NoSQL databases

NOT ONLY SQL

Introduction

What NoSQL means

The term has been first used in '98 by Carlo Strozzi

- It referred to an open-source RDBMS that used a query language different from SQL

In 2009 it was adopted by a meetup in San Francisco

- Goal: discuss open-source projects related to the newest databases from Google and Amazon
- Participants: Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, MongoDB

Today, NoSQL indicates DBMSs adopting a different data model from the relational one

- NoSQL = Not Only SQL
- According to Strozzi himself, NoREL would have been a more proper noun

Strengths of RDBMSs

ACID properties

- Provides guarantees in terms of consistency and concurrent accesses

Data Integration and Normalization of schemas

- Several application can share and reuse the same information

Standard model and query language

- The relational model and SQL are very well-known standards
- The same theoretical background is shared by the different implementations
- This ensures application portability

Robustness

- Have been used for over 40 years

Weaknesses of RDBMS

Impedance mismatch

- Data are stored according to the relational model, but applications to modify them typically rely on the object-oriented model: objects or class definitions must be mapped to database tables defined by a relational schema but the two models has different expressivity, different constraints, ecc..
- Many solutions, no standard
  - E.g.: Object Oriented DBMS (OODBMS), Object-Relational DBMS (ORDBMS), Object-Relational Mapping (ORM) frameworks

Painful scaling-out

- Not suited for a cluster architecture
- Distributing an RDBMS is far from being easy

Consistency vs latency

- Consistency is a must – even at the expense of latency
- Today's applications require high reading/writing throughput with low latency

Schema rigidity

- Schema evolution is often expensive

NoSQL common features

Not just rows and tables

- Several data model adopted to store and manipulate data

Freedom from joins

- Joins are either not supported or discouraged

Freedom from rigid schemas

- Data can be stored or queried without pre-defining a schema (*schemaless* or *soft-schema*)

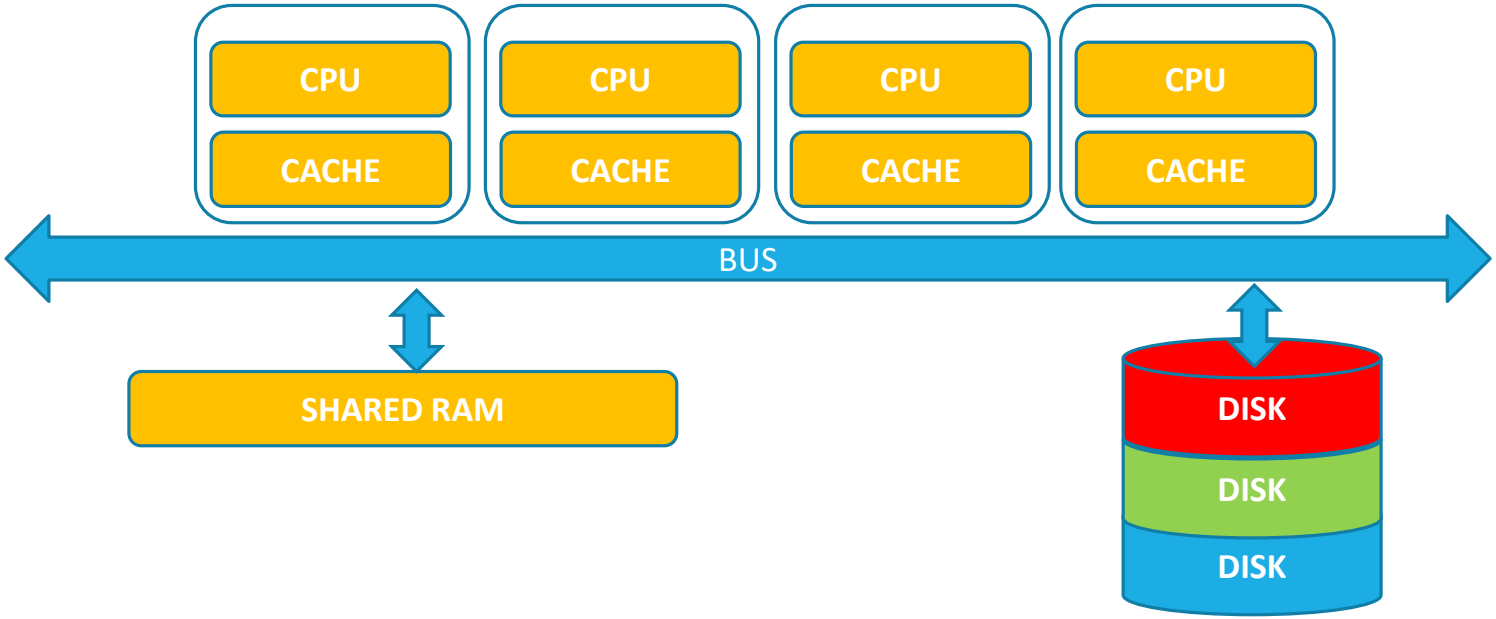
Distributed, shared-nothing architecture

- **Trivial scalability** in a distributed environment with no performance decay
- Each workstation uses its own disks and RAM

SMP Architecture

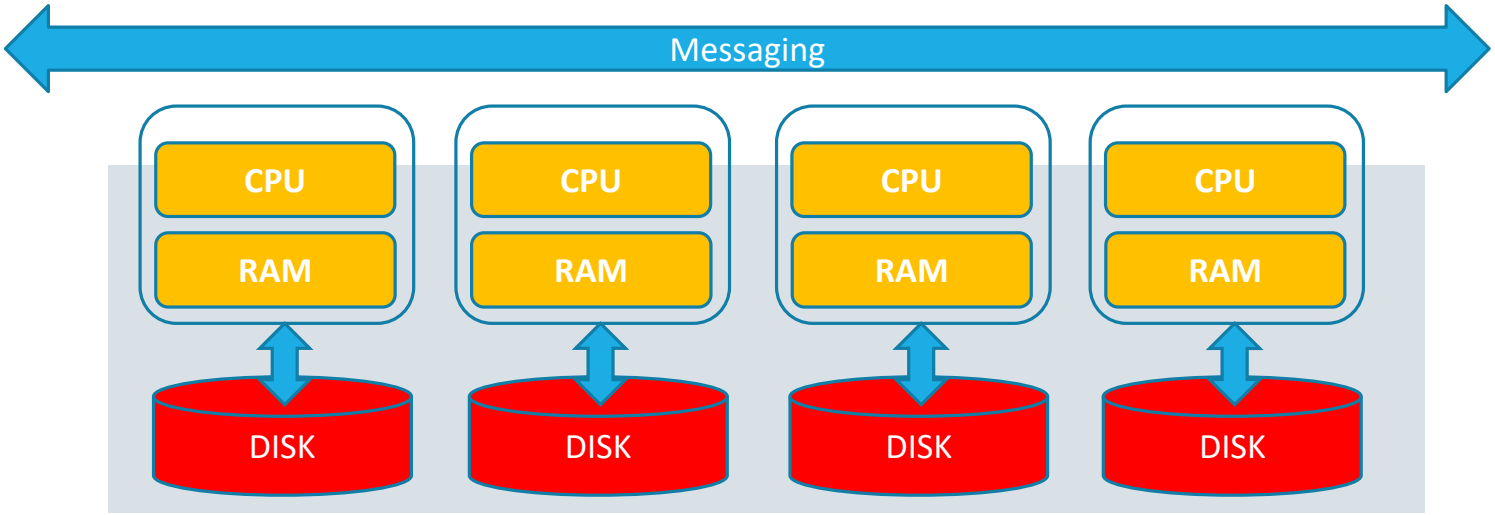
Traditional RDBMS are based on *Symmetric Multi Processing* (SMP) architecture: several processors share the same RAM, the same I/O bus and the same disk/disk array

- The limit of such architecture depends on the physical number of devices that can be mounted and by the BUS bottleneck



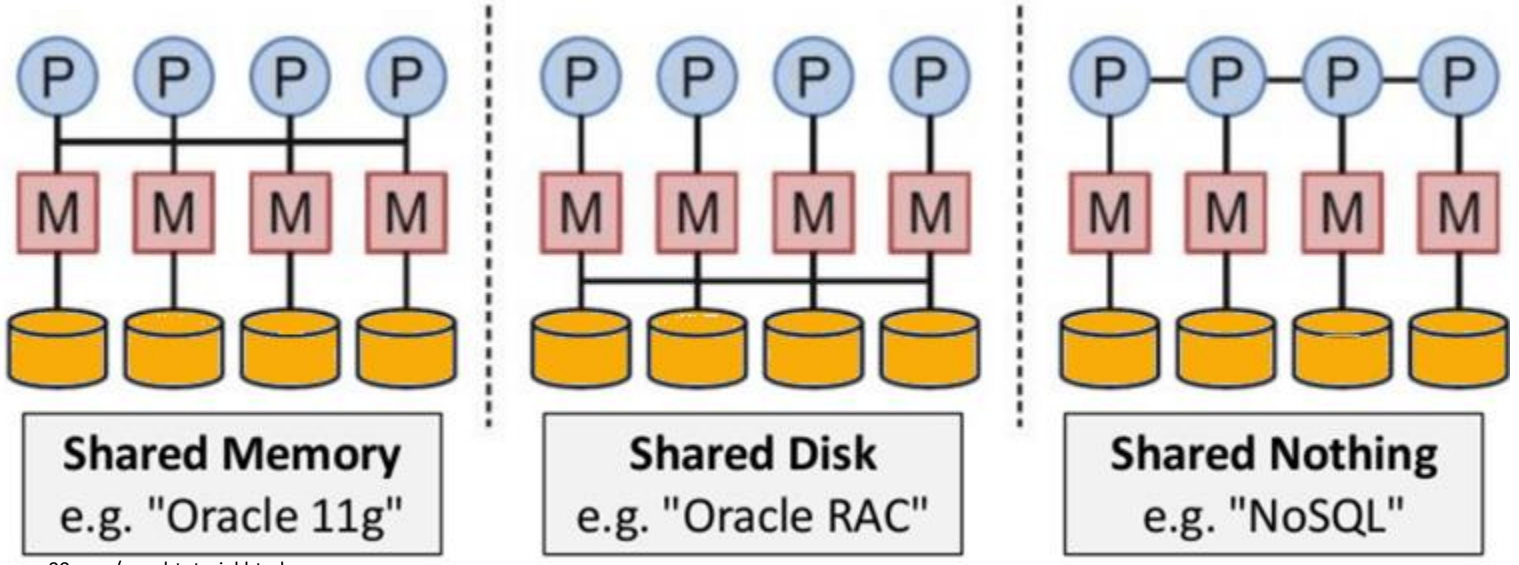
MPP Architecture

In a *Massively Parallel Processing* (MPP) Architecture several processors, each equipped with its own RAM and disks, collaborate to solve a single problem by splitting it in several independent tasks. It is also called shared nothing architecture



What about Oracle?

A standard Oracle database runs on a single instance. In the Oracle RAC architecture, the concept is different because some components are shared and others are dedicated for each instance.



Source: https://www.guru99.com/nosql-tutorial.html

Oracle RAC

Oracle Real Application clusters (RAC) allows multiple instances to access a single Oracle database. These instances often run on multiple nodes.

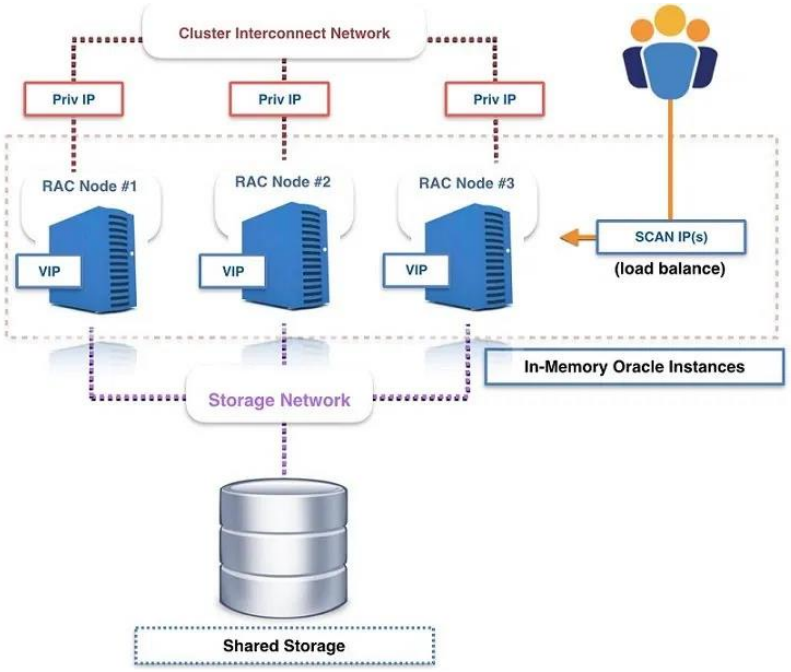
Oracle RAC is heavily dependent on the ***interconnect***, an efficient and high speed private network

Shared components through NAS, SAN technologies

- Control Files
- Datafiles
- Flash Recovery Log

Non-shared components:

- SGA
- local caches
- background processes



Source: https://www.oracle-scripts.net/rac-infrastructure-concepts/

Oracle RAC cache fusion

To ensure ACID properties **ORACLE RAC** must handle record locking and local caches on multiple instances without a centralized background process.

When a transaction T1 on instance A needs to access a record which is being locked by transaction T2 on instance B, the instance A will request instance B for that data block and hence accesses the block through the interconnect mechanism.

This concept is known as **CACHE FUSION** where one instance can work on or access a data block in other instance's cache via the high speed interconnect.

ORACLE RAC favors data consistency rather than throughput thus reducing scalability.

Source: https://www.oracle-scripts.net/rac-infrastructure-concepts/

Scalability

Scalability is ability of a system, network, or process to handle a growing amount of work in a capable manner or its ability to be enlarged to accommodate that growth.

Methods of adding more resources for a particular application fall into two broad categories: horizontal and vertical 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.
- To **scale vertically** (or **scale up**) means to add resources to a single node in a system, typically involving the addition of CPUs or memory to a single computer.

Vertical scalability is typically limited by the actual capability of the architecture to mount more resources and by the physical limitation of such resources (e.g. processor speeds). Nowadays, the availability of low cost commodity hardware and network connections make scaling out often more convenient than scale up.

What NoSQL isn't

Farewell to SQL

- Some systems do adopt SQL (or a SQL-like language)

Open-source

- There exist both open-source and commercial systems

Cloud Computing

- There exist both on premise and cloud solutions

Optimization of hardware resources

- At constant resources, a centralized RDBMS is probably better performing

NoSQL in the Big Data world

NoSQL systems are mainly used for operational workloads (OLTP)

- Optimized for high read and write throughput on small amounts of data

Big Data technologies are mainly used for analytical workloads (OLAP)

- Optimized for high read throughput on large amounts of data

In a way, NoSQL is to Big Data as the Operational Data Store is to the Data Warehouse

Can NoSQL systems be used for OLAP?

- Possibly, but only through Big Data analytical tools

The first NoSQL systems

LiveJournal, 2003

- Goal: reduce the number of queries on a DB from a pool of web servers
- Solution: Memcached, designed to keep queries and results in RAM

Google, 2005

- Goal: handle Big Data (web indexing, Maps, Gmail, etc.)
- Solution: BigTable (wide-column data model), designed for scalability and high performance on Petabytes of data

Amazon, 2007

- Goal: ensure availability and reliability of its e-commerce service 24/7
- Solution: DynamoDB (key-value data model), characterized by strong simplicity for data storage and manipulation

Sharding data

A LOOK BEHIND THE CURTAIN

Sharding data

One of the strengths of NoSQL systems is their scale-out capability

- The aggregate is well suited for being distributed within a cluster
- Graph databases do not scale as well as the others
- NoSQL systems can be used in a single server environment too

Two aspects must be considered when deploying on a cluster

- **Sharding**: distributing the data across different nodes
- **Replication**: creating copies of the data on several nodes
  - Master-slave
  - Peer-to-peer

Sharding & Replication Goals

Robustness

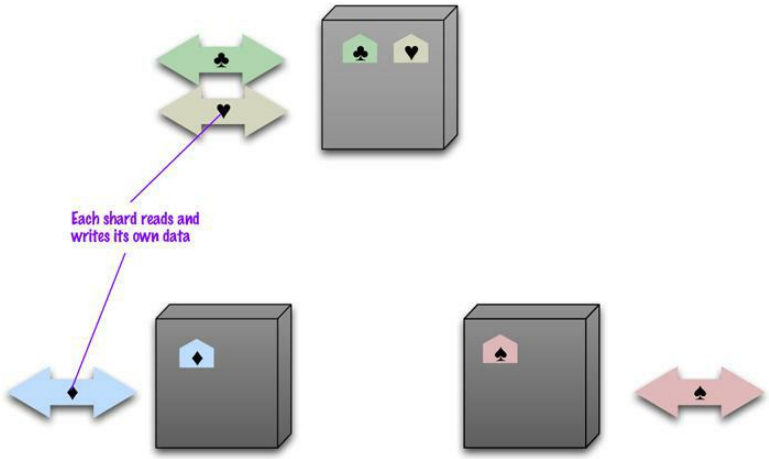
- Replication: in case of node failure, replicas prevent data loss

Efficiency

- Replication: improves efficiency since data can be read in parallel by several users on distinct copies. Furthermore, reads has more chances to happen locally to the node in charge to handle the replicas. If only one copy exists, it will have higher probability to be *far* from the node in charges to handle it, thus overall the system will occur higher network costs.
- Sharding: improves efficiency since different data partitions can be handled in parallel by different nodes

Sharding

Sharding: subdividing the data in *shards* that are stored in different machines



A good *sharding strategy* is **fundamental** to optimize performances

- Usually based on one or more fields composing the sharding key

Sharding strategy

Thumbs-up rules for a sharding strategy:

1. Data-locality: stores the data close to those that need to access them
   - E.g., store orders of Italian customers in the European data center
2. Keep a balanced distribution
   - Each node should have the same percentage of data (more or less)
3. Respect the domain optimization goal

Optimization Goals

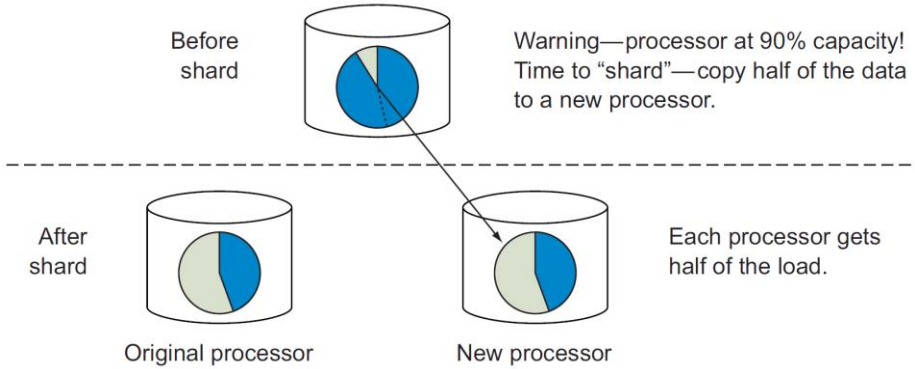
Depending on the workload features two optimization goals can be pursued

1. Inter-query balancing: Useful when the workload is composed by several light queries (i.e. queries accessing few data)
   - Store data to be accessed by each query together in order to avoid network costs
   - Balance the workload by distributing the workload, that is: different queries will access different nodes
   - **This is the case for operational workload (e.g. shopping carts, user profiles)**
2. Intra-query balancing: Useful when the workload is composed by few heavy queries (i.e. queries accessing tons of data)
   - Store in separate nodes data accessed by a single query in order to exploit several nodes to handle it
   - Balance the workload by distributing each single query
   - **This is the case for analytic workload (e.g. OLAP queries)**

Auto-sharding

Many NoSQL systems offer auto-sharding policies

- The database distributes the data according to the workload



Beware: redefining (or choosing later) the sharding strategy can be quite expensive

Sharding strategy

Sharding is carried through an attribute of the record called **sharding key**

There are some typical strategies for sharding:

- **Range strategy:** each partition contains a range of data **sorted** on the basis of a key;
  - HBASE and Google spanner adopts this strategy
  - **Pros:** efficiently run range queries that works on the sharding key values. All the rows are stored at the same node
  - **Cons:** some nodes can become bottlenecks if workload focuses on a specific range of values
  - **Cons:** ranges are defined a priori and this can determine heavy data redistribution

- **Hash strategy:** a **hash function** is used to allocate data to partitions
  - DynamoDB and Cassandra adopts this strategy
  - **Pros:** ideal for *intra-query balancing*
  - **Pros:** new nodes can be added to the cluster without requiring heavy data redistribution
  - **Cons:** range queries become inefficient when *inter-query balancing* is the goal
  - **Pros**/**Cons:** depending on the hash function new nodes can be added to the cluster without requiring heavy data redistribution. For example Consistent hashing avoid the problem, while *X mod N* would redistribute all the keys if on node is added to the cluster (i.e. #nodes = *N+1)*

Replication

Replication: the data is **copied** on several nodes

How to distribute the replicas?

- If the cluster is rack-aware, same strategy as for HDFS
  - One replica in the node receiving the data
  - One replica in a different node of another rack
  - One replica in a different node of that same rack

Each update must be pushed to every replica

Two techniques to handle updates:

- Master-slave
- Peer to peer

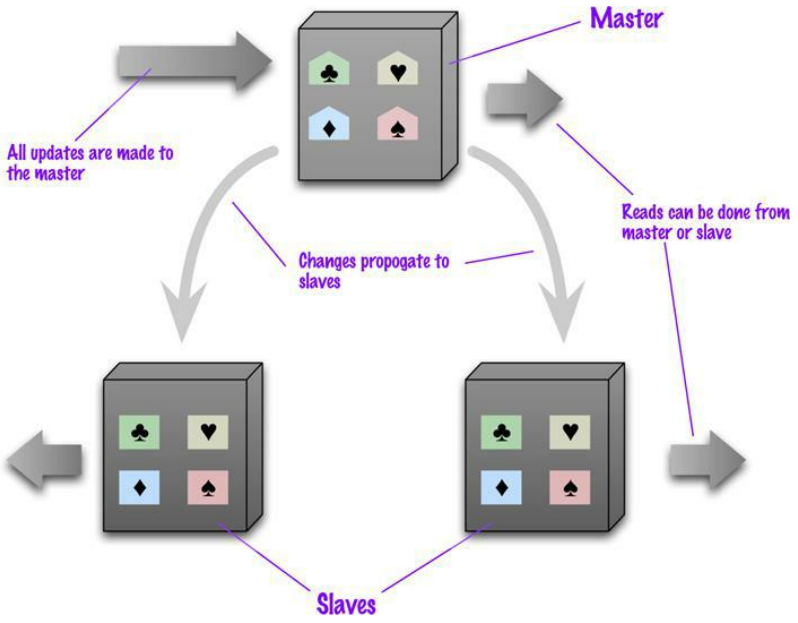
Master-Slave Replication

Master

- It's the manager of the data
- Handles each and every write operation
- Can be chosen or drawn

Slaves

- Enable read operations
- In sync with the master
- Can become master if the latter fails



Master-Slave Replication

Pros

- Easy handles many read requests
  - Slaves do not need the master to perform reads
- Useful when the workload mainly consists of reads

Cons

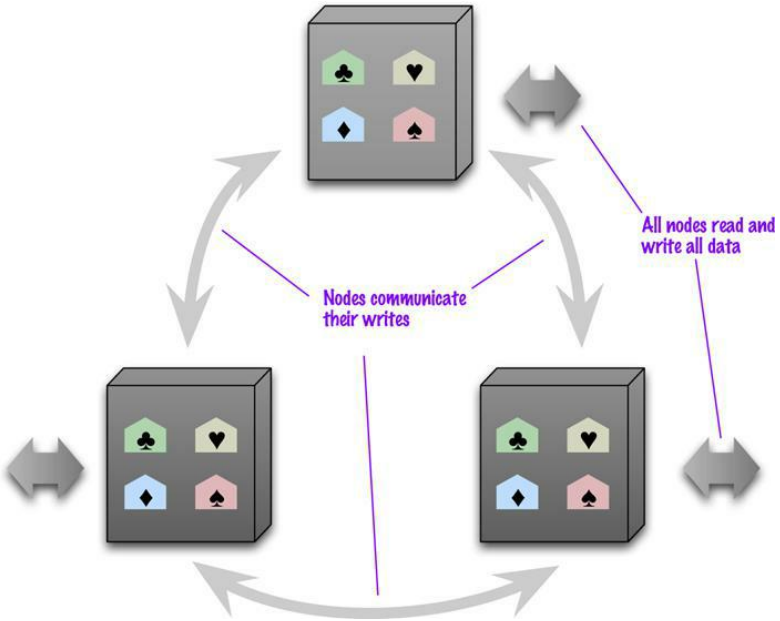
- The master is a bottleneck
  - **Only the master can handle writes**
  - In case of failure, a new master must be drawn
- Write propagation can be a source of inconsistency
- Not ideal when the workload mainly consists of writes

Peer-to-Peer Replication

Each node has the same importance

Each node can handle write operations

The loss of a node does not compromise reads nor writes



Peer-to-Peer Replication

Pro

- The failure of a node does not interrupt read nor write requests
- Write performances easily scale by adding new nodes

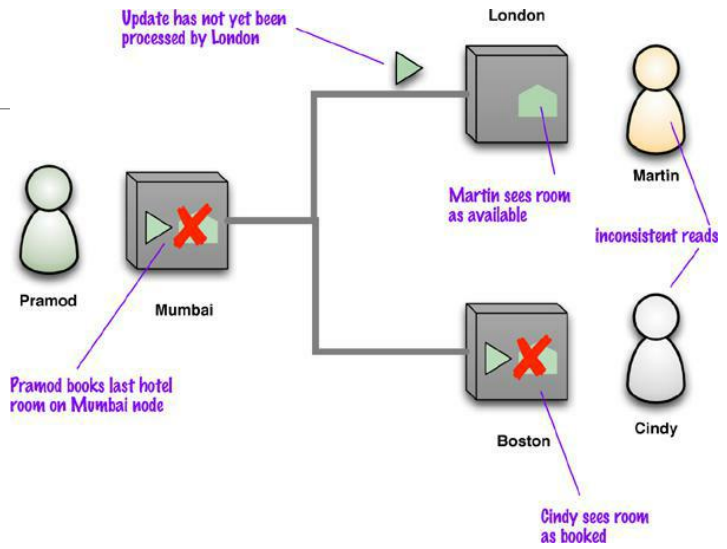
Cons

- Inconsistency!
- Write propagation can be a source of inconsistency
  - Two users may read different values at the same time
  - Read inconsistency can be problematic, but are relatively limited in time
  - The same problem occurs with master-slave replication
- Two users may update the same value from different replicas
  - Write inconsistencies are way worse

Source of Inconsistency under Replication

Delays in updates

- Write propagation is not immediate
- Two users can access different replicas finding different values
- 1 update several reads or concurrent updates
- Take place both in the master-slave and peer-to-peer models

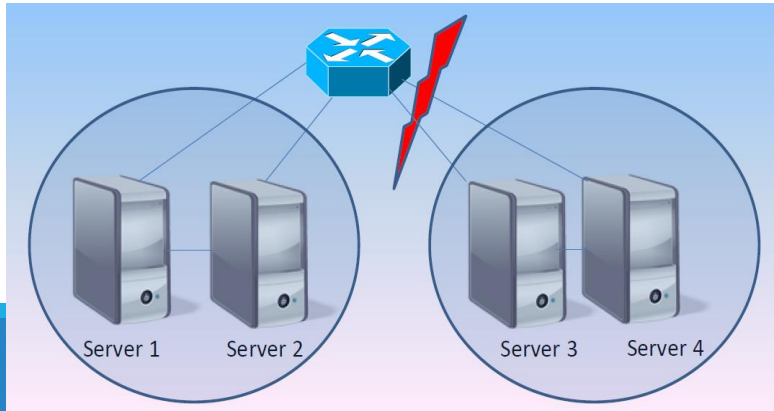


Network partitioning

- Due to network errors nodes in different partitions cannot communicates or some nodes may not be available
- Each group must decide independently whether it is allowed to carry out operations.

Applications that update many records

- … when the system ensures consistency for single records only (no ACID properties)



Handling conflicts due to delays

1 update several reads – read conflict

- Tolerate conflicts: different users reading different values of the same data at the same time
- Read-your-writes mechanism (i.e., reading consistency is guaranteed for the data written by the same user)
  - An app A updates record R on node N1
  - N1 returns a commit token C to A. C carries out a versioning of the record
  - A reads R from N2 along with C. N2 can answer only its R version is current enough with refererence to C

Concurrent updates – write conflict

- Possible in the peer-to-peer model only
- Based on update time
  - Last write wins: in case of conflict, the latest update overrides the others
  - Conflict prevention: enforce writes on the most recent version
  - Conflict detection: preserve history and let the user decide

Handling conflicts due to Network Partitioning

Beside delays, inconsistencies may be due to network partitioning

- Due to network errors nodes in different partitions cannot communicates or some nodes may not be available
- Inconsistencies due to network partitioning have a larger time windows (last longer) than those due to delays in updating replicas
- Each group must decide independently whether it is allowed to carry out operations.
- A **quorum** is the minimum number of votes that a distributed operation has to obtain in order to be allowed to perform an operation on a replicated data item

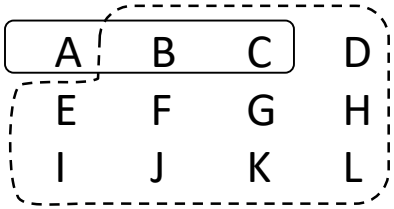
The quorum mechanism

The **quorum mechanism** ensures consistent reads and writes under replication

- Based on contacting a majority of the nodes responsible for certain data
- Each copy of a replicated data item is assigned a vote.
- Each operation has to obtain a *read quorum* (R) or a *write quorum* (W) to read or write a data item, respectively.
- If a given data item has a total of V votes, the quorums have to obey the following rules:

Given a data item with V votes (i.e. number of replicas), proper quorums must obey the following rules:

- **Writing quorum:** W > V/2
  - Ensures that two write operations from two transactions cannot occur concurrently on the same data item.
- **Reading quorum:** R + W > V
  - Ensures that a data item is not read and written by two transactions concurrently



N = 12
W = 11
R = 3

Consistency in NoSQL: summary

Source	Cause	Effect	Solution
Replication (MS and P2P)	Write propagation delay between replicas is slow	Read conflicts	- Tolerate - Read-your-writes - Quorum
Replication (P2P)	Two write operations can be issued on different replicas	Write conflicts	- Last write wins - Conflict prevention - Conflict detection - Quorum
Network partitioning	Inability to communicate with all replicas of a certain data	- Read conflicts - Possibly write conflicts	- Relax CAP - Prevent write conflicts - Handle read conflict as above
No ACID transactions	- An update over multiple records fails mid-query - Two updates over multiple records are interleaved	Unrecoverable inconsistency	- Each system provides its own mechanism to offer limited ACID-like transactions
Data de-normalization	The same data is repeated in different instances with different values	Inability to find the correct values	- Avoid denormalization if strong consistency is needed - Data cleaning before analysis

Managing consistency

A LOOK BEHIND THE CURTAIN

RDBMS vs NoSQL: different philosophies

RDBMS come from decades of widespread usage

- Strong focus on data consistency
- Years of research activities to optimize performances
- Highly complex systems (triggers, caching, security, etc.)

NoSQL systems are designed to succeed where RDBMSs fail

- Strong focus on short latency and high availability
- Currently, quite simple systems
- Speed and manageability rather than consistency at all costs

RDBMS vs NoSQL: different philosophies

RDBMS pros

- Consistency and concurrency handles by the DBMS
- Trustworthy and sound security policies and mechanisms
- SQL standard: same code, different technologies
- Schema and constraints guarantee data quality
- Well known ER model and SQL
- Immediate integration with analytical tools

NoSQL pros

- Schema definition is not necessary to begin working
- Efficient data sharding
- Linear scalability with respect to the number of nodes in the cluster
- (Possibly) integrated text-search capabilities
- Freedom from ORM frameworks
- Easy management of high-variety data

Consistency: an example

Consider 1000€ to be transferred from bank account A to B; the transfer is made by:

- Removing 1000€ from A
- Adding 1000€ to B

What should never happen

- The money is removed from A but not added to B
- The money is added twice to B
- A query on the database shows an intermediate state
  - E.g., A+B = 0€

RDBMS adopt transactions to avoid this kind of issue

Consistency in RDBMSs: ACID

Transactions guarantee four fundamental properties: ACID

Atomicity
- The transaction is indivisible: either it fully completes, or it fails
- It cannot be completed partially

Consistency
- The transaction leaves the DB in a consistent state
- Integrity constraints can never be violated

Isolation
- The transaction is independent from the others
- In case of concurrent transactions, the effect is the same of their sequential execution

Durability
- The DBMS protects the DB from failures

Consistency in RDBMSs: ACID

Implementation of ACID properties relies on a locking mechanisms and logs

- Resources are locked, updates are logged
- In case of problems, rollback to the original state
- If no error occurs, unlock the resources

Consistency is guaranteed to the detriment of speed and availability

- User may have to wait
- Hard to replicate this mechanism in a distributed environment

But, sometimes, consistency is not that important

- E.g.: e-commerce application
- Shopping cart management requires speed and availability
- Order emission requires consistency

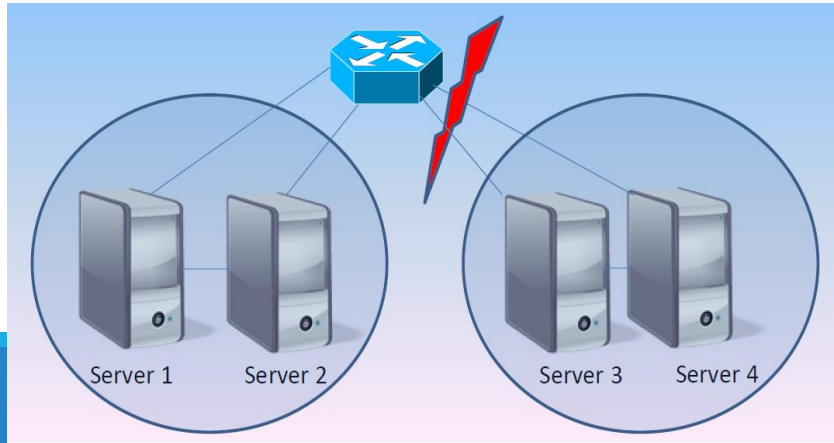
Consistency in NoSQL

Several attempts have been made to *describe* NoSQL properties with respect to ACID properties

- CAP theorem
- BASE properties
- PACELC theorem

The main cause of complexity increase in distributed database is **Network partitioning** that happens when a network failure causes the isolation of a part of nodes in a collection.

- It can occur within a datacenter or between datacenters.
- An arbitrary number of messages is lost



Consistency in NoSQL: CAP

Prof. Eric. A. Brewer
(Berkeley, Google)

"Theorem": only two of the following three properties can be guaranteed

Consistency: every node in a distributed cluster returns the same, most recent, successful write. Consistency refers to every client having the same view of the data.

- There are various types of consistency models. Consistency in CAP (used to prove the theorem) refers to linearizability or sequential consistency, a very strong form of consistency.

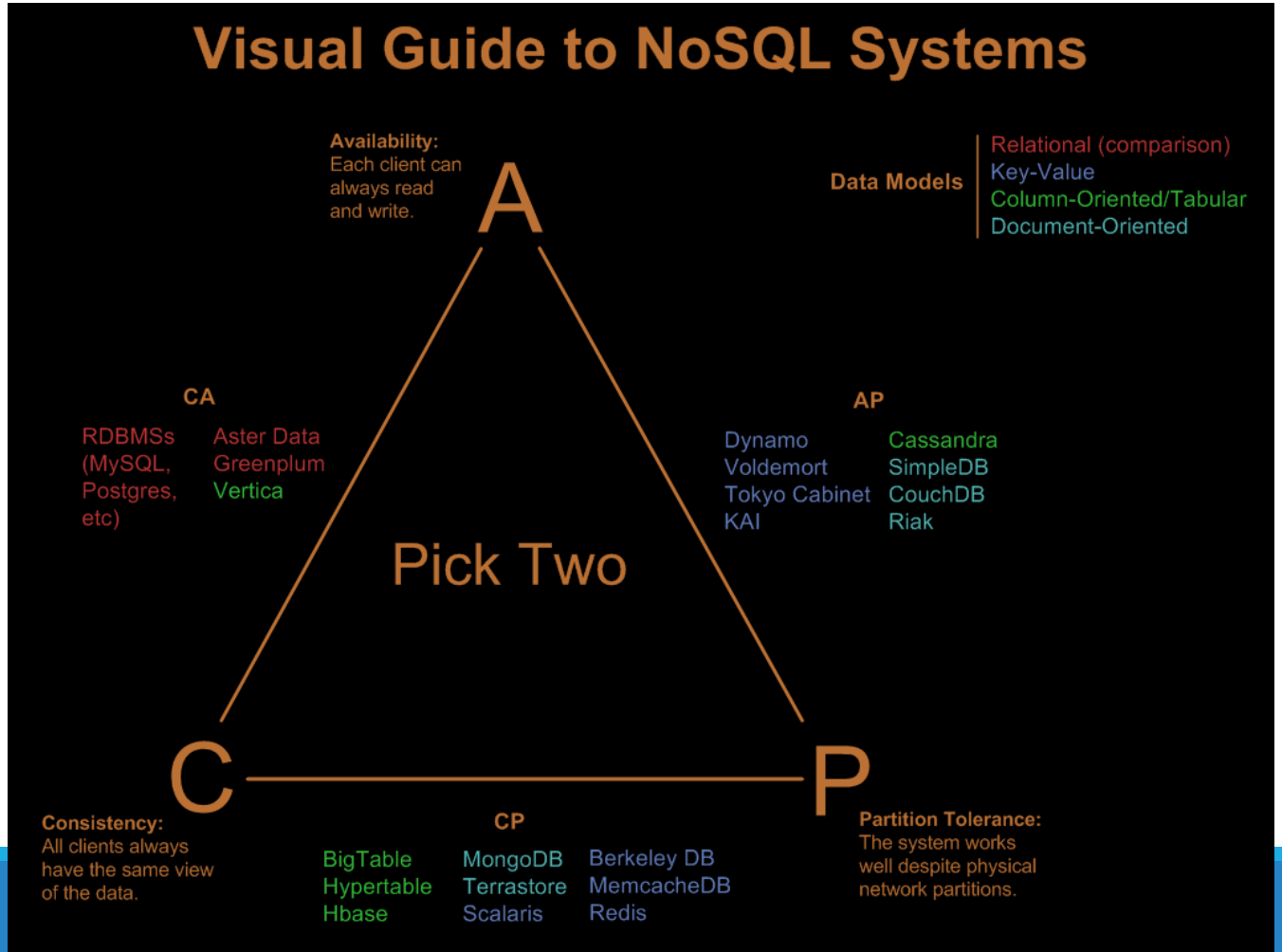
Availability: Every non-failing node returns a response for all read and write requests in a reasonable amount of time. The key word here is every. To be available, every node on (either side of a network partition) must be able to respond in a reasonable amount of time.

Partition tolerance: The system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

Theorem interpretation is not trivial

- Asymmetric properties: consistency is sacrificed to favor speed at all times, not just when partitioning happens
- Partition tolerance, consistency and availability behaves differently based on how strict is the requisite and the algorithms implementing them

Consistency in NoSQL: CAP



Consistency in NoSQL: CAP

Prof. Eric. A. Brewer
(Berkeley, Google)

Three situations

- CA: the system cannot suffer from network partitioning (single server)
- AP: in case of partitioning, the system sacrifices consistency (overbooking)
- CP: in case of partitioning, the system sacrifices availability (bookings prevented)
- Since different DBMSs make different choiches, choosing the right DBMS depends on the application constraints:

When network partitioning is plausible:

DBMSs enforcing CP: when it is imperative that all clients have a consistent view of the database, users of a node will have to wait for any other node to agree before they can read or write to the database, availability is affected since users have to wait an unspecified period of time. HBase supports such CP's model

DBMSs enforcing AP: when the database must remain available at all times, DBMSs that allow clients to write data to a node in the database without waiting for other nodes to agree. The DBMS then takes care of the reconciliation of the data at a later time. This is the final state of consistency. In applications that could sacrifice data consistency in exchange for huge performance, databases such as CouchDB, Cassandra can be selected.

Consistency in NoSQL: BASE

The CAP theorem is often cited as a justification for the use of weaker consistency models, for example **BASE** (Basically Available Soft-state (services with) Eventual consistency).

Basic **A**vailability

- The system should always be available

Soft-state

- It is acceptable for the system to be temporarily inconsistent

Eventual consistency

- Eventually, the system becomes consistent
- The most important property

Consistency in NoSQL: BASE

ACID

- Pessimistic approach (better safe than sorry)

BASE

- Optimistic approach (everything is going to be ok)
- Higher throughput is better than enforcing consistency

In summary, the BASE model is characterized by a high availability for first level services; a cleaning mechanism is adopted to solve the problems created by optimistic actions which then appear to have violated the consistency.

Consistency in NoSQL: PACELC

Prof. Daniel Abadi
(Maryland)

Evolution of the CAP theorem (less known, but more precise)

- if (**P**artition) then { **A**vaialability or **C**onsistency? }
- **E**lse { **L**atency or **C**onsistency? }

Different behavior in case or in absence of partitioning

- PA: in case of partitioning, the system sacrifices consistency (overbooking)
- PC: in case of partitioning, the system sacrifices availability (bookings prevented)
- EL: otherwise, the system sacrifices consistency in favor of speed
- EC: otherwise, the system sacrifices speed in favor of consistency

Four situations:

- PA EL: system focused on speed and availability (main NoSQL philosophy)
- PA EC: consistency sacrificed only when partitioning happens
- PC EL: consistency enforced only when partitioning happens (e.g., Yahoo Sherpa)
- PC EC: system focused on consistency (RDBMS)

Consistency in NoSQL: relaxing CAP

Consider two users that want to book the same room when a partitioning happens

CP: no one can book (A is sacrificed)
- Not the fastest solution

AP: both can book (C is sacrificed)
- Possible overbooking: writing conflict to handle

caP: only one can book
- The other will see the room available but cannot book it

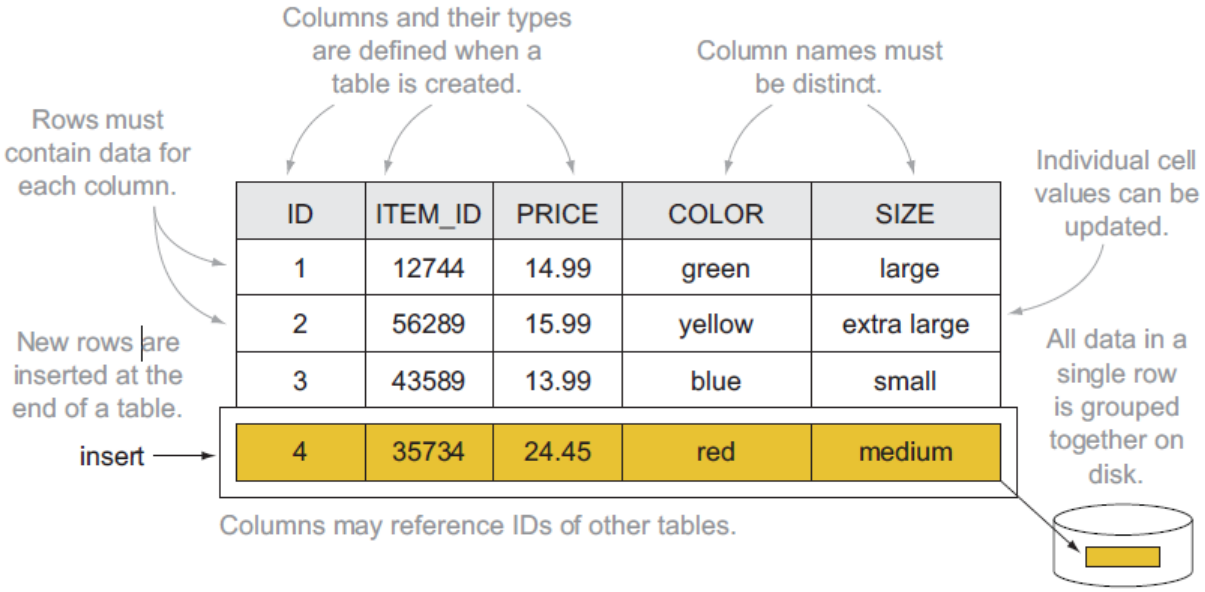
This is admissible only in certain scenarios
- Finance? Blogs? E-commerce?

It's important to understand:
- What is the tolerance in reading obsolete data
- How large can the inconsistency window be

Data models

Relational model

Based on tables and rows

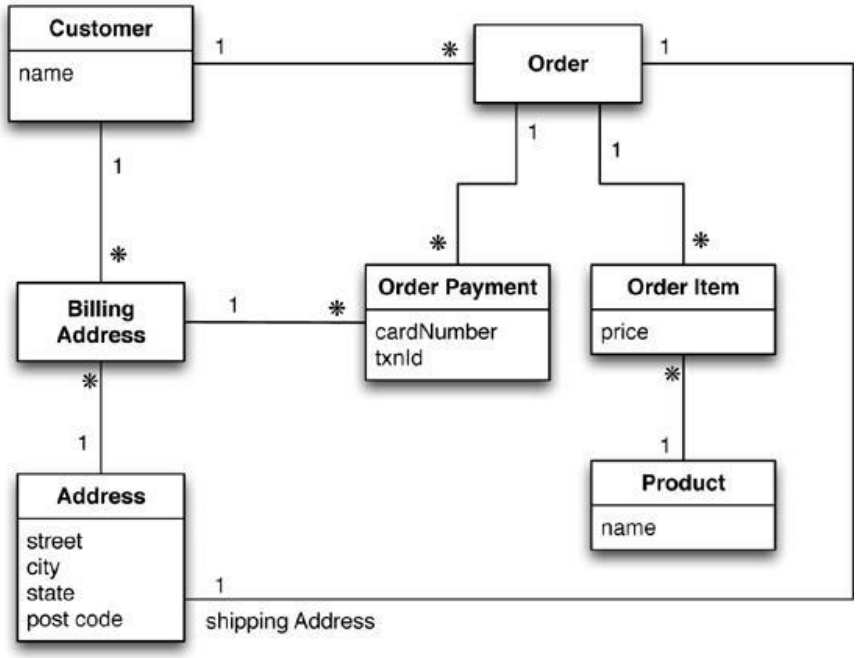


NoSQL: several data models

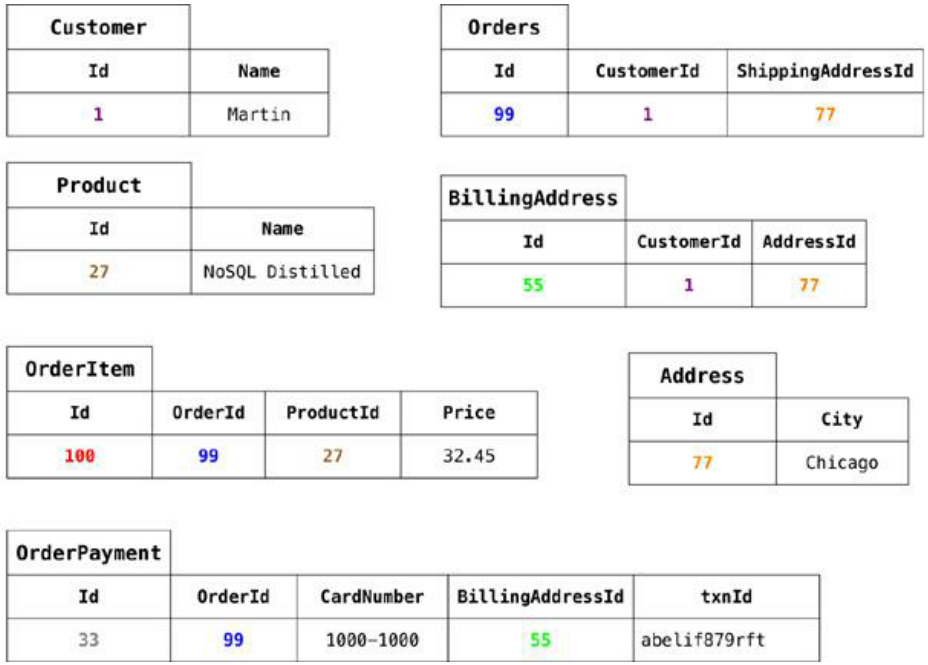
One of the key challenges is to understand which one fits best with the required application

Model	Descritption	Use cases
Key-value	Associates any kind of value to a string	Dictionary, lookup table, cache, file and images storage
Document	Stores hierarchical data in a tree-like structure	Documents, anything that fits into a hierarchical structure
Wide column	Stores sparse matrixes where a cell is identified by the row and column keys	Crawling, high-variability systems, sparse matrixes
Graph	Stores vertices and arches	Social network queries, inference, pattern matching

An ER running example



Relational modeling



Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Key-value: data model

Each DB contains one or more collections (corresponding to tables)

Each collection contains a list of key-value pairs

- Key: a unique string
  - E.g.: ids, hashes, paths, queries, REST calls
- Value: a BLOB (binary large object)
  - E.g.: text, documents, web pages, multimedia files

Key	Value
image-12345.jpg	Binary image file
http://www.example.com/my-web-page.html	HTML of a web page
N:/folder/subfolder/myfile.pdf	PDF document
9e107d9d372bb6826bd81d3542a419d6	The quick brown fox jumps over the lazy dog
view-person?person-id=12345&format=xml	<Person><id>12345</id .</Person>
SELECT PERSON FROM PEOPLE WHERE PID="12345"	<Person><id>12345</id .</Person>

Atomicity level: the key-value pair

Sharding: based on keys

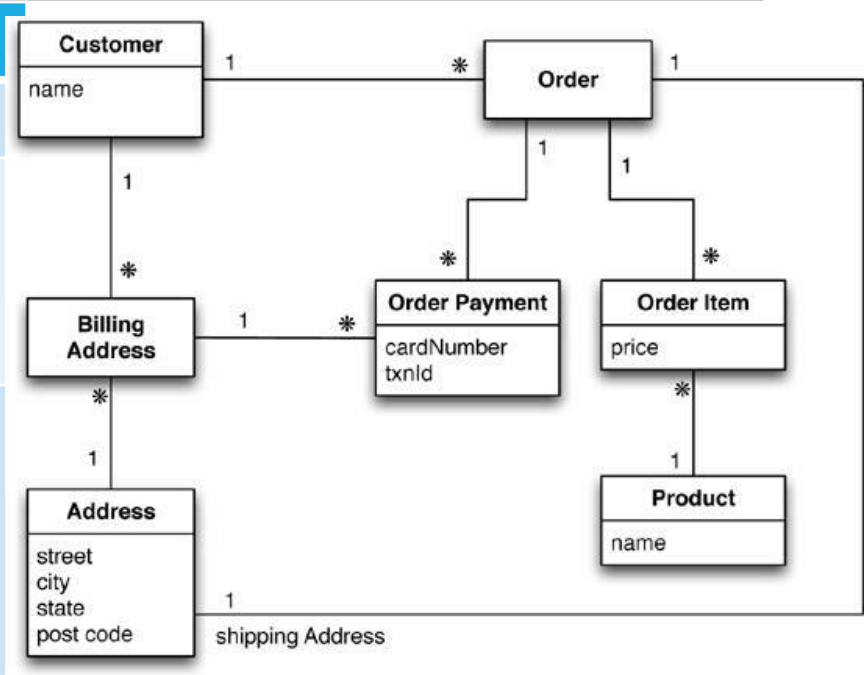
Looks like a simple dictionary

- The collection is indexed by key
- The value may contain several information: one or more definitions, synonyms and antonyms, images, etc.
- Rows can have completely different schemas

Amazon Dynamo DB adopts the key-value model

Key-value modeling

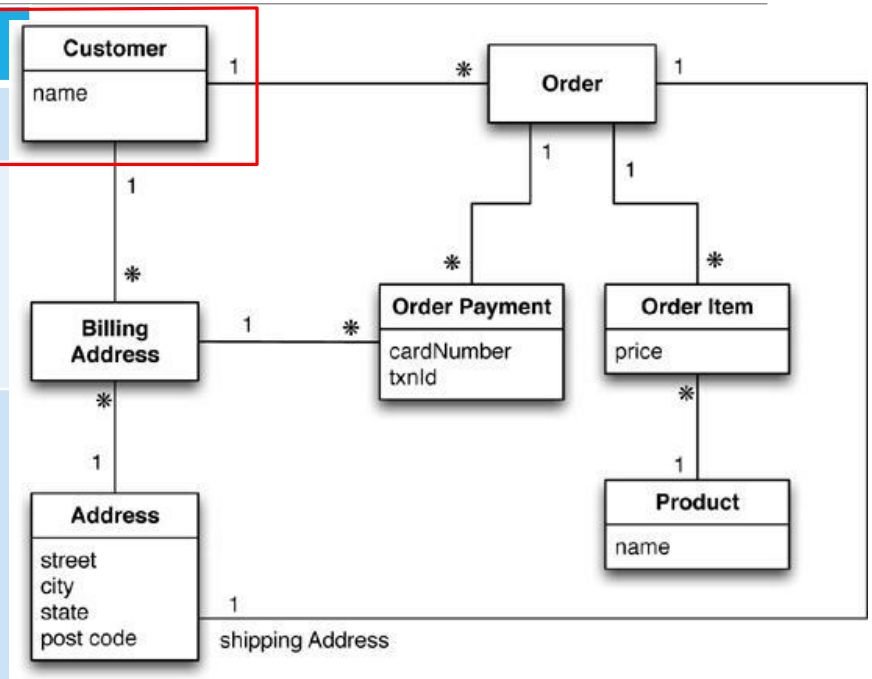
key	value
cust-1:name	Martin
cust-1:adrs	[{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}, {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}]
cust-1:order-99	{"orderpayments":[{"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}}, {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}], "products":[{"id":1, "name":"Cola", "price":12.4}, {"id":2, "name":"Fanta", "price":14.4}], "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001} }



key	value
p-1:name	Cola
p-2:name	Fanta

Key-value modeling

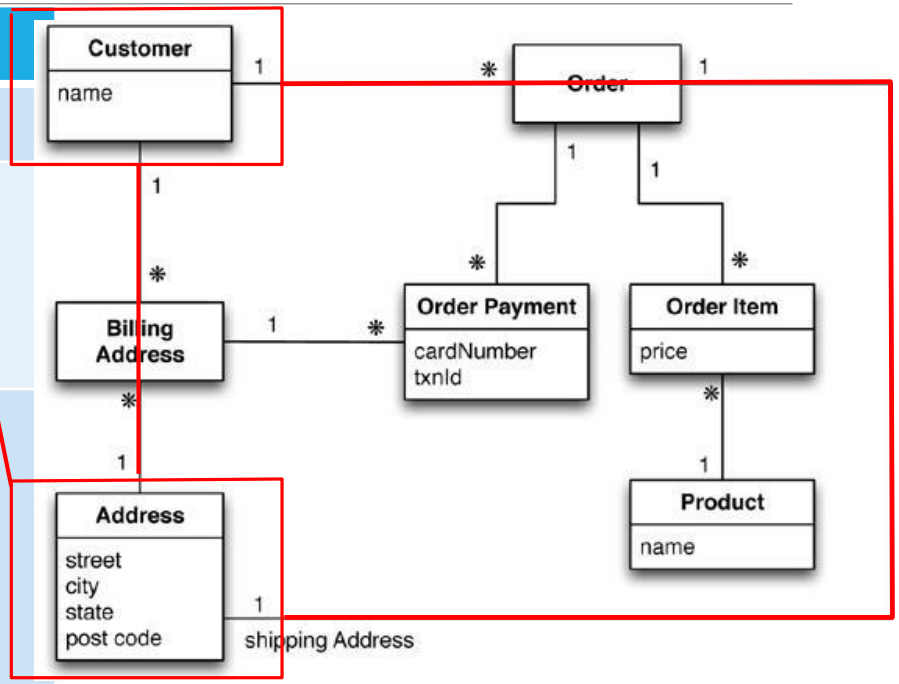
key	value
cust-1:name	Martin
cust-1:adrs	[{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}, {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}]
cust-1:order-99	{"orderpayments":[{"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}}, {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}], "products":[{"id":1, "name":"Cola", "price":12.4}, {"id":2, "name":"Fanta", "price":14.4}], "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001} }



key	value
p-1:name	Cola
p-2:name	Fanta

Key-value modeling

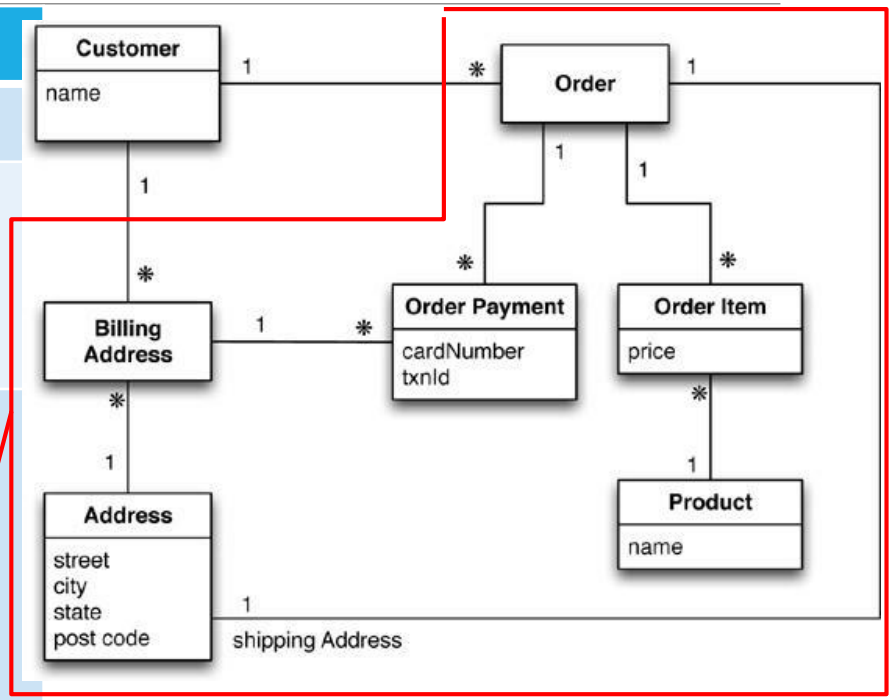
key	value
cust-1:name	Martin
cust-1:adrs	[{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}, {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}]
cust-1:order-99	{"orderpayments":[{"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}}, {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}], "products":[{"id":1, "name":"Cola", "price":12.4}, {"id":2, "name":"Fanta", "price":14.4}], "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001} }



key	value
p-1:name	Cola
p-2:name	Fanta

Key-value modeling

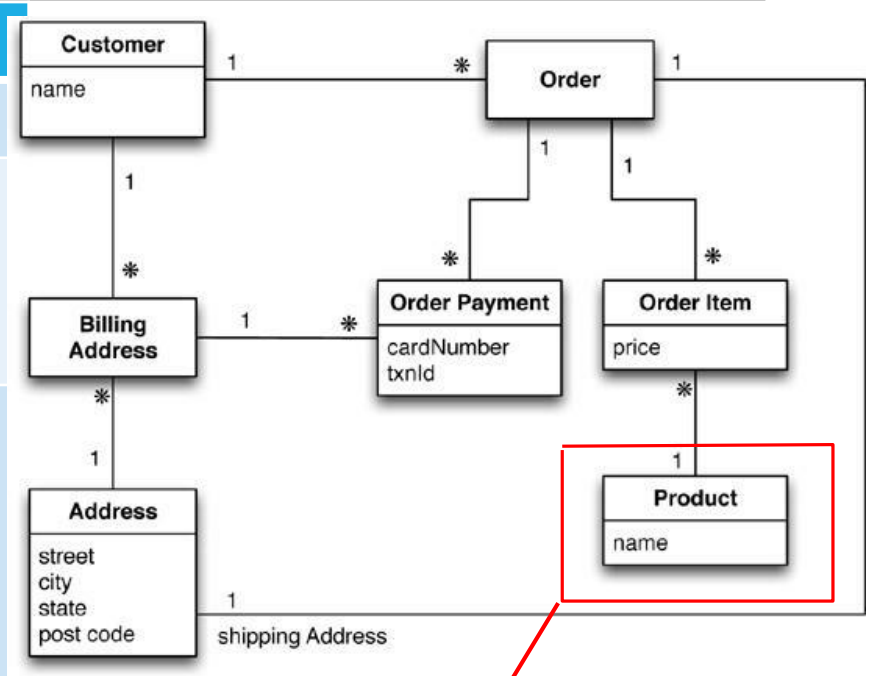
key	value
cust-1:name	Martin
cust-1:adrs	[{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}, {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}]
cust-1:order-99	{"orderpayments":[{"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}}, {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}], "products":[{"id":1, "name":"Cola", "price":12.4}, {"id":2, "name":"Fanta", "price":14.4}], "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001} }



key	value
p-1:name	Cola
p-2:name	Fanta

Key-value modeling

key	value
cust-1:name	Martin
cust-1:adrs	[{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}, {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}]
cust-1:order-99	{"orderpayments":[{"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}}, {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}], "products":[{"id":1, "name":"Cola", "price":12.4}, {"id":2, "name":"Fanta", "price":14.4}], "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001} }



key	value
p-1:name	Cola
p-2:name	Fanta

Key-value: querying

Three simple kinds of query:

- put($key as xs:string, $value as item())
  - Adds a key-value pair to the collection
  - If the key already exists, the value is replaced
- get($key as xs:string) as item()
  - Returns the value corresponding to the key (if it exists)
- delete($key as xs:string)
  - Deletes the key-value pair

The value is a *black box*: it cannot be queried!

- No "where" clauses
- No indexes on the values
- Schema information is often indicated in the key

Key	Value
user:1234:name	Enrico
user:1234:age	30
post:9876:written-by	user:1234
post:9876:title	NoSQL Databases
comment:5050:reply-to	post:9876

Document: data model

Each DB contains one or more collections (corresponding to tables)

Each collection contains a list of documents (usually JSON)

- Documents are hierarchically structured

Each document contains a set of fields

- The ID is mandatory

Each field corresponds to a key-value pair

- Key: unique string in the document
- Value: either simple (string, number, boolean) or complex (object, array, BLOB)
  - A complex field can contain other field

Atomicity level: the document

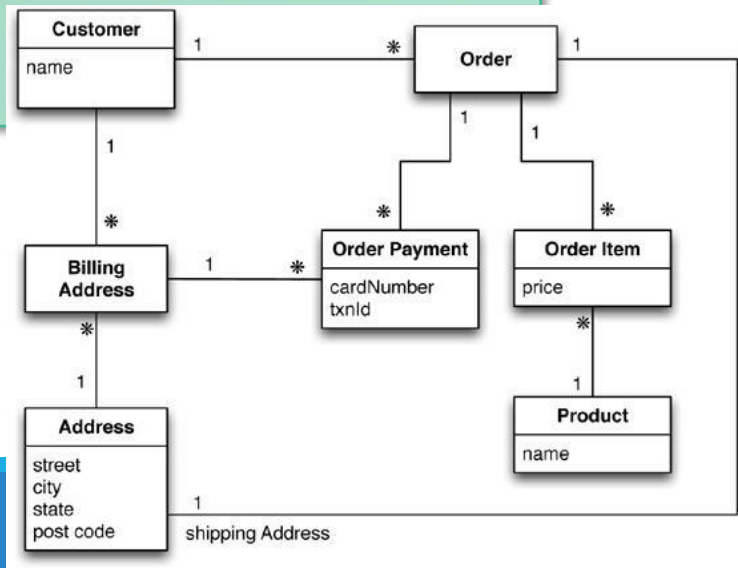
MongoDB adopts the document model

```
{
    "_id": 1234,
    "name": "Enrico",
    "age": 30,
    "address": {
        "city": "Ravenna",
        "postalCode": 48124
    },
    "contacts": [ {
        "type": "office",
        "contact": "0547-338835"
    }, {
        "type": "skype",
        "contact": "egallinucci"
    } ]
}
```

Document modeling 1

```
{   "_id": 1,
    "name": "Martin",
    "adrs": [ {"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007},
              {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
            ],
    "orders": [
               {"orderpayments":[ {"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}},
                                  {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}
                                ],
                 "products":[ {"id":1, "name":"Cola", "price":12.4},
                              {"id":2, "name":"Fanta", "price":14.4}
                            ],
                 "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
               }
             ]
}
```

```
[
        {    "_id":1,
             "name":"Cola",
             "price":12.4
        },
        {    "_id":2,
            "name":"Fanta",
            "price":14.4
        }
]
```



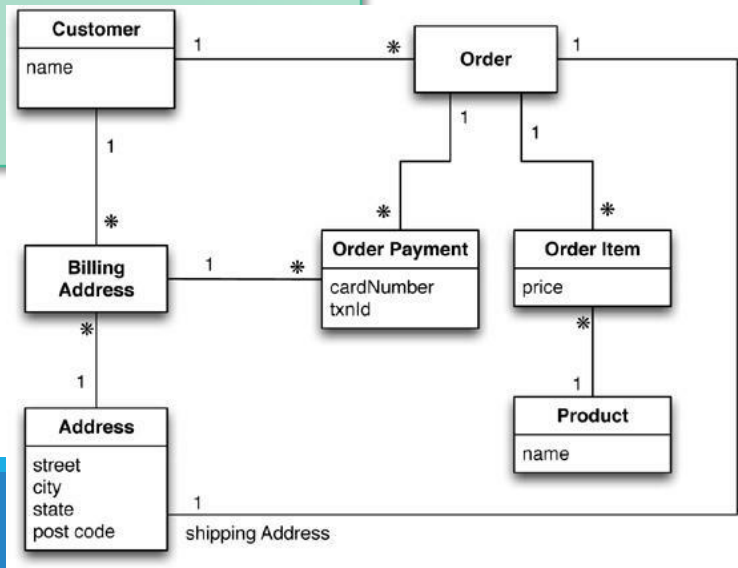
Document modeling 2

```
{   "_id": 1,
    "name": "Martin",
    "adrs": [ {"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007},
              {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
            ]
}
```

Requires a join to put together information about customers and products

```
{"_id": 1,
 "customer":1,
 "orderpayments":[ {"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}},
                   {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}
                 ],
 "products":[ {"id":1, "name":"Cola", "price":12.4},
              {"id":2, "name":"Fanta", "price":14.4}
            ],
 "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
}
```

```
{    "_id":1,
      "name":"Cola",
      "price":12.4
 },
 {"_id":2,
    "name":"Fanta",
          "price":14.4
 }
```



Document modeling 3

```
{"_id": 1,
 "customer":{"id":1, "name": "Martin","adrs": [ {"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007},
                                                {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
                                              ]
             }
,
 "orderpayments":[ {"card":477, "billadrs":{"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007}},
                   {"card":457, "billadrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}}
                  ],
"products":[ {"id":1, "name":"Cola", "price":12.4},
             {"id":2, "name":"Fanta", "price":14.4}
           ],
 "shipAdrs":{"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
}
```

```
{    "_id":1,
      "name":"Cola",
      "price":12.4
 },
 {"_id":2,
    "name":"Fanta",
           "price":14.4
 }
```

```
{   "_id": 1,
    "name": "Martin",
    "adrs": [ {"street":"Adam", "city":"Chicago", "state":"illinois", "code":60007},
              {"street":"9th", "city":"NewYork", "state":"NewYork", "code":10001}
            ]
}
```

Document: querying

Differently from the key-value, the value is *visible* by the DBMS

Thus, query languages are quite expressive

- Can create indexes on fields
- Can filter on the fields
- Can return more documents with one query
- Can select which fields to project
- Can update specific fields

Different implementations, different functionalities

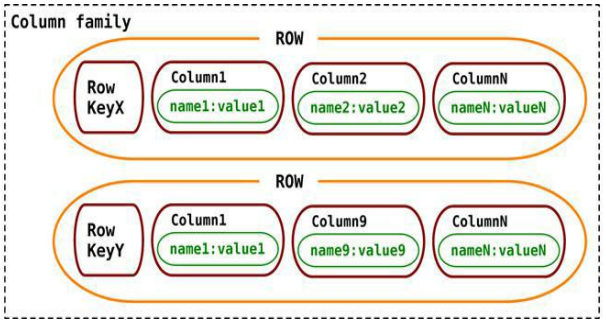
- Some enable (possibly materialized) views
- Some enable MapReduce queries
- Some provide connectors to Big Data tools (e.g., Spark, Hive)
- Some provide *full-text search* capabilities

Wide column: data model

Each DB contains one or more column families (corresponding to tables)

Each column family contains a list of row in the form of a key-value pair

- Key: unique string in the column family
- Value: a set of columns



Each column is a key-value pair itself

- Key: unique string in the row
- Value: simple or complex (*super-column*)

Atomicity level: the row

Sharding:

- based on a set of attributes
  - Shading on the rowkey it is not recomended since a selection access would impact on several shards
- it is applied on a row (of a column family)

With respect to the relational model:

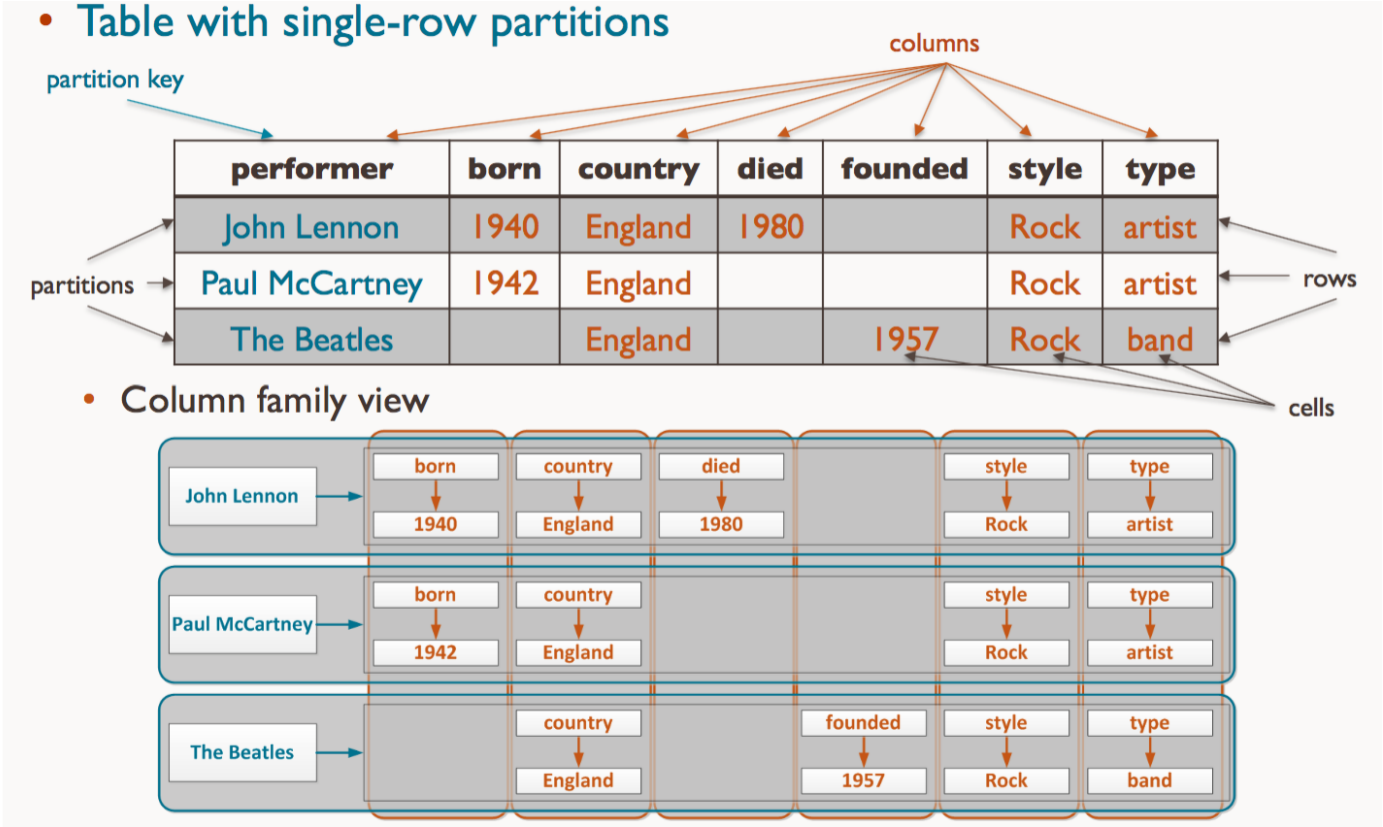
- Rows specify only the columns for which a value exists
  - Particularly suited for sparse matrixes
- Timestamps can be used to defines *versions* of column values

Google Big table and Hbase adopts the wide-column model

Wide column: data model

With respect to the relational model:

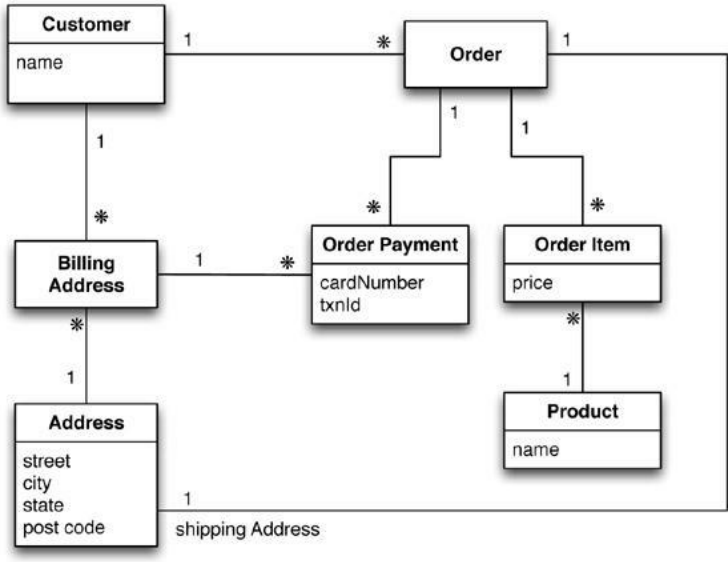
- Rows specify only the columns for which a value exists
  - Particularly suited for sparse matrixes
- Timestamps can be used to defines *versions* of column values



Wide-column modeling

Ord	CName	Pepsi	Cola	Fanta	…
1	Martin	-	12.4	14.4	-
2					

Ord	OrderPayments
1	Card | Steet | City | State | Code 477 | 9th | NewYork | NewYork | 10001 457 | Adam | Chicago | Illinois | 60007
2	…..



Wide column: querying

The query language expressiveness is in between key-value and document data models

- Column indexes are discouraged
- Can filter on column values (not always)
- Can return more rows with one query
- Can select which columns to project
- Can update specific columns (not always)

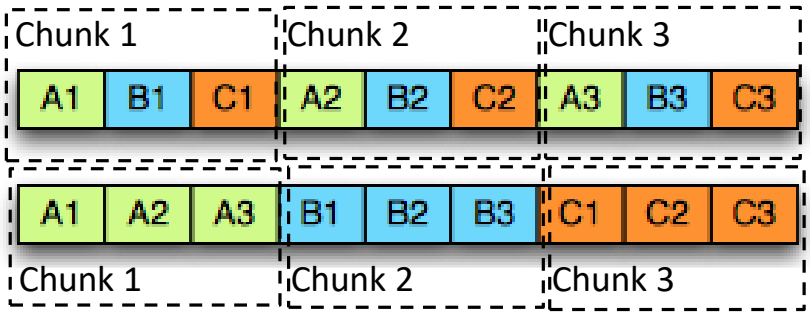
Given the similarity with the relational model, a SQL-like language is often used

Wide column: ≠ columnar

Do not mistake the wide column data model with the columnar storage where tables are stored in physical chunks as sequnces column values, rather than sequences of row values. Columnar DBs has fixed schemas. Columnar DBs are the best for applications that manage large amounts of data and require many reads and few updates (e.g. Data Warehouses).

Relational table

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3



Row-oriented

Column-oriented

Row-oriented

- Pros: inserting a record is easy
- Cons: several unnecessary data may be accessed when reading a record

Column-oriented

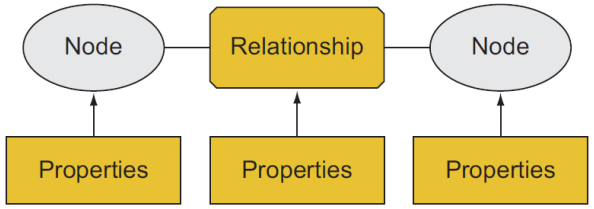
- Pros: only the required values are accessed
- Cons: writing a record requires multiple accesses

Graph: data model

Each DB contains one or more graphs

Each graph contains vertices and arcs

- Vertices: usually represent real-world entities
  - E.g.: people, organizations, web pages, workstations, cells, books, etc.
- Arcs: represent directed relationships between the vertices
  - E.g.: friendship, work relationship, hyperlink, ethernet links, copyright, etc.
- Vertices and arcs are described by properties



Atomicity level: the transaction

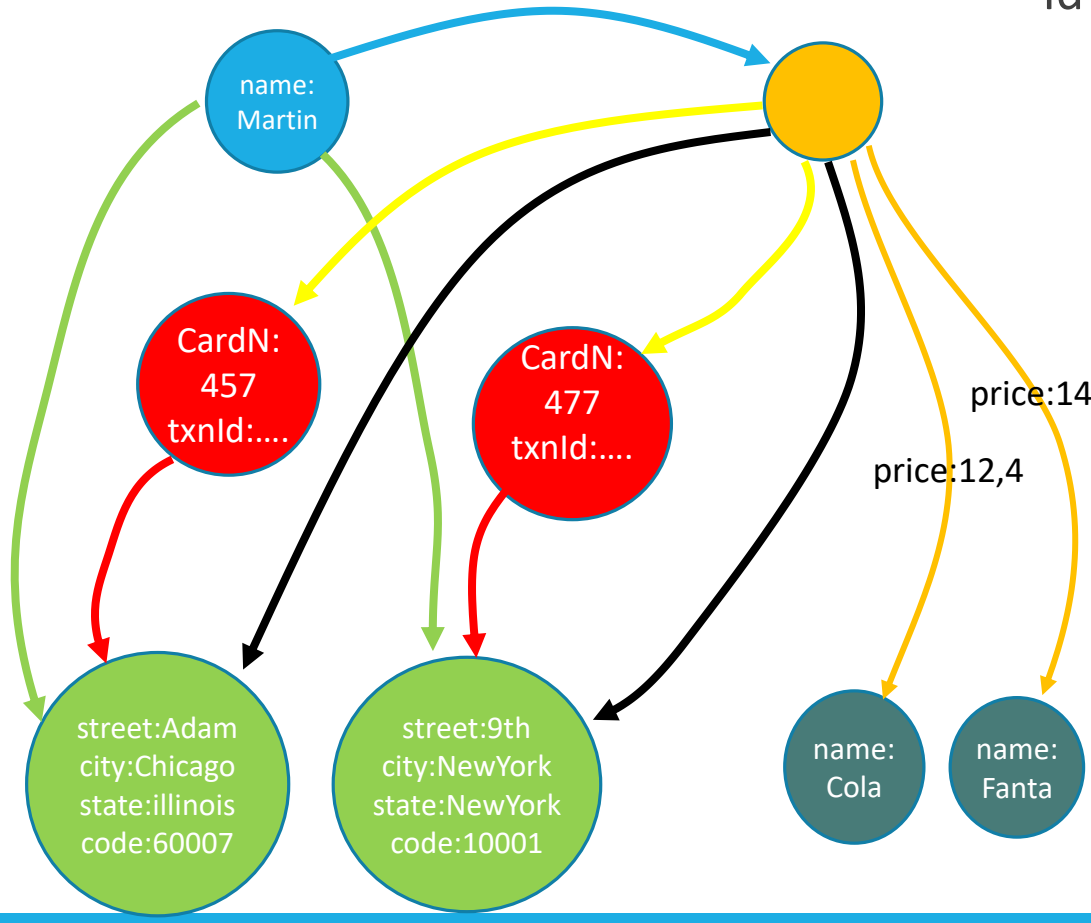
Most known specializations:

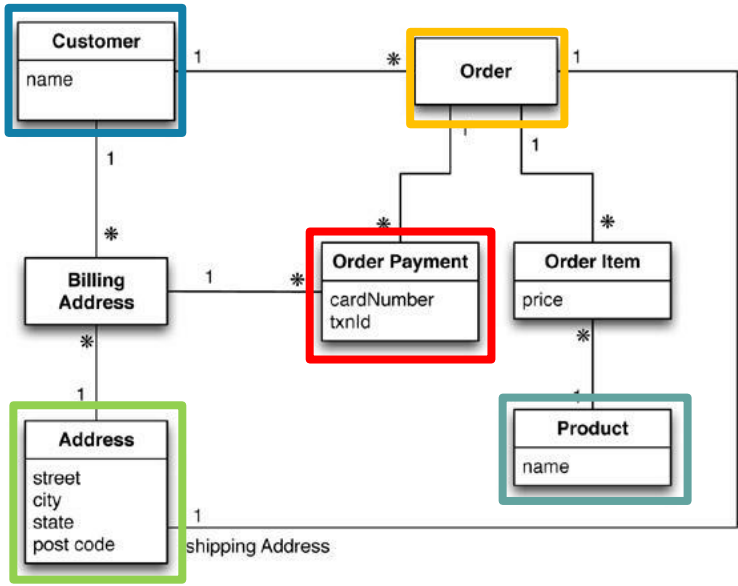
- Reticular data model
  - Parent-child or owner-member relationships
- Triplestore
  - Subject-predicate-object relationships (e.g., RDF)

Neo4J adopts the graph data model

Graph modeling

Id are implictly handled by the DBMS





Graph: querying

Graph databases usually model completely different contexts

Thus, query language and mechanism is quite different

- Support for transactions
- Support for indexes, selections and projections
- Query language is based on patterns

Query	Pattern
Find friends of friends	(user)-[:KNOWS]-(friend)-[:KNOWS]-(foaf)
Find shortest path from A to B	shortestPath((userA)-[:KNOWS*..5]-(userB))
What has been bought by those who bought my same products?	(us1)-[:PURCHASED]->(prod)<-[:PURCHASED]-(us2)-[:PURCHASED]->(otherProd)

In the Neo4J syntax:

- () identifies nodes (*name*) *name* is a label for a generic node
- [] identifies arcs
- - - / - -> identifies a relationships independently/dependently on its direction
- :TYPE constraints on a node/arc type

Graph vs Aggregate-oriented

The graph data model is intrinsically different from the others

- Focused on the relationships rather than on the entities per-se
- Limited scalability
- Data-driven modeling (more focus on atomic data like relational model)

Other data models (key-value, document and wide column) are defined aggregate-oriented

- The aggregate is the atomic block
- More encapsulation, less joins
- High scalability
- Query-driven modeling

Why scalability is limited in Graph DBMS?

Real graphs are strongly connected:

- **Six degrees of separation** is the **theory** that any person on the planet can be connected to any other person on the planet through a chain of acquaintances that has no more than five intermediaries

This makes very hard, or even impossible, to shard a graph on several machines without "*cutting*" several arcs (i.e. having several cross-machine links)

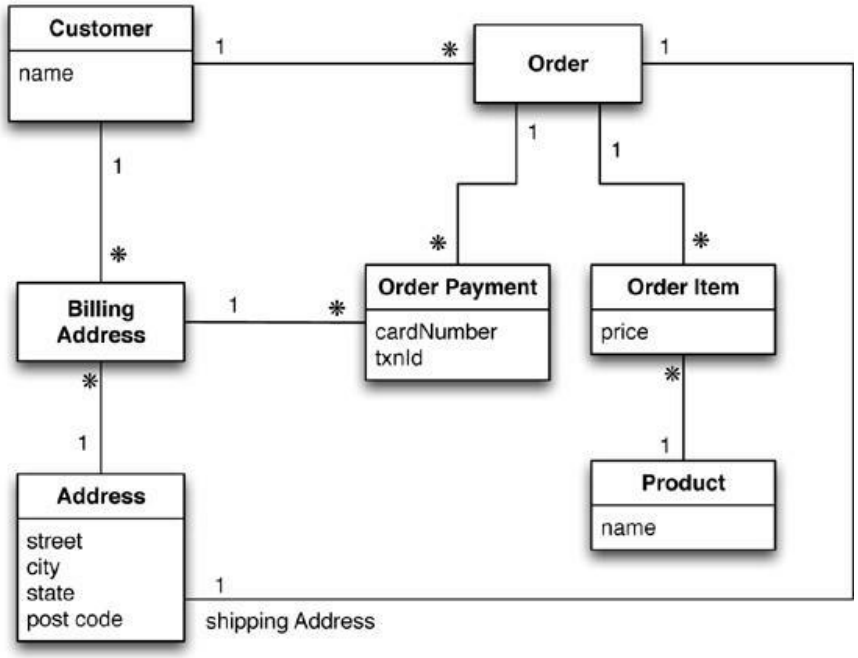
Applications will mostly probably query across machine boundaries due to data normalization and absence of redundancy that prevent joins in aggregate modeling

Different approaches can be used to limit the effects:

- Batching cross machine queries, so we only perform them at the close of each breadth first step.
- Limiting the depth of cross machine node searches (FlockDB is optimized for being very fast in returning one-hop relationships)

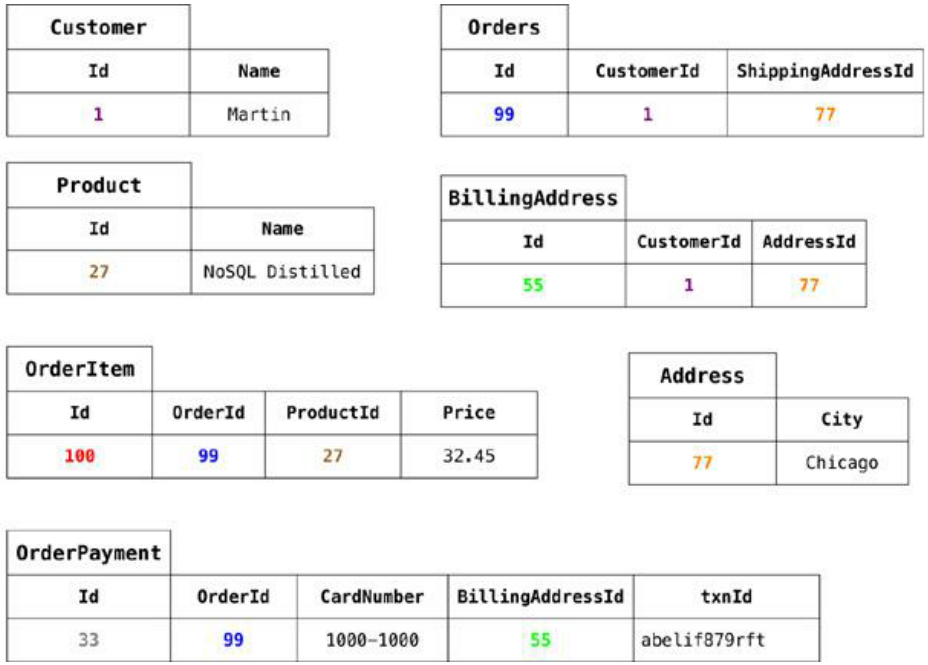
Aggregate modeling: an example

Typical use case: customers, orders and products



Aggregate modeling: an example

Relational modeling



Customer

Id	Name
1	Martin

Orders

Id	CustomerId	ShippingAddressId
99	1	77

Product

Id	Name
27	NoSQL Distilled

BillingAddress

Id	CustomerId	AddressId
55	1	77

OrderItem

Id	OrderId	ProductId	Price
100	99	27	32.45

Address

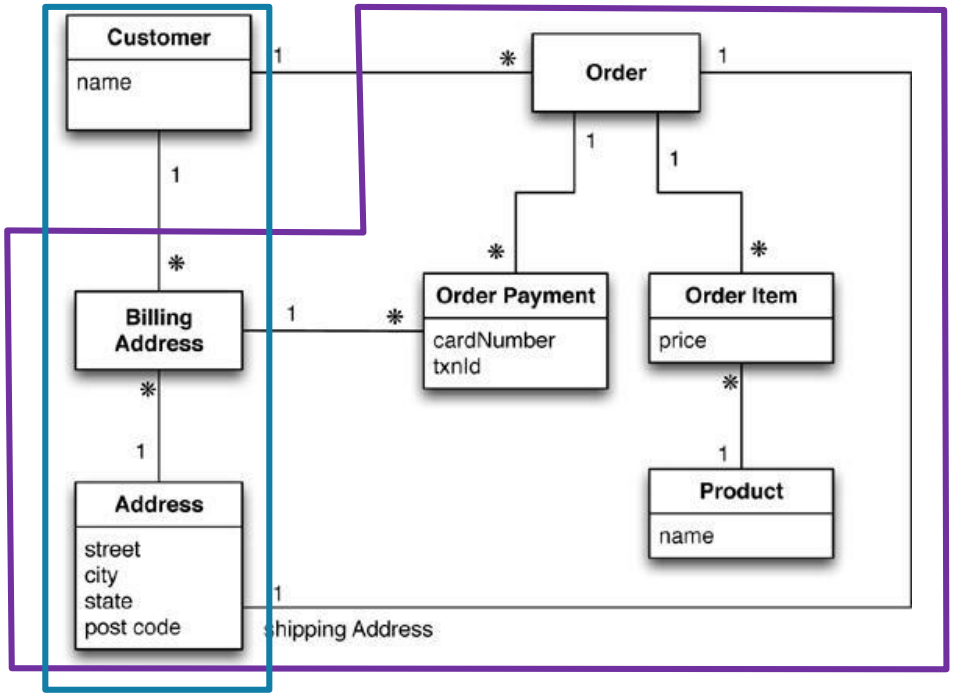
Id	City
77	Chicago

OrderPayment

Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Aggregate modeling: an example

One possible aggregate modeling



Aggregate modeling: an example

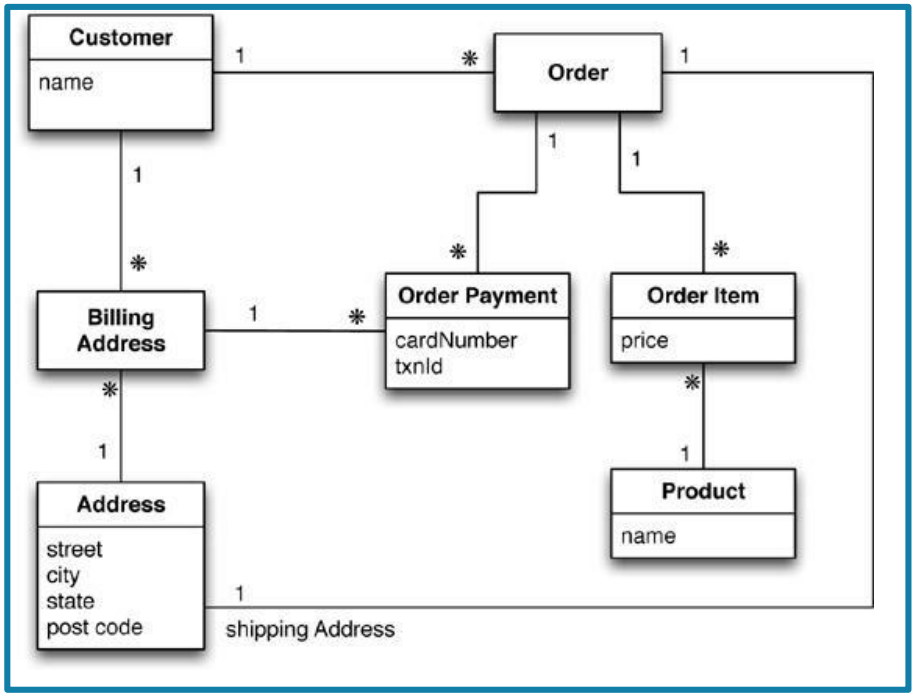
Sample instances (in a document database)

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}
```

```
// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

Aggregate modeling: an example

Another possible aggregate modeling



Aggregate modeling: an example

Sample instances (in a document database)

```
// in customers
{
"customer": {
"id": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"orders": [
  {
    "id":99,
    "customerId":1,
    "orderItems":[
    {
    "productId":27,
    "price": 32.45,
    "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
```

Aggregate modeling strategy

The *aggregate* term comes from Domain-Driven Design

- An aggregate is a group of tightly coupled objects to be handled as a block
- Aggregates are the basic unit for data manipulation and consistency management

Advantages

- Can be distributed trivially
  - Data the should be used together (e.g., orders and details) are stored together
- Facilitate the developer's job
  - By surpassing the impedance mismatch problem

Disadvantages

- **No design strategy exists for aggregates**
  - It only depends on how they are meant to be used
- Can optimize only a limited set of queries

RDBMS are agnostic from this point of view

One size does not fit all

TO EACH APPLICATION ITS OWN DATA MODEL

Key-Value: popular DBs

Riak: http://basho.com/riak/

Redis (Data Structure server): http://redis.io/

- Supports complex fields (liste, set, …) and operations on values (range, diff, …)

Memcached DB: http://memcached.org/

Berkeley DB: http://www.oracle.com/us/products/database/berkeley-db/

HamsterDB: http://hamsterdb.com/

- Especially used for embedded contexts

Amazon DynamoDB: https://aws.amazon.com/dynamodb/

- Commercial solution

Project Voldemort: http://www.project-voldemort.com/

- Open-source version of Amazon DynamoDB

Key-Value: when to use

Store session informations

- Each web session is identified by its own `sessionId`. All related data can be stored with a `PUT` request and returned with a `GET` request.

User profiles, preferences

- Each user is uniquely identified (`userId`, username) and has her own preferences in terms of language, colors, time zone, products, etc. – data that fits well within an aggregate

Shopping cart, chat services

- Each e-commerce websites associates a shopping cart to a user; it can be stored as an aggregate identified by the user ID.

Key-Value: real use cases

Crawling of web pages

- The URL is the key, the whole page content (HTML, CSS, JS, images, ..) is the value

Key	Value
http://www.example.com/index.html	<html>…
http://www.example.com/about.html	<html>…
http://www.example.com/products.html	<html>…
http://www.example.com/logo.png	Binary…

Twitter timeline

- The user ID is the key, the list of most recent tweets to be shown is the value

Amazon S3 (Simple Storage Service)

- A cloud-based file system service
- Useful for personal backups, file sharing, website or apps publication
- The more you store, the more you pay
  - Storage: approx. $0.03 per GB per month
  - Uploading files: approx. $0.005 per 1000 items
  - Downloading files: approx. $0.004 per 10,000 files* PLUS $0.09 per GB (first GB free)

Key-Value: when to avoid

Data with many relationships

- When relationships between data (in the same or in different collections) must be followed – even though some systems offer link-walking mechanisms.

Multi-operation transactions

- If it is necessary to ensure the atomicity of operations involving more keys

Querying the data

- If it is necessary to query the values, not just the key
  Few systems offer limited functionalities (e.g., Riak Search)

Multi-key operations

- Because operations involve only one key at a time

Document: popular DBs

MongoDB: http://www.mongodb.org

CouchDB: http://couchdb.apache.org

Terrastore: https://code.google.com/p/terrastore

OrientDB: http://www.orientechnologies.com

RavenDB: http://ravendb.net

Lotus Notes: http://www-03.ibm.com/software/products

Document: when to use

Event logs

- Often used as central repositories to store event logs from several applications
- It is a good practice to shard on the app name or on the type of event

CMS, blogging platforms

- The absence of a predefined schema fits well within content management systems (CMS) or website management applications, to handle comments, registrations and user profiles

Web Analytics or Real-Time Analytics

- The ability to update only specific fields enables fast update of analytical metrics
- Text indexing enables real-time sentiment analysis and social media monitoring

E-commerce applications

- Schema flexibility is often required to store products and orders, as well as to enable schema evolution without incurring into refactoring or migration costs

Document: real use cases

Adversting services

- MongoDB was born as a system for banner ads
  - 24/7 availability and high performance
  - Complex rules to find the right banner based on user's interests
  - Handle several kinds of ads and show detailed analytics

Internet of Things

- Real-time management of sensor-based data
- Bosch uses MongoDB to capture data from cars (breaks, ABS, windscreen wiper, etc.) and aircrafts maintenance tools
  - Business rules are applied to warn the pilot when the breaking system pressure falls under a critical threshold, or the maintenance operator when the tool is used improperly
- Technogym uses MongoDB to capture data from gym equipment

Document: when to avoid

Multi-operation transaction

- If not for a few exceptions (e.g., RavenDB), document databases are not suited for cross-document atomicity

Queries on high-variety data

- If the aggregate structure continuously evolves, queries must be constantly updated (and normalization clashes with the concept of aggregate)

Wide column: popular DBs

Cassandra: http://cassandra.apache.org

HBase: https://hbase.apache.org

Hypertable: http://hypertable.org

SimpleDB: https://aws.amazon.com/simpledb

Google BigTable: https://cloud.google.com/bigtable

Wide column: when to use

Event logs; CMS, blogging platforms

- Similarly to document databases, different applications may use different columns

Sparse matrixed

- While an RDBMS would store *null* values, a wide column stores only the columns for which a value is specified

GIS applications

- Pieces of a map (tiles) can be stored as couples of latitude and longitude

Counters

- Analytical metrics (e.g., accesses to web pages) can be stored as special increment-only fields

Time To Live

- Data can be associated to a Time To Live (TTL); e.g., to provide temporary permissions, or to enable ads for a limited period of time

Wide column: real use cases 

Google applications

- BigTable is the DB used by Google for most of its applications, including Search, Analytics, Maps and Gmail

User profiles and preferences

- Spotify uses Cassandra to store metadata about users, artists, songs, playlists, etc.

Manhattan

- After using Cassandra, Twitter ha developed its own proprietary NoSQL system to support most of its services

Wide column: when to avoid

Systems requiring ACID transactions

Aggregation of data (e.g., sum, average) **is not supported**, it must be handled by the application

- However, analytical tools (e.g., Apache Hive, Spark) can connect to the database and enable these kind of queries

Graph: popular DBs

Neo4J: http://neo4j.com

InfiniteGraph: http://www.objectivity.com/products/infinitegraph/

OrientDB: http://orientdb.com/orientdb/

FlockDB: http://github.com/twitter-archive/flockdb

Graph: when to use

Interlinked data

- Social networks are one of the most typical use case of graph databases (e.g., to store friendships or work relationships); every relationship-centric domain is a good one

Routing and location-based services

- Applications working on the TSP (Travelling Salesman Problem) problem
- Location-based application that, for instance, recommend the best restaurant nearby; in this case, relationships model the distance between node

Recommendation applications, fraud-detection

- Systems recommending «the products bought by your friends», or «the products bought by those who bought your same products»
- When relationships model behaviors, outlier detection may be useful to identify frauds

Graph: real use cases

Relationships analysis

- Finding common friends (e.g., friend-of-a-friend) in a social network
- Identifying clusters of phone calls that identify a criminal network
- Analyzing flows of money to identifying money recycling patterns or credit card theft
- Main users: law firms, police, intelligence agencies
  - https://neo4j.com/use-cases/fraud-detection/
- Useful for text analysis as well (Natural Language Processing)

Inference

- Creating rules that define new knowledge based on existing patterns (e.g., transitive relationships, trust mechanisms)

Graph: when to avoid

Data-intensive applications

- Traversing the graph is trivial, but analyzing the whole graph ca be expensive
- There exist framework for distributed graph analysis (e.g., Apache Giraph), but they do not rely on a graph DB

Polyglot persistence

Different databases are designed to solve different problems

Using a single DBMS to handle everything …

- Operational data
- Temporary session information
- Graph traversing
- OLAP analyses
- …

… usually lead to inefficient solutions

Each activity has its own requirements (availability, consistency, fault tolerance, etc.)

Choosing the right database

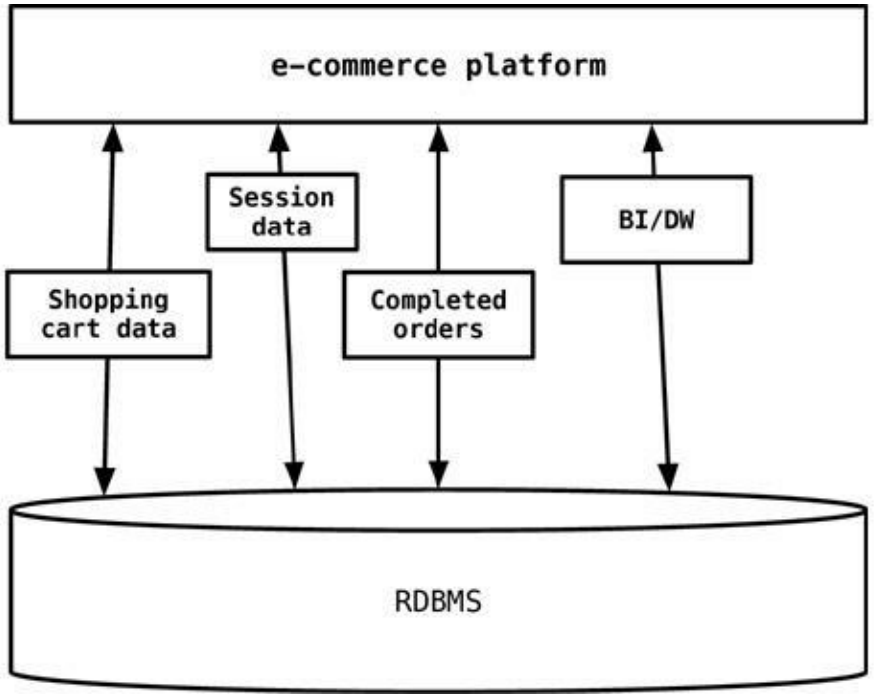
Why choose NoSQL over an RDBMS?

- Scale-out!
- Requiring features that only NoSQL systems provide (e.g., graph, nested structure, schemaless feature)
- Increase performance when immediate consistency is not a must
- Improve developers' productivity by avoiding the impedance mismatch problem
  - Although many other issues are dumped on the application…

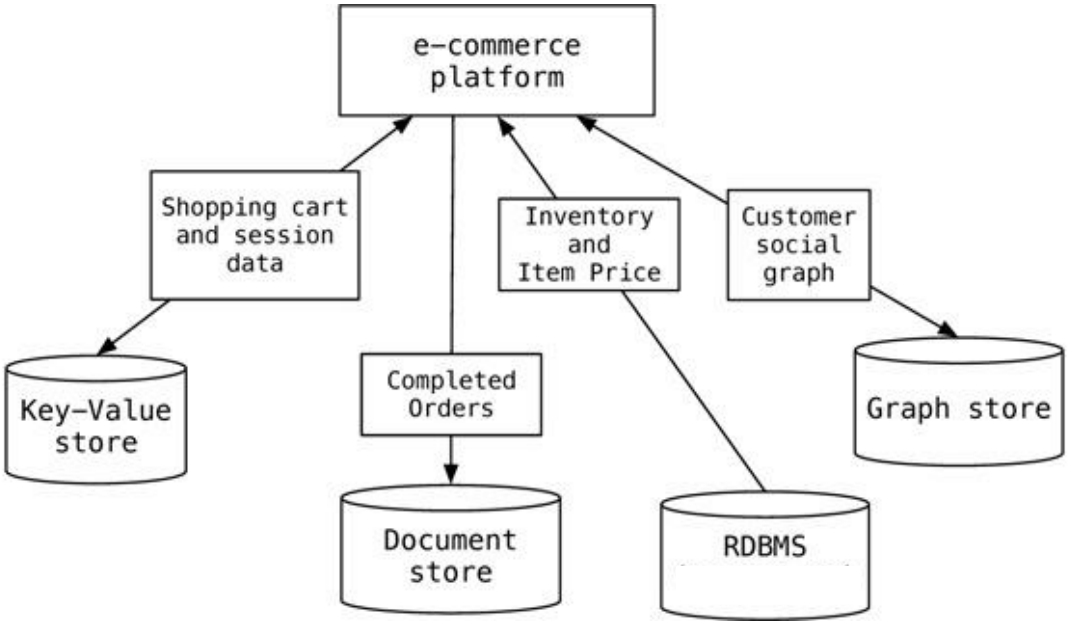
The NoSQL world is not as mature as the relational one

- But it is evolving
- It is essential to verify the expectations and the expected improvements before choosing the right technology

Traditional approach



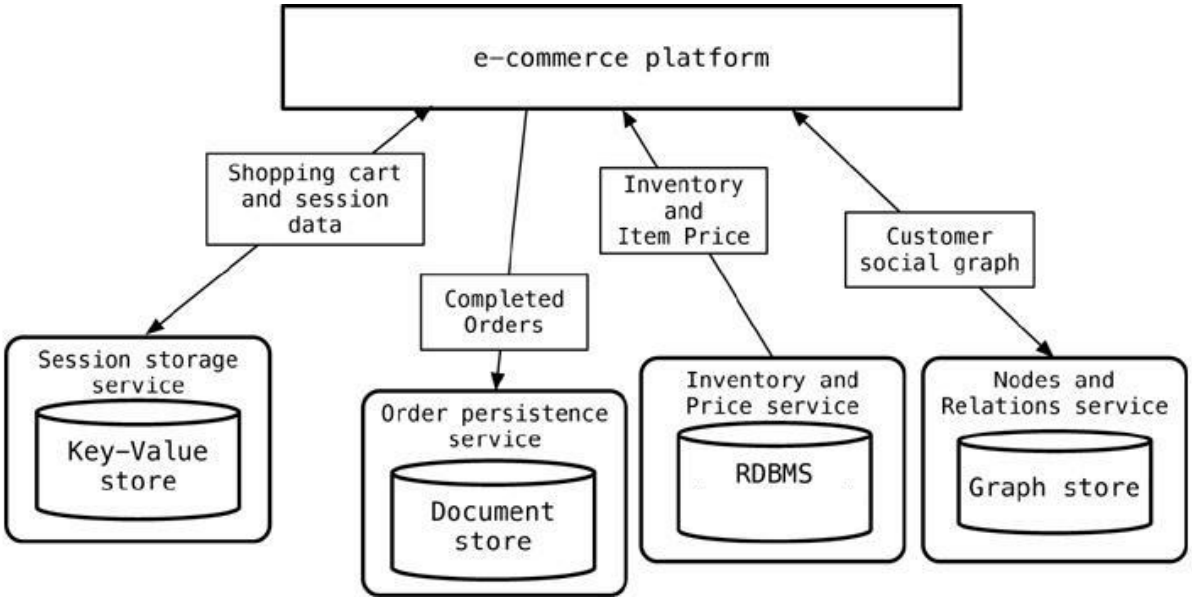
Polyglot data management



Service-oriented polyglot data management

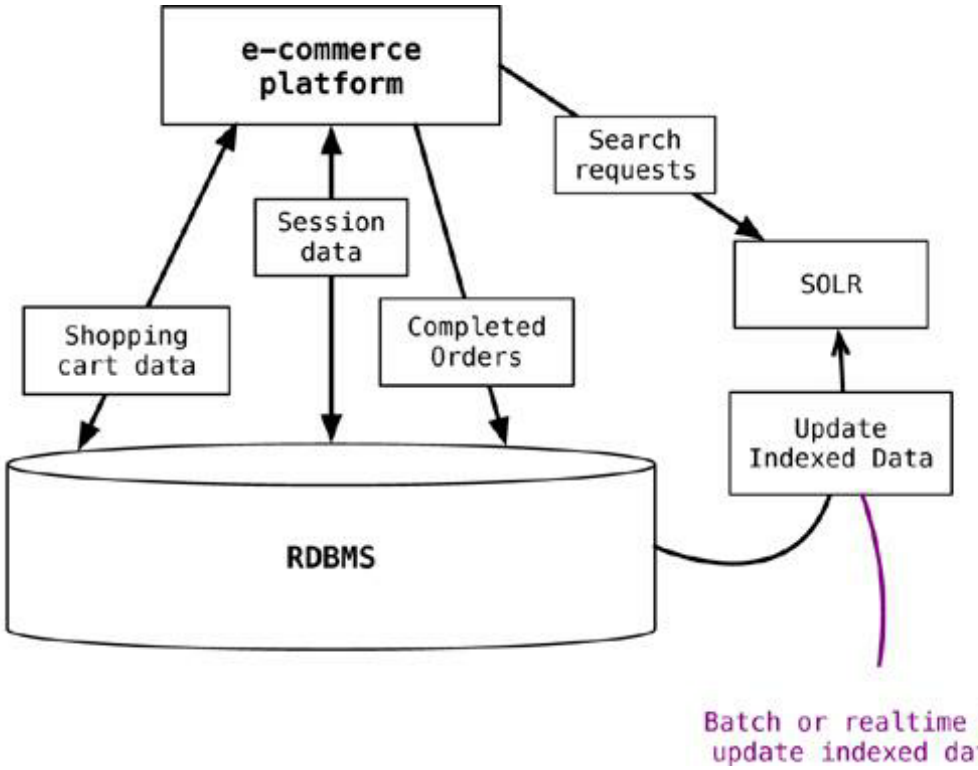
Each DB should be "embedded" within services, which offer API services to enable data access and manipulation

- Several NoSQL systems (e.g., Riak, Neo4J) already provide REST APIs



Supporting existing technologies

If the current solution cannot be changed, NoSQL systems can still support the existing ones



Polyglot persistence in the business

DBAs are becoming more and more polyglot

- Important to understand how these systems work and how they can be exploited

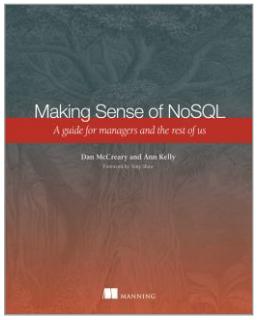
Beware about security

- Many NoSQL systems are still young and **do not** offer strong security mechanisms
- Responsibility is dumped on the applications

What about licences, support, upgrade, drivers, auditing, support systems

- Most systems are open-source and have an active community; some companies offer enterprise support
- Support tool are being released (e.g., MongoDB Monitoring Service, Datastax Ops Center, Rekon browser)
- ETL tools are adding support for NoSQL systems

Suggested reading and resources



D. McReary, A. Kelly
Making Sense of NoSQL
(Manning)

P.J. Sadalage, M. Fowler
NoSQL Distilled
(Pearson Education)