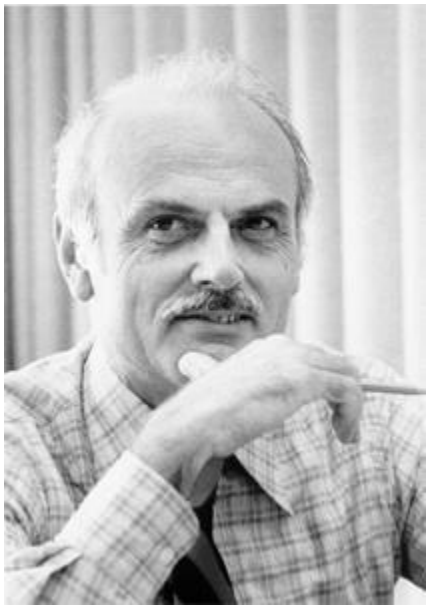
HO
GENT

Understanding NoSQL

HO
GENT

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



HO
GENT

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. A transaction, if executed in isolation, renders the database from one consistent state into another consistent state
  - Isolated: In situations where multiple transactions are executed concurrently, the outcome should be the same as if every transaction were executed in isolation
  - Durable: Durability refers to the fact that the effects of a committed transaction should always be persisted into the database

HO GENT

Introduction

- Data is becoming easier to access and capture through third parties such as Facebook, Google and others.
- User-generated content and machine logging data are just a few examples where the data has been increasing exponentially.
- SQL databases were never designed to process these huge amounts of data.
- NoSQL data stores are becoming widely adopted in situations that involve large volumes of data as well as large rates of data growth.

HO
GENT

NoSQL what does it mean

- NoSQL = Not Only SQL = There is more than one storage mechanism that could be used 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

HO
GENT

Limitations of 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

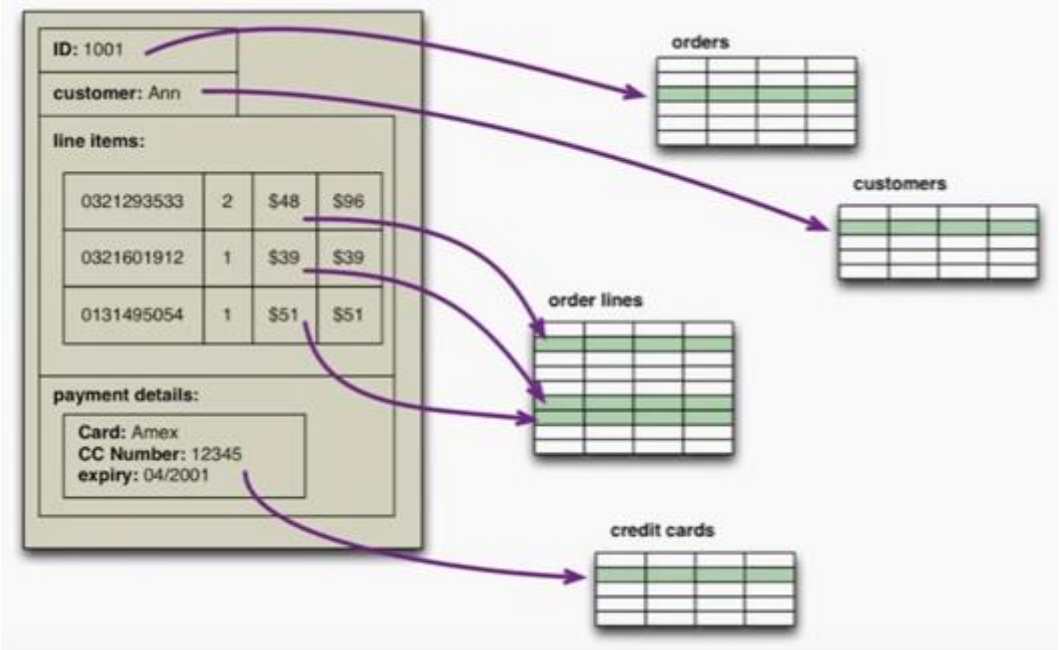
HO
GENT

Why NoSQL databases?

HO
GENT

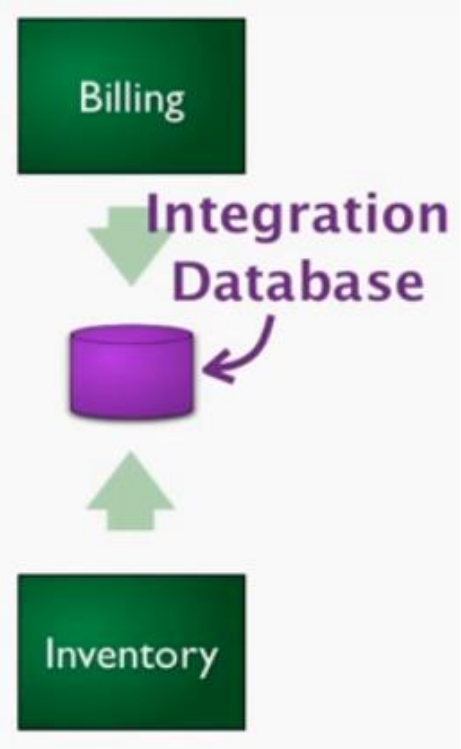
Impedance mismatch

- 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



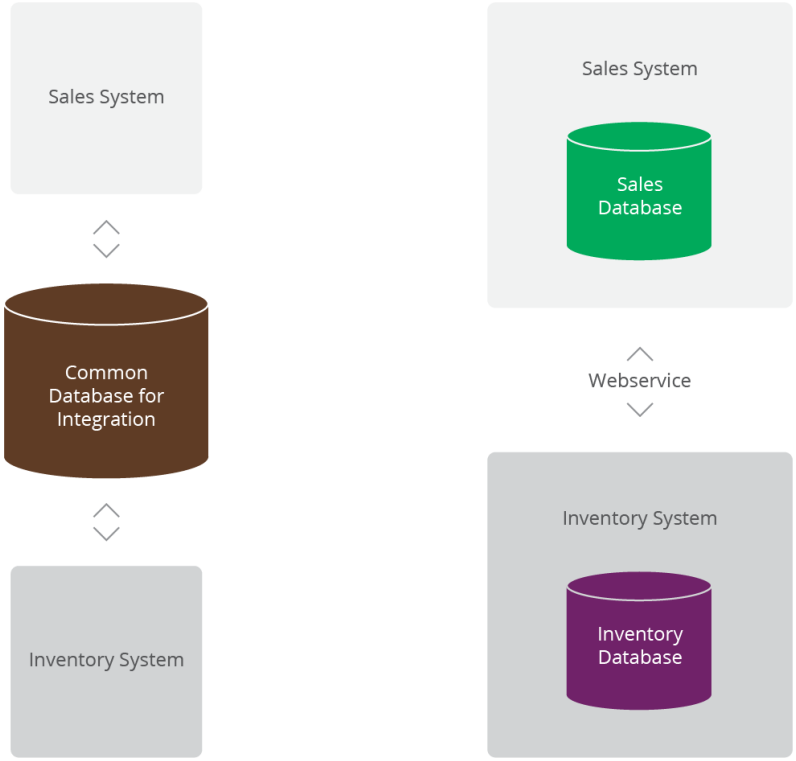
Impedance mismatch

- 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

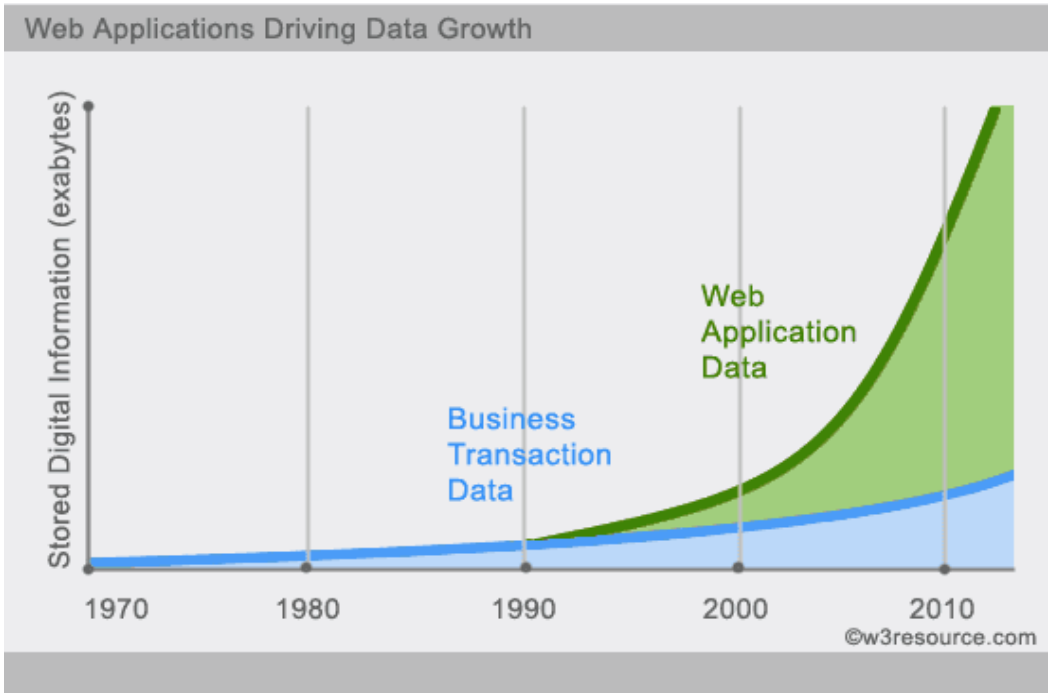
- Nowadays there is a movement away from using databases as integration points in favor of encapsulating databases using services.



HO GENT

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



HO
GENT

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

HO
GENT

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 torun efficiently on clusters. It's very hard to spread relational databases and run them on clusters

HO
GENT

Common characteristics of NoSQL databases

HO
GENT

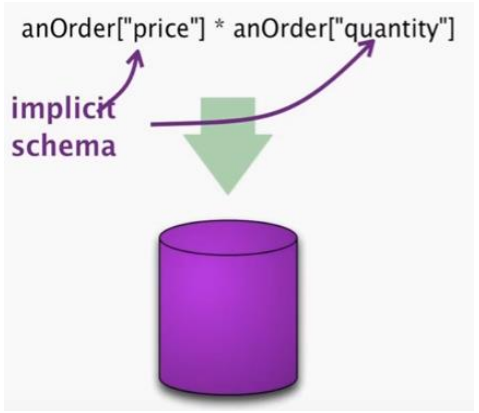
The most common characteristics of NoSQL databases

- non – 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

HO
GENT

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 ...)
  - There is no strict schema but 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."



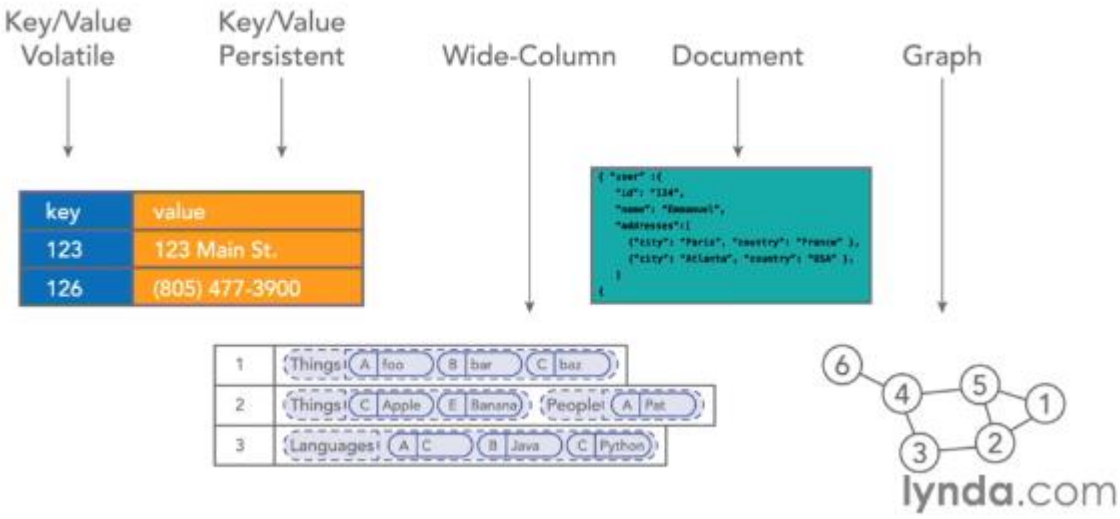
HO GENT

Types of NoSQL databases

HO
GENT

Overview

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



HO GENT

Types of NoSQL databases

Key-Value stores

Document stores

Column-oriented databases

Graph based databases

HO
GENT

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

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

HO
GENT

Type 1: Key-Value databases

- 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/

HO
GENT

Type 1: Key-Value databases

- Typically the API supports the following functions:
  - get(key). This function returns the value that was stored in the record with the key that is passed as a parameter.
  - put(key, value). This function adds a new record to the dictionary with key (key) and value (value).
  - delete(key). This function removes the (key, value) – record with the given key
- No function to change the data
- No ad-hoc searching or data-analysis

HO
GENT

Type 1: Key-Value databases

- Example
  - Assume there is an art museum that has a mobile application by which registered visitors can give their opninion about paintings during their tour in the museum. The app registers every minute automatically where the visitor is located and gives information about the paintings on this location. The visitor subsequently can give his opinion.
  - The app generates a lot of data that must be written to the database quickly, so the developer has chosen to use a key-value database ‘Visitors opinions’.

HO
GENT

Type 1: Key-Value databases

- Example

VisitorID, timestamp	Value
V1, 15/1:14h00	‘Room 1’
V1, 15/1:14h01	‘Room 1, not nice, crowded’
V1, 15/1:14h01	‘Room 1, 2/10’
V2, 15/1:14h02	‘Room 1, Rembrandt is amazing’
V1, 15/1:14h03	‘Room 1, not having a good time’
V2, 15/1:14h03	‘Room 1, 9/10’
V1, 15/1:14h04	‘Room 2, this is more like it, 7/10’
V2, 15/1:14h04	‘Room 1, really nice in here’
V3, 15/1:14h04	‘Room 1, crowded’

HO
GENT

Types of NoSQL databases

Key-Value stores

Document stores

Column-oriented databases

Graph based databases

HO
GENT

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, …
- These documents consist of components that each have a name and a value.
- The name of the component should be unique within the document.
- Each document has a key-component, of which the value is unique through the entire database.
- 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

Type 2: Document databases

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

HO GENT

Type 2: Document databases

- Most modern NoSQL databases choose to represent documents using JSON

```
{
    "title": "Harry Potter",
    "authors": ["J.K.  Rowling", "R.J.  Kowling"],
    "price": 32.00,
    "genres": ["fantasy"],
    "dimensions": {
            "width": 8.5,
            "height": 11.0,
            "depth": 0.5
    },
    "pages": 234,
    "in_publication": true,
    "subtitle": null
}
```

HO GENT

Type 2: Document databases

- Example

The address document is embedded in the visitors document.

The visitor document is referred to in the opinion documents

```
{
  _id: "V1",
  name: "Eva",
  address: {
        street: "Funstraat 3",
        city: "Aalst",
        zipcode: 9300,
        country: "Belgium"
        },
  language: "dutch"
}

{
  _id: "V2",
  name: "Adam",
  language: "dutch"
}
```

```
{
  _id: 234567891,
  day: "15/1/2017",
  hour: "14:00",
  visitor_id: "V1",
  place: "Room 1"
}

{
  _id: 234567892,
  day: "15/1/2017",
  hour: "14:01",
  visitor_id: "V1",
  place: "Room 1",
  comments: ["not nice", "crowded"],
  score: 2
}

{
  _id: 234567893,
  day: "15/1/2017",
  hour: "14:02",
  visitor_id: "V2",
  place: "Room 1",
  comments: ["Rembrandt is amazing"]
}
```

Type 2: Document databases

- Example
  - db.opinions.insert({_id: 234567894, day: "15/1/2017", hour: "14:02", visitor_id: "B1", place: "Room 1"})
  - db.opinions.find()
  - db.opinions.find({place: "Room 1"})
  - db.opinions.find({score: {$lt: 5}})
  - db.opinions.createIndex({place: 1})
  - db.visitors.update({"_id": "V1"}, {$set: {"address.street": "Funstraat 5"}})
  - db.visitors.remove({"_id": "V1"})

HO
GENT

Types of NoSQL databases

Key-Value stores

Document stores

Column-oriented databases

Graph based databases

HO
GENT

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

Type 3: Column family databases

- In an row-oriented relation each row is an entity.
- The relation has a schema that describes the attributes of all entities.
- In a column-oriented relation each row is just a component (attribute) of an entity.
- The components are grouped in column families according to the following rule: components that are retrieved or adjusted often together, are grouped in the same column family.

Type 3: Column family databases

- Example

Row_id	Column_name	Version	Column_value
V1	name	v1	Eva
V1	street	v1	Funstreet 3
V1	street	v2	Funstreet 5
V1	city	v1	Aalst
V1	zipcode	v1	9300
V1	country	v1	Belgium
V1	language	v1	dutch
V2	name	v1	Adam
V2	language	v1	dutch

Row_id	Column_name	Version	Column_value
O1	day	v1	15/1/2017
O1	hour	v1	14:00
O1	visitor_id	v1	V1
O1	place	v1	Room 1
O2	day	v1	15/1/2017
O2	hour	v1	14:01
O2	visitor_id	v1	V1
O2	place	v1	Room 1
O2	comment	v1	not nice
O2	comment	v1	crowded
O2	score	v1	2
O3	day	v1	15/1/2017
O3	hour	v1	14:02
O3	visitor_id	v1	V2
O3	place	v1	Room 1
O3	comment	v1	Rembrandt is amazing

- Row_id
- Column_name to save the name of the component
- Column_value to save the value of this component
- Version

HO GENT

Type 3: Column family databases

- A few possible instructions
  - get(Column_family, Row_id, Column_name, Version).
  - insert(Column_family, Row_id, Column_name, Version, Column_value)
  - delete(Column_family, Row_id, Column_name, Version, Column_value)
- Easy, on SQL based SELECT-FROM-WHERE instructions
- Most popular column family databases
  - Cassandra
  - Hbase
  - …

HO GENT

Comparison

- Performance → how fast are read- and write-operations?
- Scalability → can the data easily be expanded?
- Flexibility → can the structure of the data easily be adapted?
- Complexity → how complex is the database model?
- Functionality → are there a lot of possibilities to manipulate and query the data?

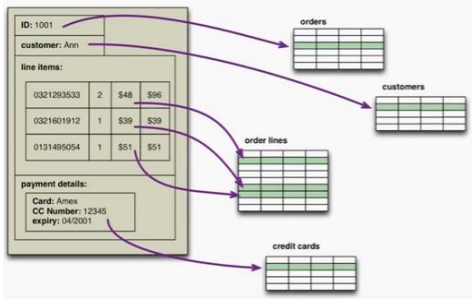
	Key-value databases	Document databases	Column family databases	Relational databases
Performance	very high	high	high	average
Scalability	very high	average / high	high	average
Flexibility	very high	high	average	low
Complexity	none	variable	low	high
Functionality	none	average / high	low	high

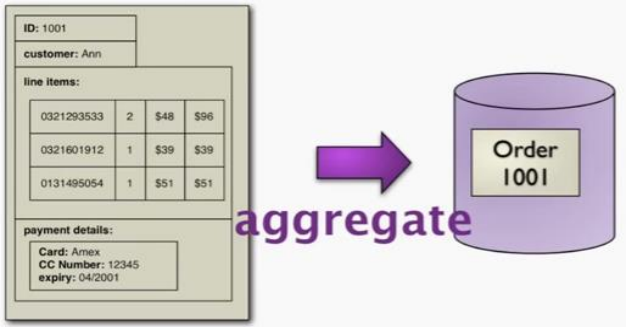
Aggregate – Oriented database

- 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.

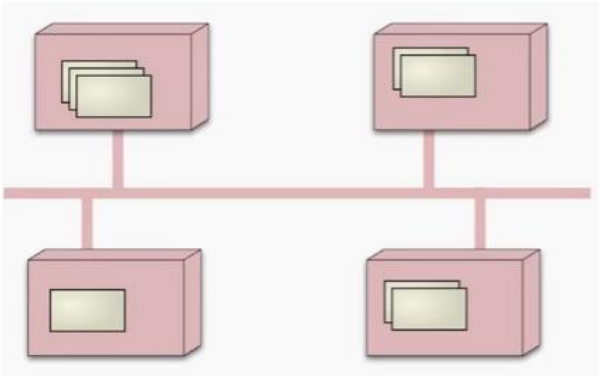




HO GENT

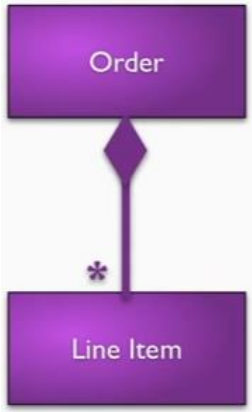
Aggregate – Oriented database

- 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.



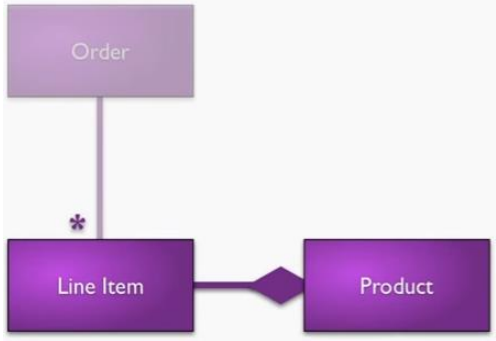
HO
GENT

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

HO
GENT

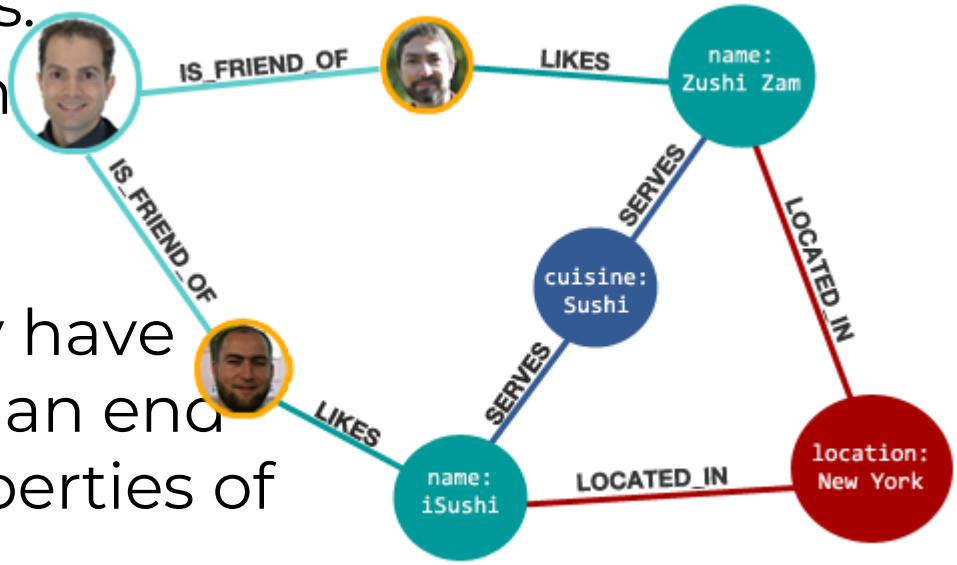
Types of NoSQL databases

Key-Value stores
Document stores
Column-oriented databases
Graph based databases

HO
GENT

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



HO GENT

Type 4: Graph databases

- Some graph databases
  - Neo4J
  - Infinite Graph
  - …
- Demo: http://console.neo4j.org/
- https://neo4j.com/sandbox/?ref=home > Launch the Free Sandbox > Log In > Movies > Create

HO
GENT

NoSQL and consistency

HO
GENT

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
- That's what transactions are all about.
- 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 decompose the data even more than relational databases

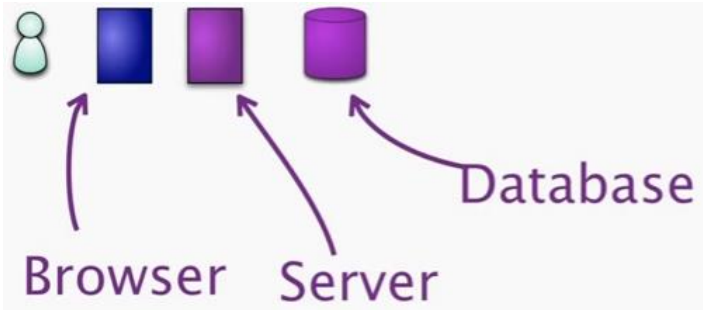
HO
GENT

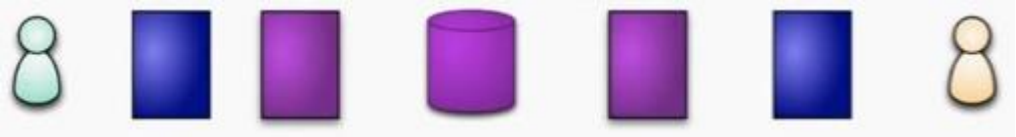
Introduction

- 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.

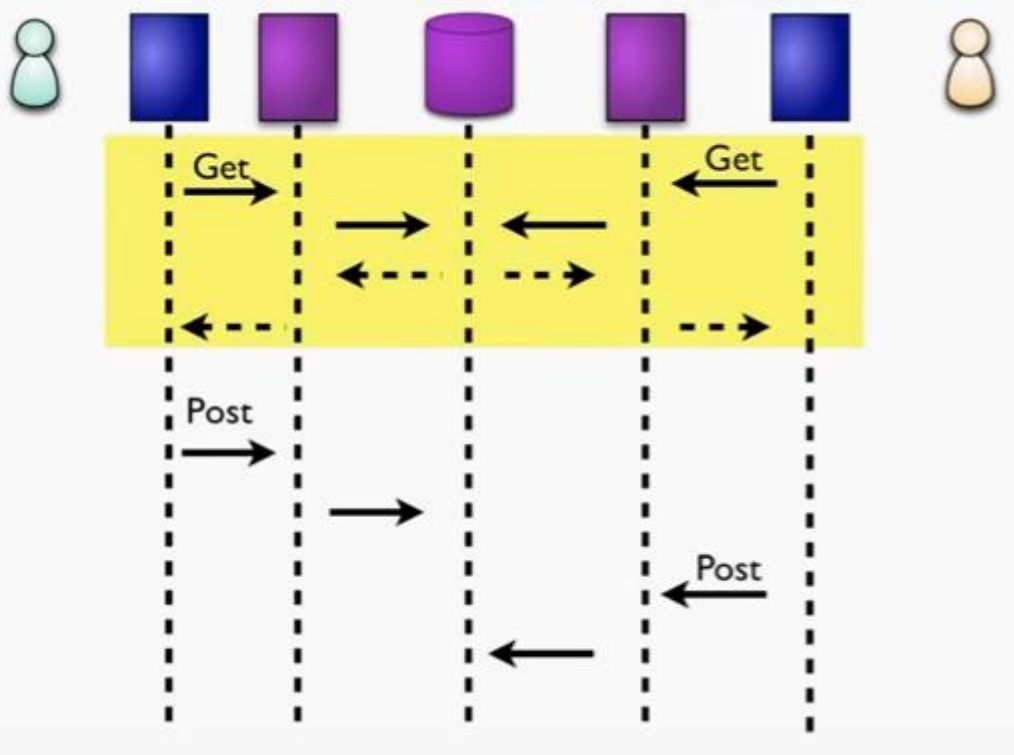




HO
GENT

Example

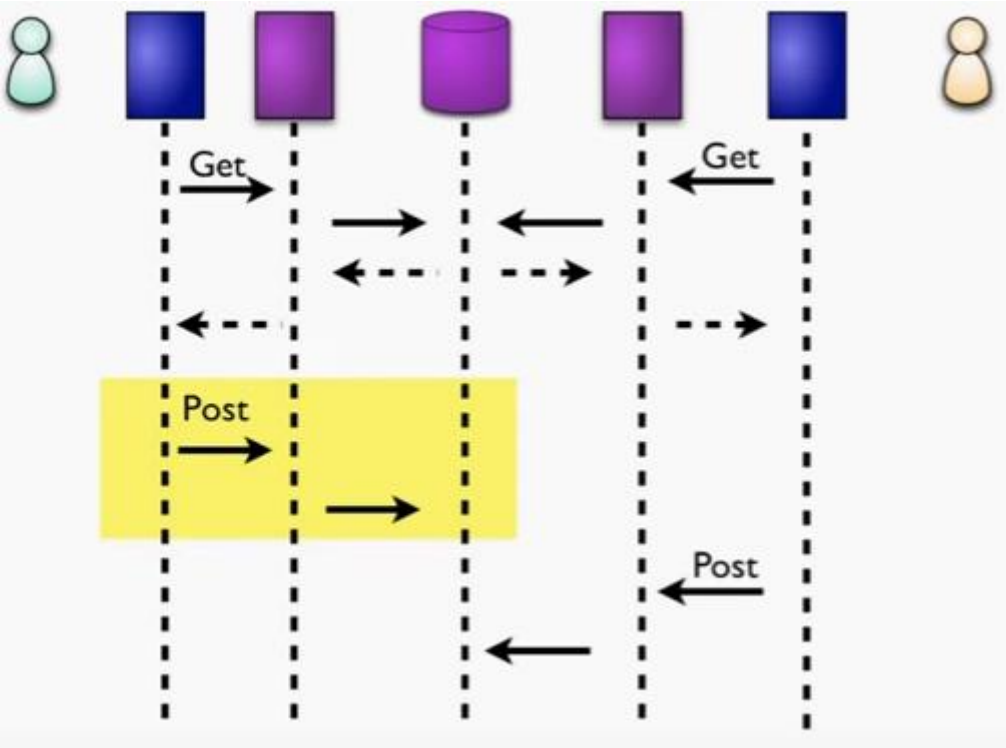
- Both people left and right take the same piece of data with a GET request. Essentially they bring it up onto their browser screen and both human beings think "I need to make some changes to this".



HO
GENT

Example

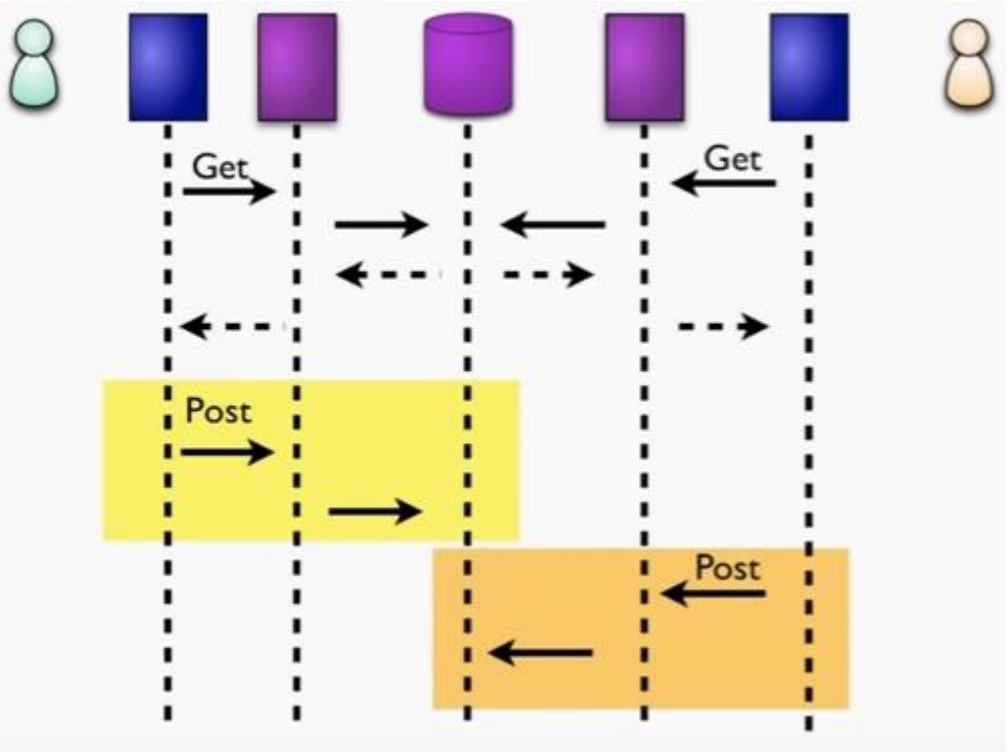
- Eventually the guy on the left says: "Okay I've got my updated data, let's post some changes."



HO
GENT

Example

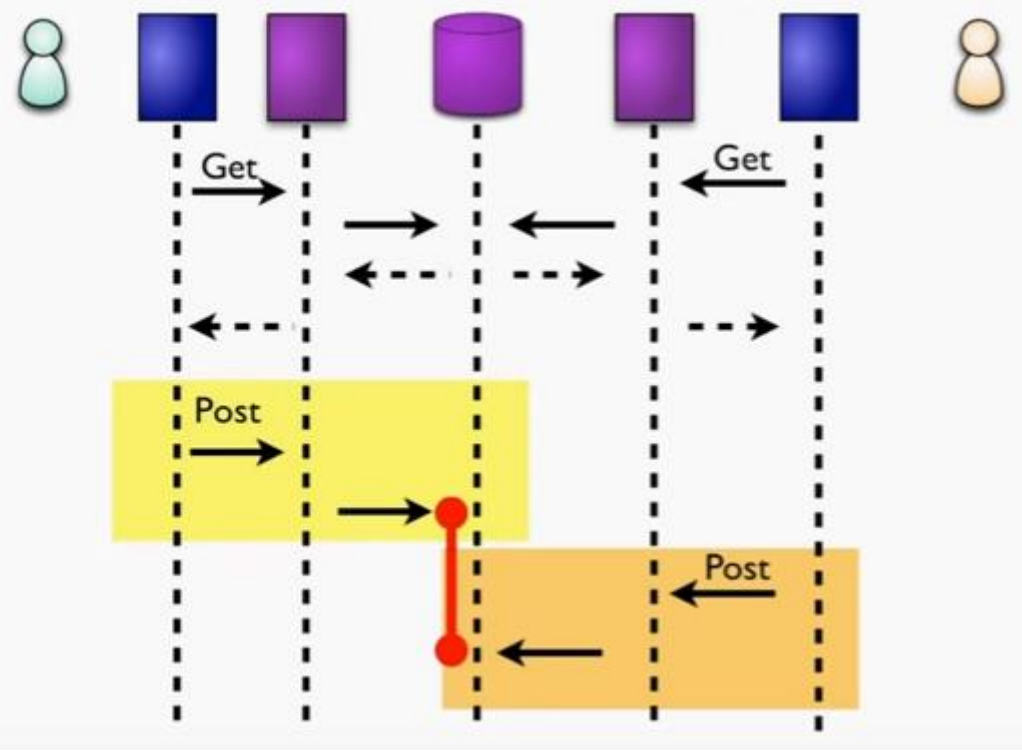
- Then shortly afterwards the guy on the right says: "I've updated my data now. Let's post some changes now."



HO
GENT

Example

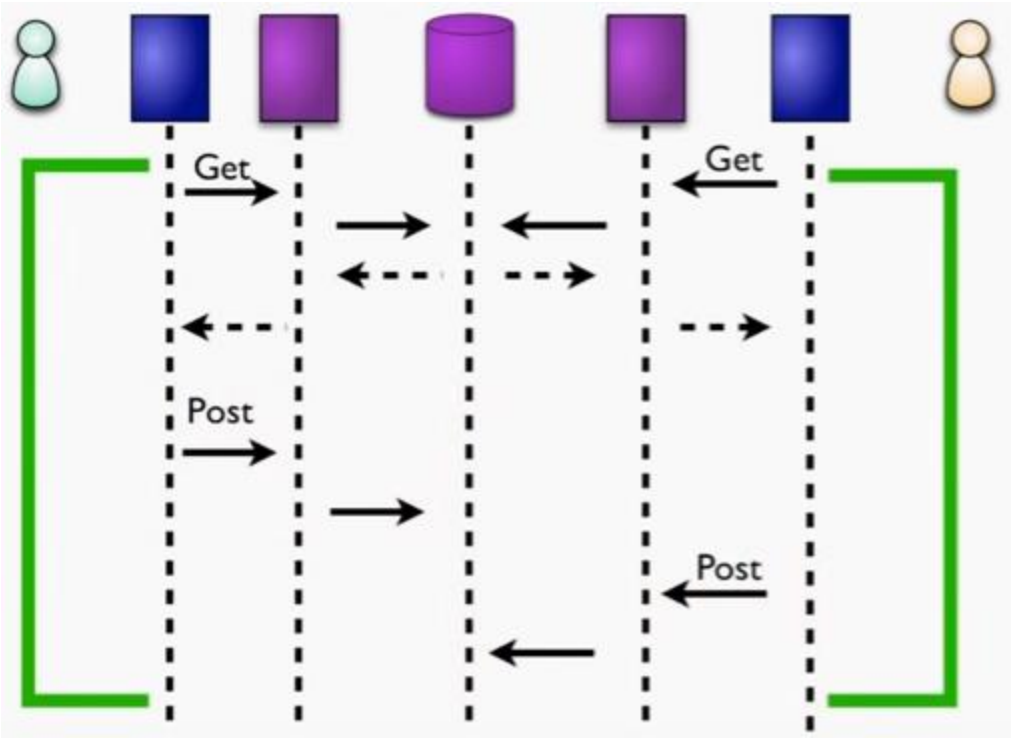
- Of course if you let this happen just like that, this is a write write conflict: two people have updated the same piece of information. They weren't aware of each other's update and they've got themselves in trouble.



HO
GENT

Example

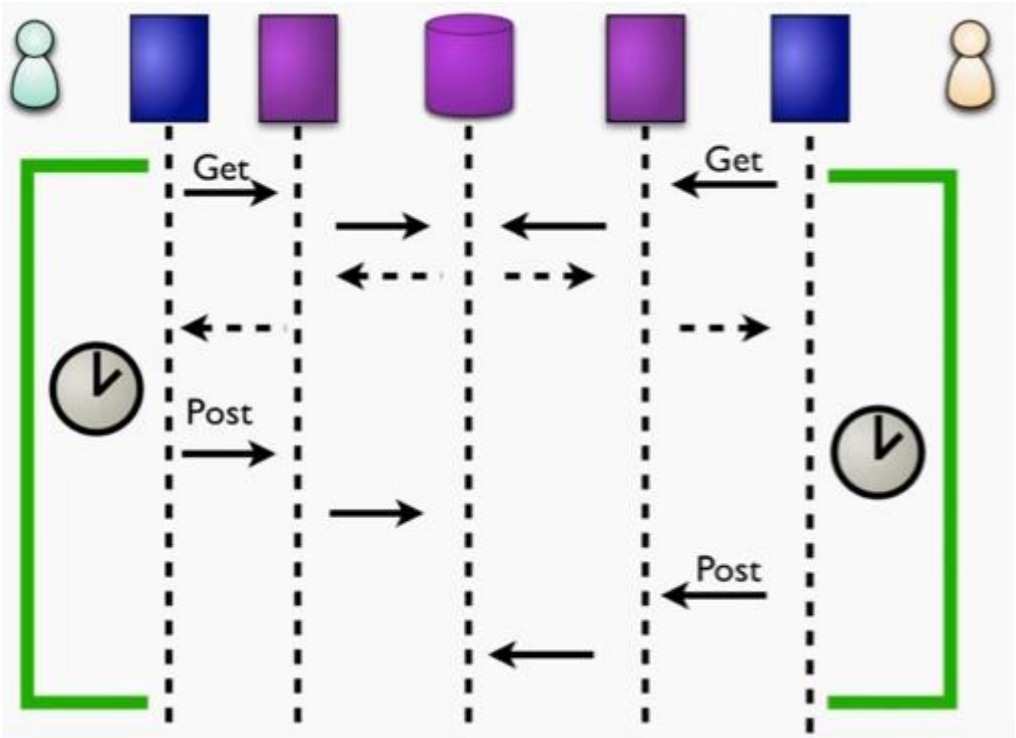
- ACID to the rescue.
- To prevent this conflict, you can wrap the entire interaction from getting the data on the screen and posting it back again in a transaction. That way, one of both will be told: "You have to do this again."



HO
GENT

Example

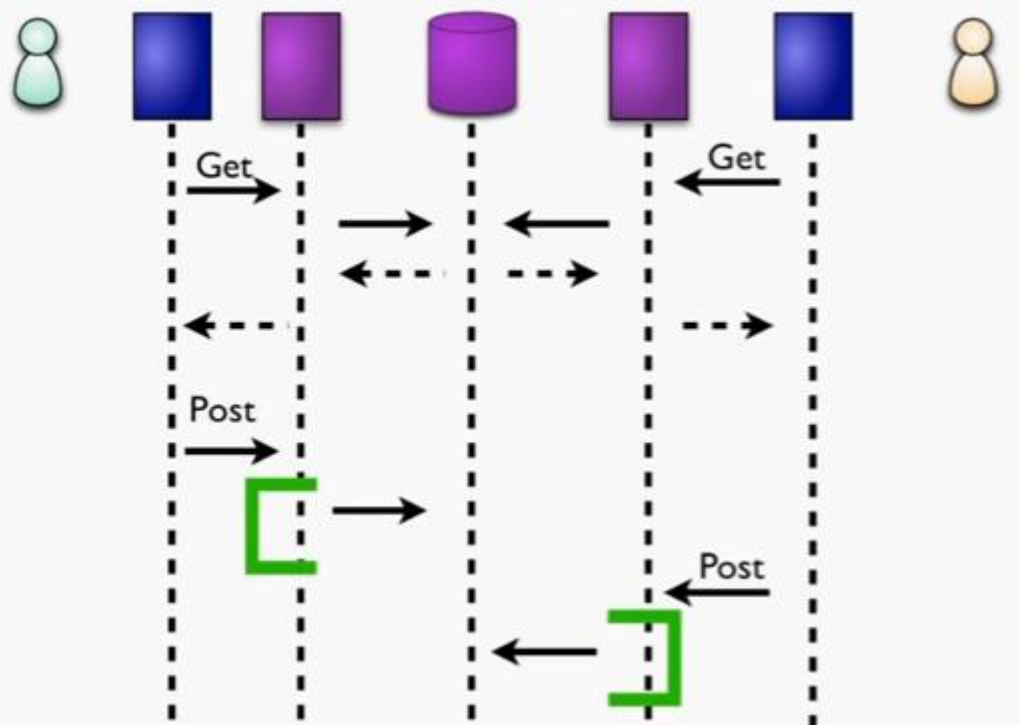
- Holding a transaction open for that length of time, while you've got a user looking and updating the data through the UI, that's going to really suck your performance out of your system.



HO
GENT

Example

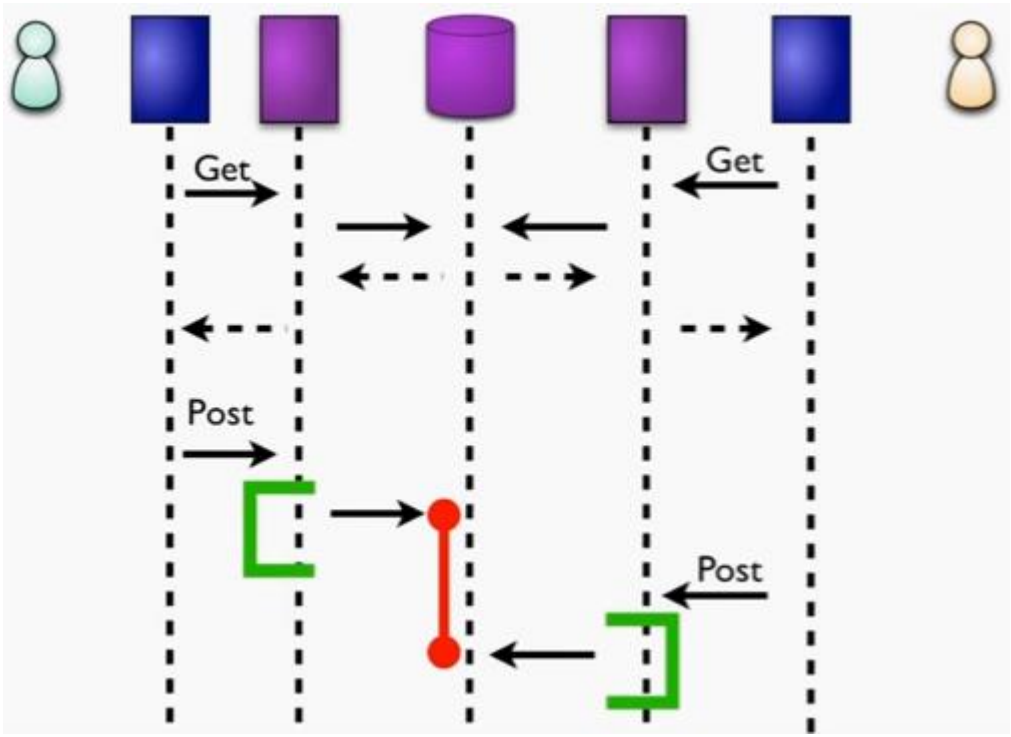
- Just wrap the transaction around the update



HO
GENT

Example

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



HO
GENT

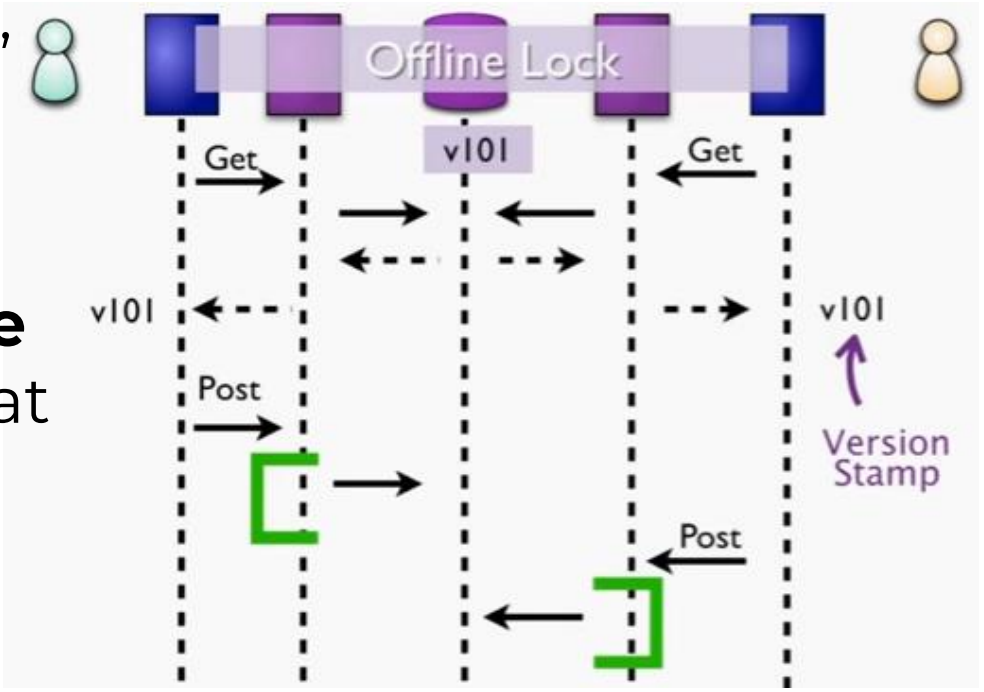
Example

- 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.

HO
GENT

Example

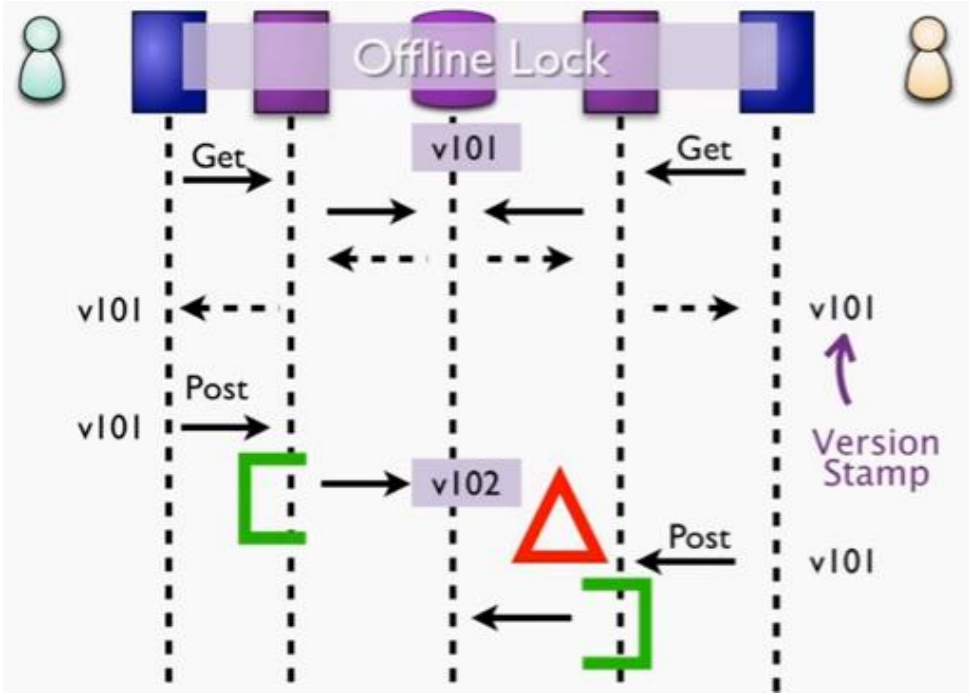
- If you come across this, you can solve this and basically use a technique which is referred to as an **offline lock**. Basically what that means: you give each data record or each aggregate a version stamp and when you retrieve it, you retrieve the version stamp with the aggregate data.



HO GENT

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.

HO GENT

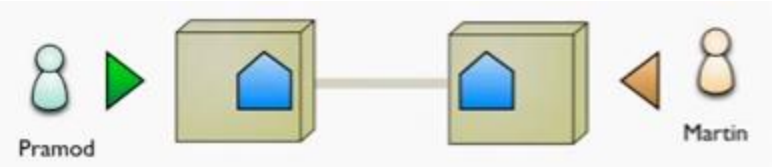
Consistency and availability

- 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

HO GENT

Consistency and availability

- As soon as you replicate data a new class of consistency problems starts coming in.
- This will be illustrated with a following example.
- 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

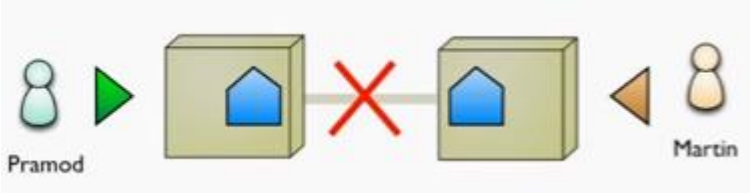


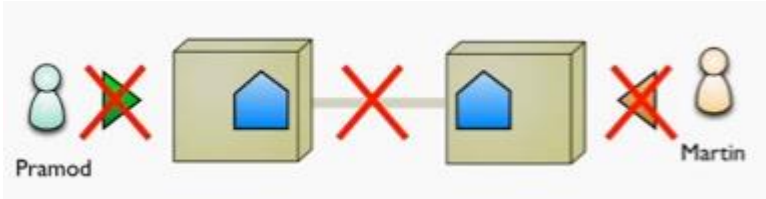
- The processing nodes need to communicate ensuring only one of both can book the hotel room

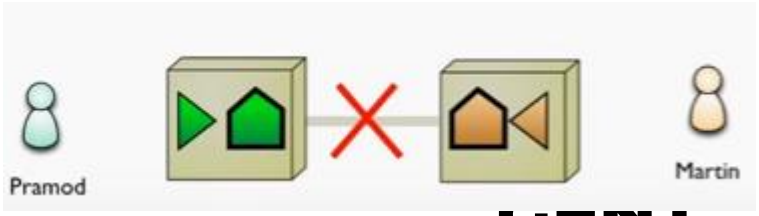
HO
GENT

Consistency and availability

- Again they both want to book a hotel room 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 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 risk of introducing an inconsistent behavior”)
- This is a business choice

HO
GENT

CAP theorem

- In most NoSQL databases the complex aspects of transaction processing are thrown overboard to gain speed.
- Therefore NoSQL systems are sometimes referred to as No ACID systems.
- The responsibility for the integrity of the data shifts from the DBMS to the application
- The DBMS will become faster, but the application will need extra time to validate the data.

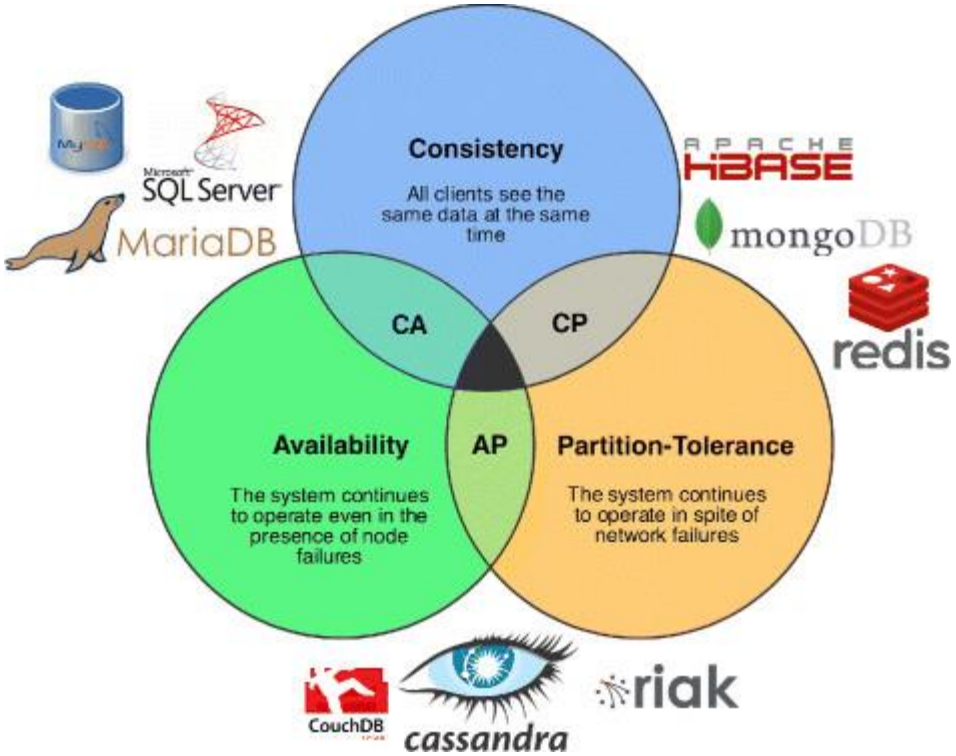
CAP theorem

- **Consistency**: The data in the database remains consistent after the execution of an operation. In a distributed context this means that after an update operation all replica's see the same data.
- **Availability**: The system is always on, no downtime. If a node or any other hardware – or softwarecomponent fails, the DBMS has to be able to handle this and should continue working using the replica's on other nodes.
- **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.

HO GENT

CAP theorem

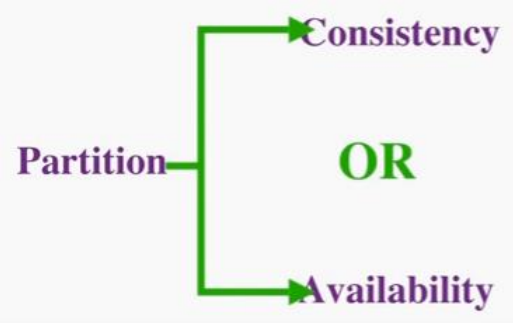
- As soon as you have a distributed system, you only can pick 2 out of 3
- This isn't a single binary choice. You can trade off levels of consistency and availability



HO GENT

CAP theorem

- NoSQL databases are distributed systems.
- You always have to take into account that the network can fail, so this leaves you with 2 choices: do you want to be consistent or do you want to be available.



HO
GENT

CAP theorem

- Consistency / Partition tolerance (CP). If the NoSQL DBMS needs to be consistent, then transactions and ACID characteristics are needed. The DBMS will be slower because of waiting times, time-outs or error messages that are needed when data consistency cannot be guaranteed, for example as a result of network partitioning
- Availability / Partition tolerance (AP). If the NoSQL DBMS always needs to be available then it will be faster because the DBMS always works with the most recently available version of the data, even if it can’t be guaranteed that these data are actually correct.

HO
GENT

BASE

- An AP NoSQL database has BASE characteristics
  - Basic availability → the availability of the data is more important than the consistency. The database will almost always be available.
  - Soft-state → the database will not always be in a consistent state. The validation of the data is postponed to the application and it's possible – through data distribution – that changes to the data aren't immediately available on each of the replica's.
  - Eventual consistency → after a while the validation of the data will take place and the changes will be available on each of the replica's. At some point in the future the database will eventual be consistent.

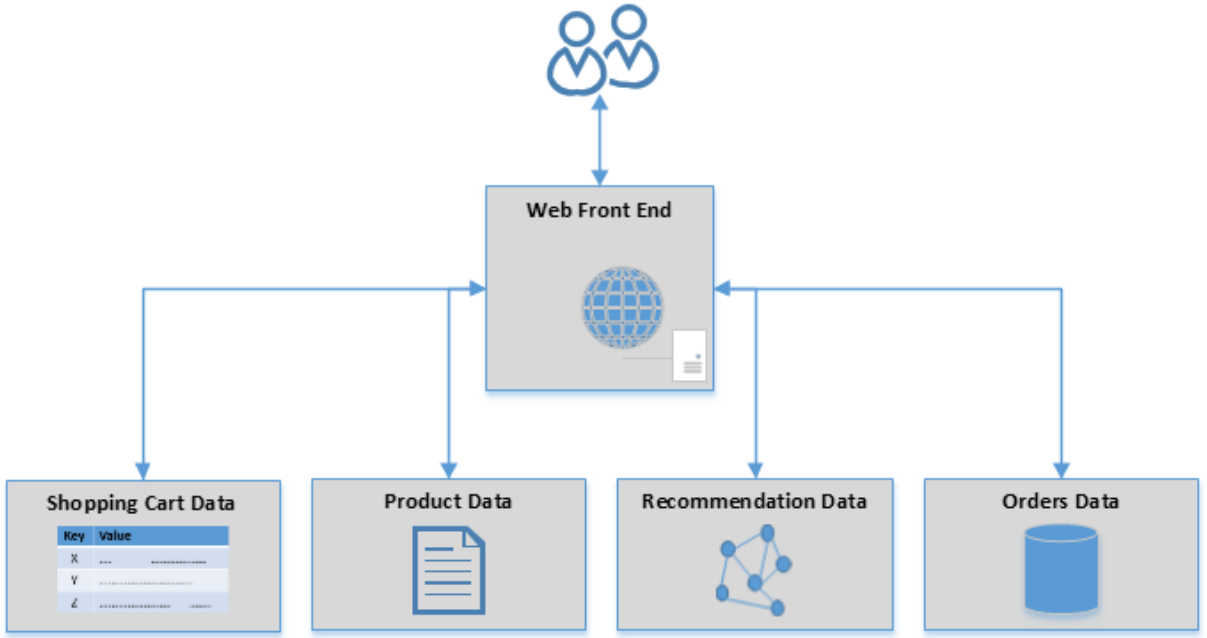
HO
GENT

Summary

HO
GENT

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 car apply across ar enterprise or within a single application.



GENT

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

HO
GENT

Exercises

HO
GENT

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

HO
GENT

What type of database would you use?

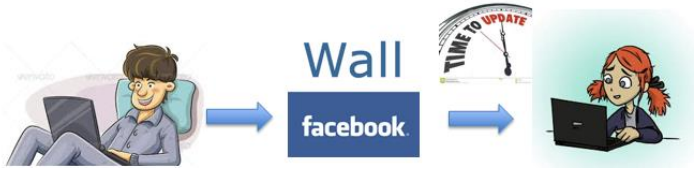
- (6) If your application needs programmer friendliness in the form of programmer friendly data types like JSON, HTTP, REST, JavaScript
- (7) If your application needs to dynamically build relationships between objects that have dynamic properties
- (8) If your application needs to cache or store BLOB data
- (9) If you want to create a website that shows the user a list of interesting workshops based on his interests.
- (10) If your application needs powerful offline reporting with large datasets

HO
GENT

Eventual consistency

- Illustrate the concept of Eventual consistency using Facebook.
  - Bob finds an interesting story and shares with Alice by posting on her Facebook wall
  - Bob asks Alice to check it out
  - Alice logs in her account, checks her Facebook wall but finds: Nothing is there!

  - Bob tells Alice to wait a bit and check out later
  - Alice waits for a minute or so and checks back: She finds the story Bob shared with her!



HO GENT