NoSQL Databases

About me

Matteo Francia, PhD

- Adjunct Professor, will teach *Big Data and Cloud Platforms* next year
- Member of the Business Intelligence Group
- Main research interests:
  - big data
  - data modeling
  - precision agriculture

Acknowledgements

- Thanks to Enrico Gallinucci for this teaching material

About today

A quick introduction on NoSQL databases

- Origins
- Characteristics
- Data models
- Issues
- Usage
- Demo

Let's start from the beginning

RDBMSs are full of strength

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

Robustness

- Have been used for over 40 years

RDBMSs have weaknesses as well

Impedance mismatch

- Data are stored according to the relational model, but applications to modify them typically rely on the object-oriented model
- 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 neither easy nor cheap (e.g., Oracle RAC)

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

What is "NoSQL"

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

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, 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, characterized by strong simplicity for data storage and manipulation

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

NoSQL misconceptions

Not a farewell to SQL

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

Not necessarily open-source

- There exist both open-source and commercial systems

Not only Cloud Computing

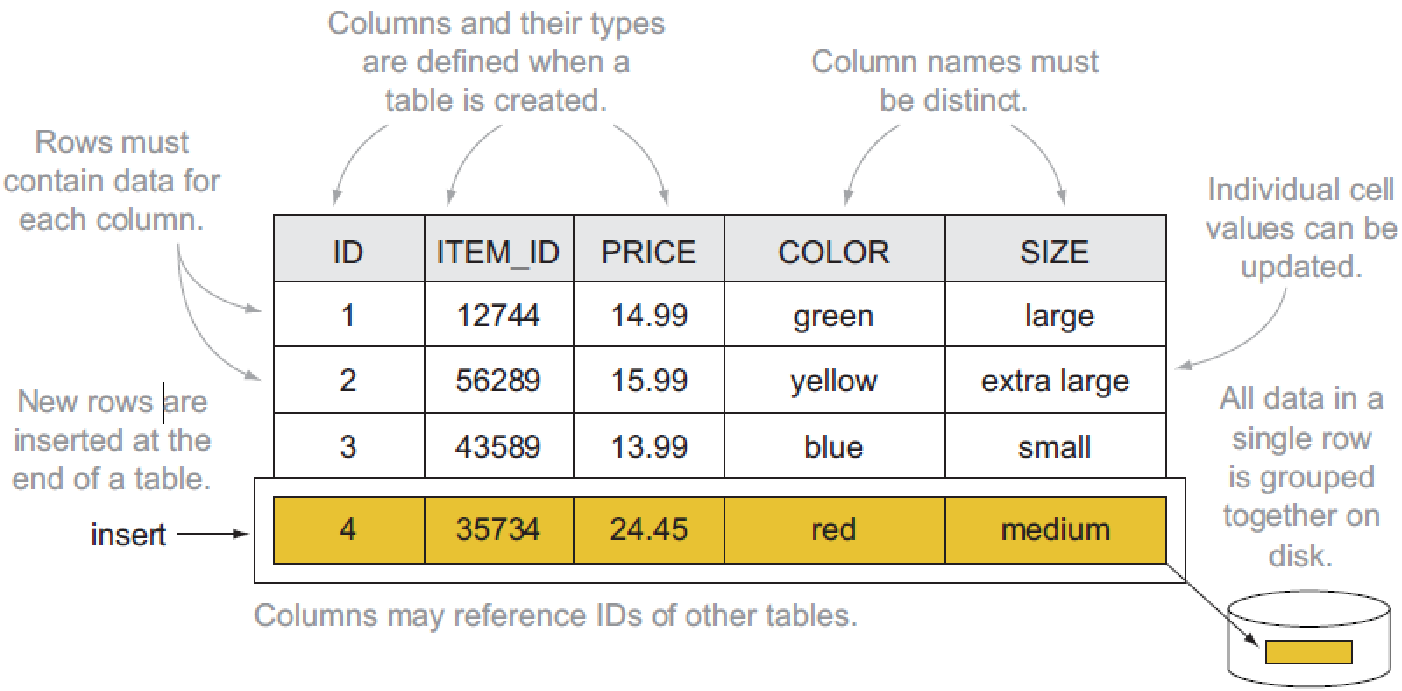
- There exist both on-premise and cloud solutions

Not an optimization of hardware resources

- With the same resources, a centralized RDBMS is probably better performing

Data modeling

Relational model



ID	ITEM_ID	PRICE	COLOR	SIZE
1	12744	14.99	green	large
2	56289	15.99	yellow	extra large
3	43589	13.99	blue	small
4	35734	24.45	red	medium

NoSQL: several data models

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

Model	Description	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

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

Atomicity level: the key-value pair

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>

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.

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

Key	Value
user:1234:name	Matteo
user:1234:age	33
post:9876:written-by	user:1234
post:9876:title	NoSQL Databases
comment:5050:reply-to	post:9876

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

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: unque 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

```
{
    "_id": 1234,
    "name": "Matteo",
    "age": 33,
    "address": {
        "city": "Ravenna",
        "postalCode": 48124
    },
    "contacts": [ {
        "type": "office",
        "contact": "0547-338835"
    }, {
        "type": "skype",
        "contact": "mfrancia"
    } ]
}
```

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

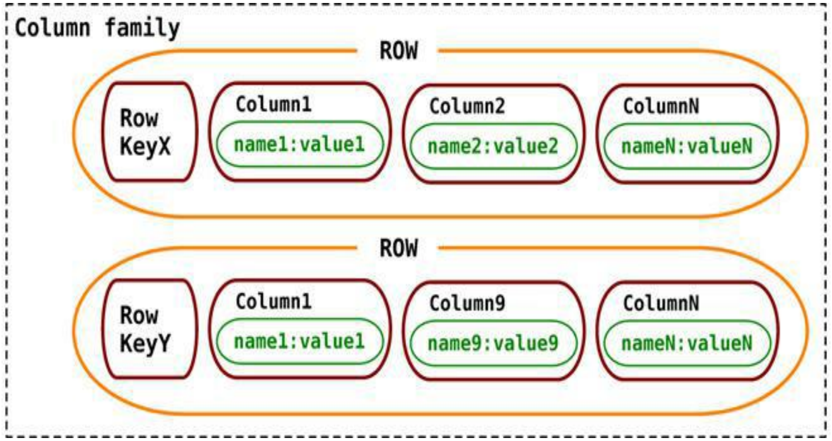
- Some enable (possibly materialized) views
- Some enable MapReduce queries
- Some provide connectors to Big Data tools (e.g., Spark, Hive)
- Some provide *full-text search* capabilities

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 (*supercolumn*)

Atomicity level: the row

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

Column family	Row key	Column key	Timestamp	Value

Wide column: querying

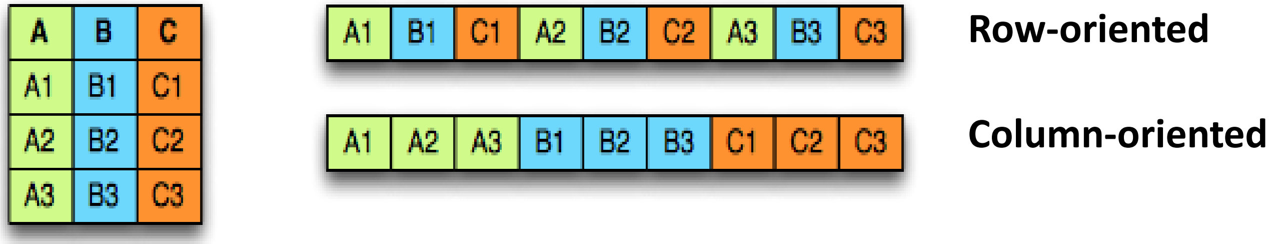
The query language expressiveness is in between key-value and document data models

- 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 used for OLAP applications



Row-oriented

- Pro: inserting a record is easy
- Con: several unnecessary data may be accessed when reading a record

Column-oriented

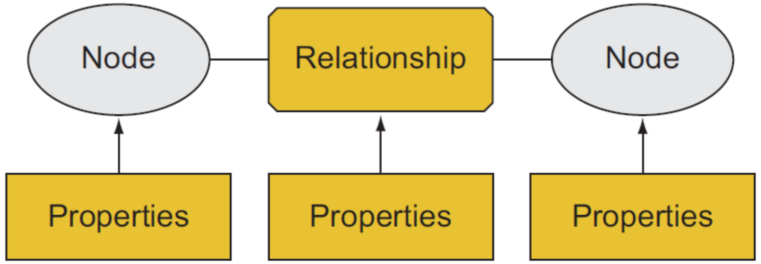
- Pro: only the required values are accessed
- Con: writing a record requires multiple accesses

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
- Arcs are stored as physical pointers



Atomicity level: the transaction

Most known specializations:

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

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 based on detecting 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?	(user)-[:PURCHASED]->(product)<-[:PURCHASED]-()-[:PURCHASED]->(otherProduct)

Aggregate vs Graph modeling

Key-value, document and wide column are called aggregate-oriented

- Aggregate = key-value pair, document, row (respectively)
- The aggregate is the atomic block (no guarantees for multi-aggregate operations)

Based on the concept of encapsulation

- Pro: avoid joins as much as possible → achieve **high scalability**
- Con: data denormalization → **potential inconsistencies in the data**
- Query-driven modeling

The graph data model is intrinsically different from the others

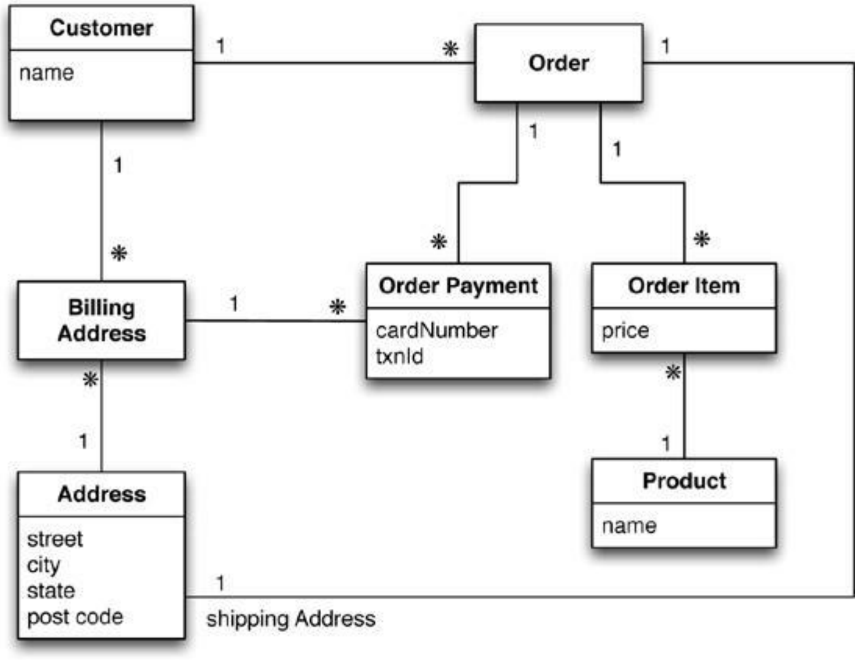
- Focused on the relationships rather than on the entities per-se
- **Limited scalability**: it is often impossible to shard a graph on several machines without "cutting" several arcs (i.e. having several cross-machine links)
  - Batch cross-machine queries: don’t follow relationships one by one, but "group them" to make less requests
  - Limit the depth of cross-machine node searches
- Data-driven modeling

Data modeling

Let's see some examples

Data modeling

Typical use case: customers, orders and products



Data modeling: relational

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

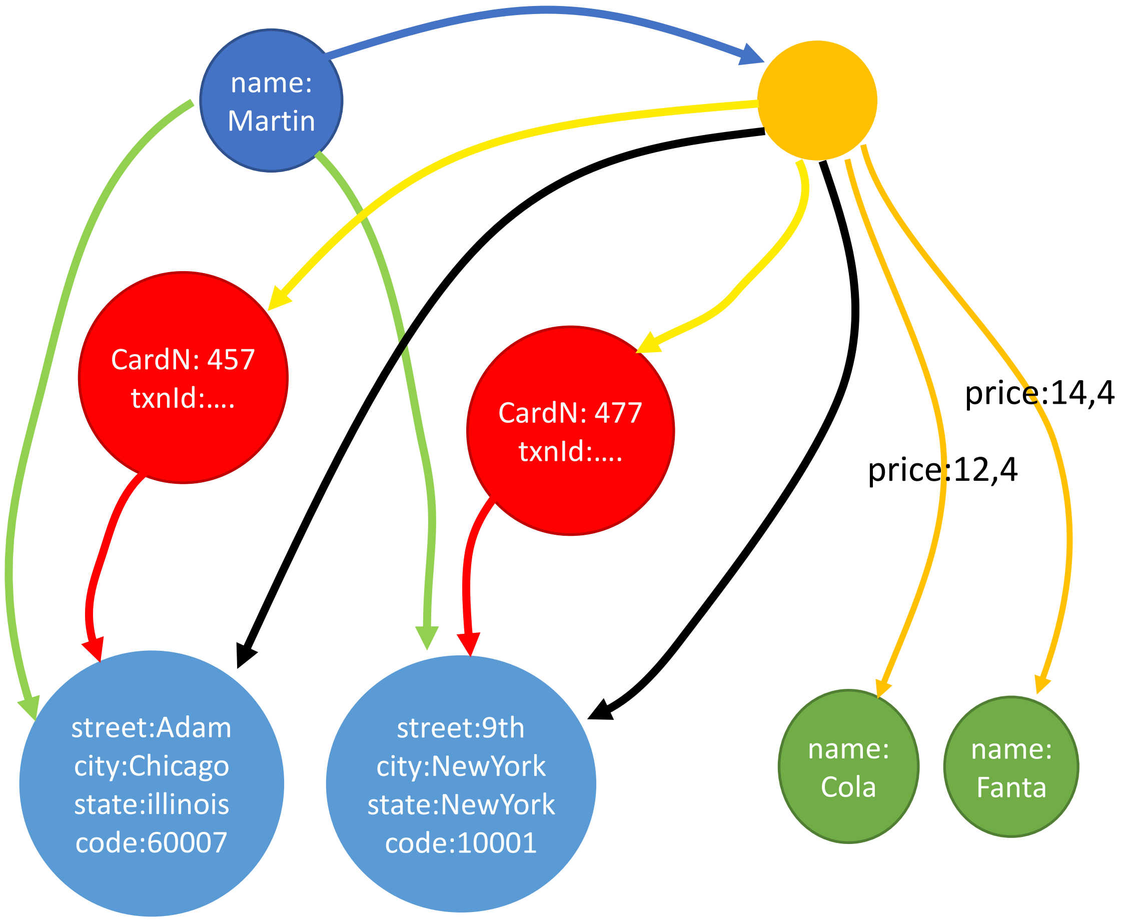
BillingAddress		
Id	CustomerId	AddressId
55	1	77

OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

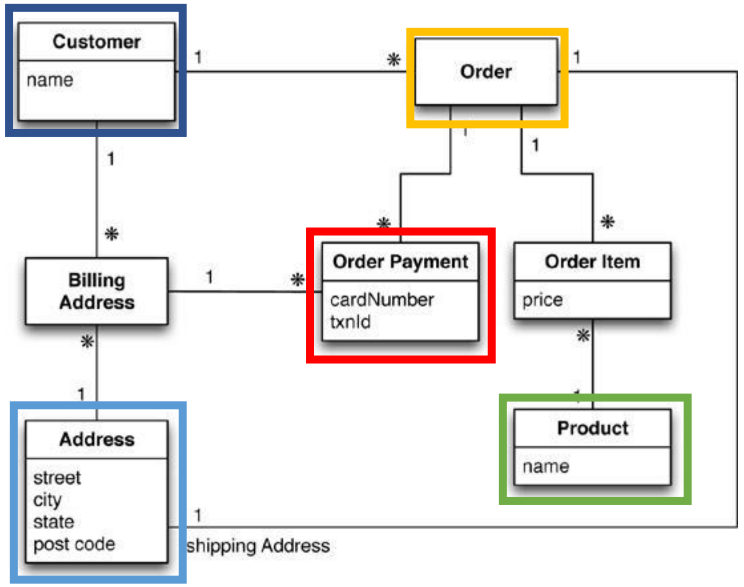
Address	
Id	City
77	Chicago

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

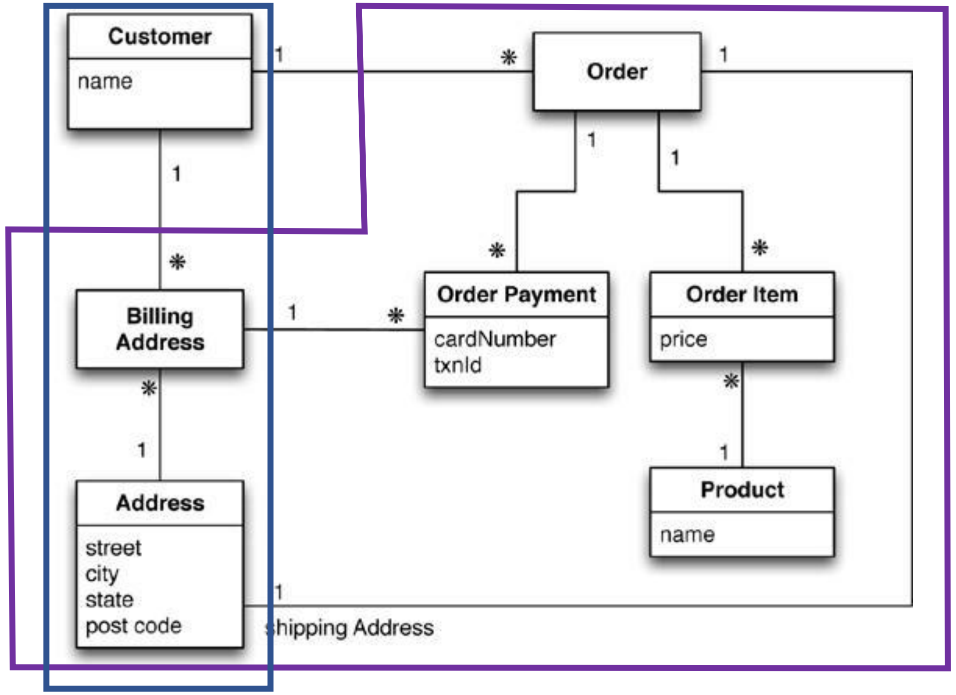
Data modeling: graph



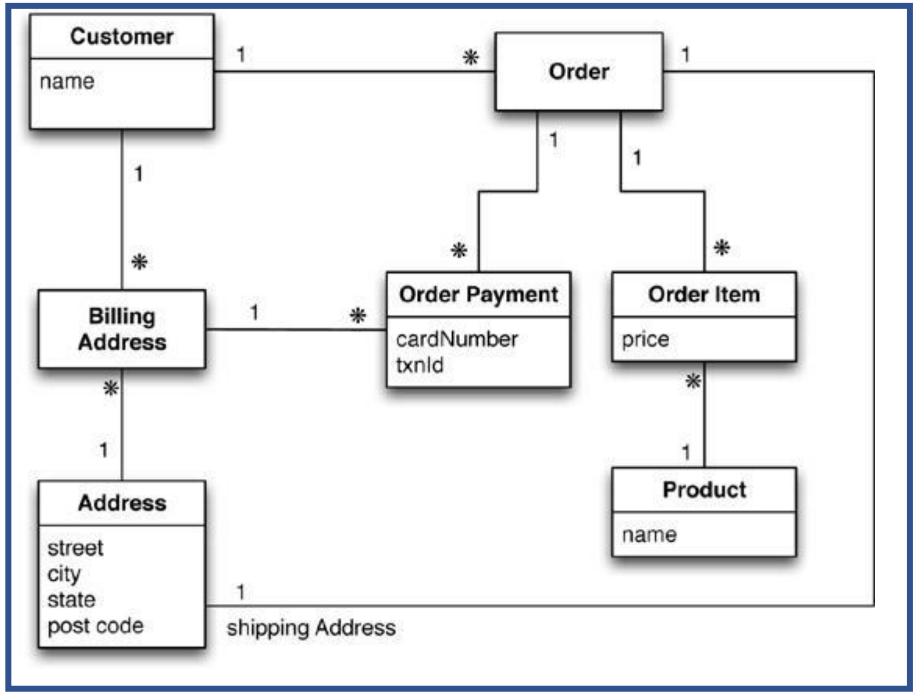
- IDs are implicit
- Different edge colors imply different edge types



Data modeling: aggregate-oriented



Data modeling: aggregate-oriented



Data modeling: key-value

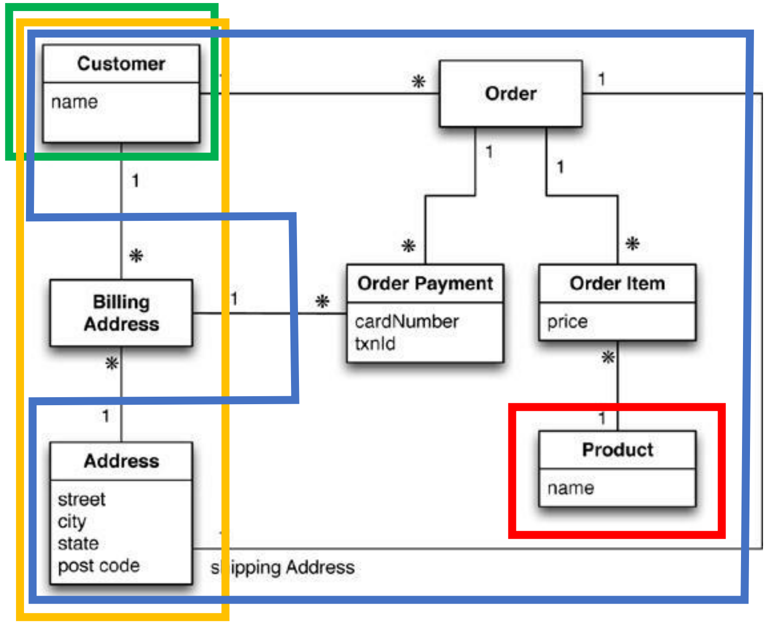
Customer collection

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:ord-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}
	}

Product collection

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



Data modeling: document-based

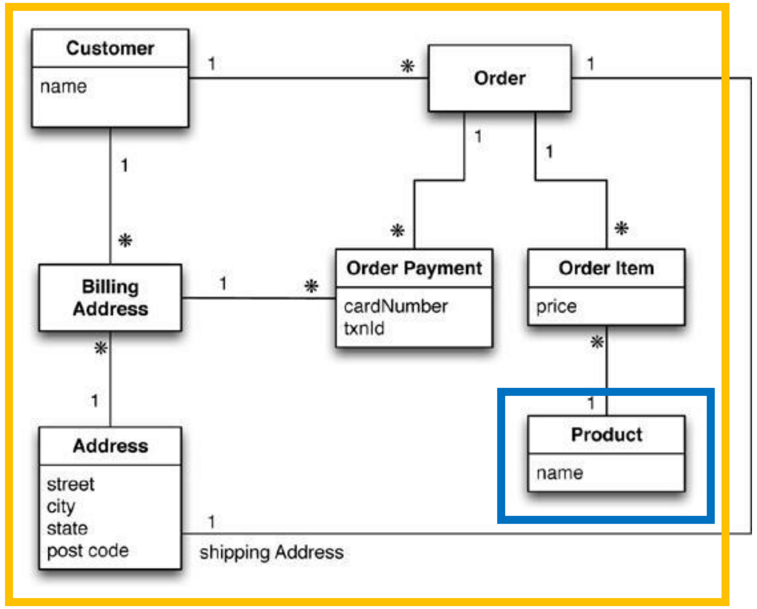
Customer collection

```
{
  "_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}
  }]
}
```

Product collection

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

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



Data modeling: document-based

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

Customer collection

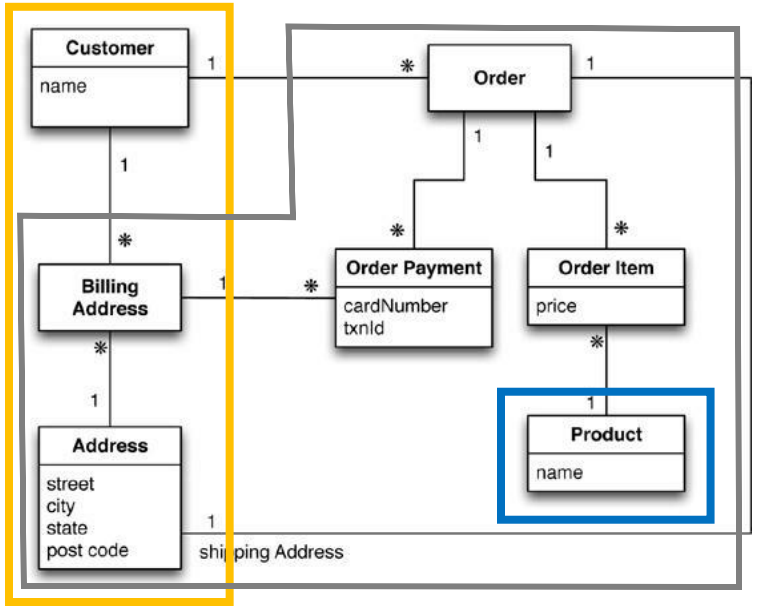
```
{
  "_id":1,
  "name":"Cola",
  "price":12.4
}
```

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

Product collection

```
{
  "_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}
}
```

Order collection



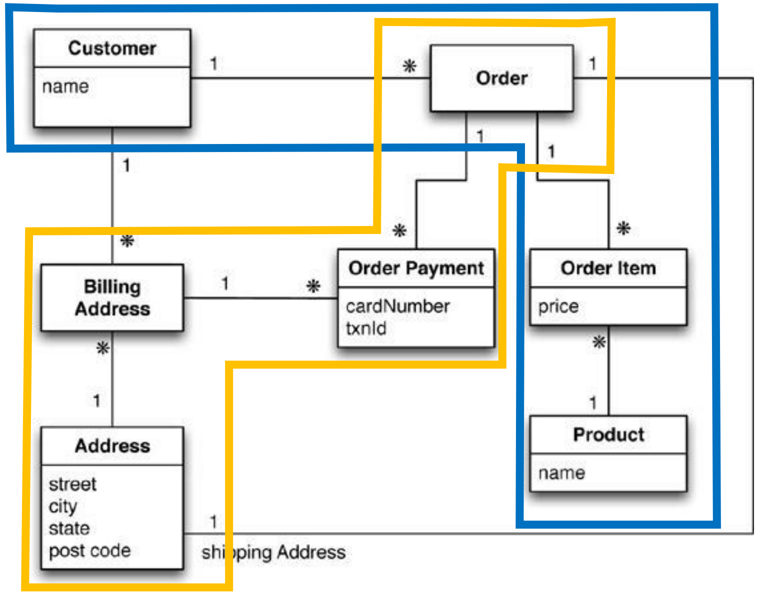
Data modeling: wide-column

Order details column family

Ord	CustName	Pepsi	Cola	Fanta	…
1	Martin		12.4	14.4	
2	…	…			…

Order payments column family

<table>
<tr><th>Ord</th><th>OrderPayments</th></tr>
<tr><td>1</td><td>
<table>
<tr><th>Card</th><th>Steet</th><th>City</th><th>State</th><th>Code</th></tr>
<tr><td>477</td><td>9th</td><td>NewYork</td><td>NewYork</td><td>10001</td></tr>
<tr><td>457</td><td>Adam</td><td>Chicago</td><td>Illinois</td><td>60007</td></tr>
</table>
</td></tr>
<tr><td>2</td><td>…</td></tr>
</table>



Summary

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 (group the data that are used altogether)
- Facilitate the developer's job (by surpassing the impedance mismatch problem)

Disadvantages

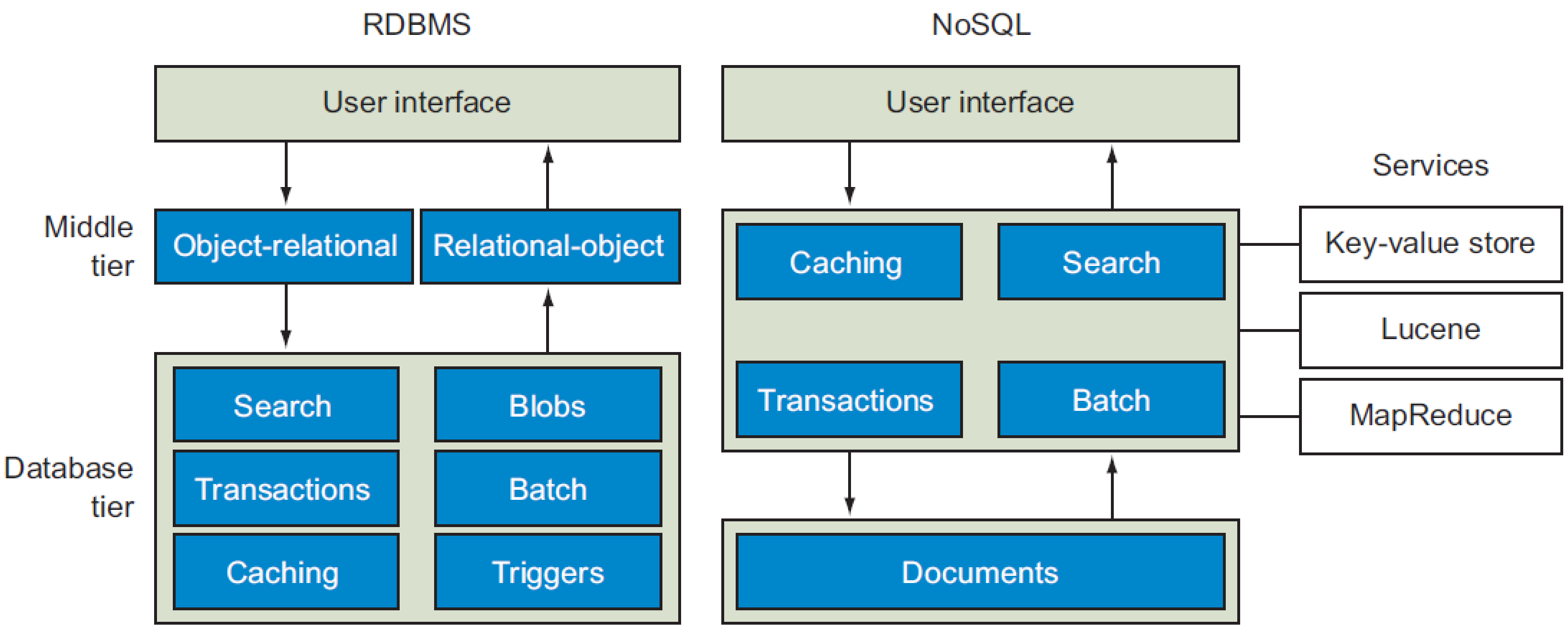
- No real design strategy exists for aggregates
  - It mainly depends on how data are meant to be used
- Can optimize only a limited set of queries
- Data denormalization → possible inconsistencies

A data-driven approach is agnostic to how the data is used (mostly)

Dealing with data consistency

In short

RDBMS vs NoSQL: different philosophies



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 NoSQL: CAP

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

Consistency: the system is always consistent

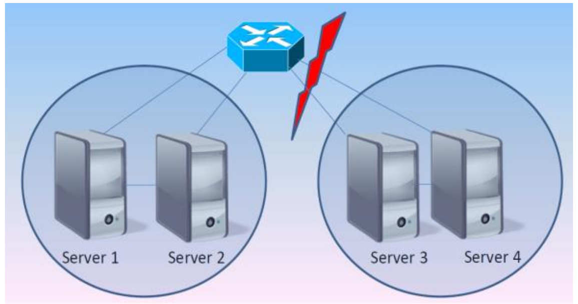
- Every node returns the same, most recent, successful write
- Every client has the same view of the data

Availability: the system is always available

- Every non-failing node returns a response for all read and write requests in a reasonable amount of time

Partition tolerance: the system continues to function and upholds its consistency guarantees in spite of network partitions

- In distributed systems, network partitioning is an *inevitable possibility*



Consistency in NoSQL: CAP

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)

Theorem interpretation is not trivial

- Asymmetric properties: consistency is sacrificed to favor speed at all times, not just when partitioning happens
- Different application requirements → different algorithms handle these properties more strictly/loosely

Consistency in NoSQL: CAP relaxed

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

CP: no one can book (A is sacrificed)

- Not the best solution

AP: both can book (C is sacrificed)

- Possible overbooking: writing conflict to handle

caP: only one can book

- The other will se 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 to obsolete reads
- How large can the inconsistency window be

Consistency in NoSQL: summary

NoSQL databases rely on the *eventual consistency* principle

- Trade absolute consistency for latency
- Inconsistencies are tolerated within short time windows

Still, *unrecoverable inconsistencies* may arise

- Because of data denormalization
- Because of the absence of transaction

One size does not fit all

Key-Value: popular DBs

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

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

Memcached DB: http://memcached.org/

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

Key-Value: when to use

Very simple use cases

- Independent data (no need to model relationships)
- The typical query is a simple lookup
- Need super-fast performance

Examples

- **Session information**
  - 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, timezone, 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
- Some systems offer limited link-walking mechanisms

Multi-record operations

- Because operations (mostly) involve one record at a time

Querying the data

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

Document: popular DBs

MongoDB: http://www.mongodb.org

Couchbase: http://www.couchbase.com

CouchDB: http://couchdb.apache.org

Document: when to use

Higher expressiveness

- Store data according to a highly nested data model
- Need to formulate complex queries on many fields

Examples

- **Event logs**
  - Central repo to store event logs from many applications; shard on app name or event type
- **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

Advertising 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

ACID transactions requirement

- 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

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

Wide column: when to use

Compromise between expressiveness and simplicity

- Limited (but some) requirements in terms of data model
- Limited (but some) requirements in terms of querying records

Examples

- **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

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.

Wide column: when to avoid

Same as for document model

- ACID transactions requirement
- Queries on high-variety data

Need for full query expressiveness

- Joins are usually *not* supported
- Limited support for filters and group bys

Graph: popular DBs

Neo4J: http://neo4j.com

TigerGraph: https://www.tigergraph.com/

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

So... are RDBMSs dead?

No! Polyglot persistence

Different databases are designed to solve different problems

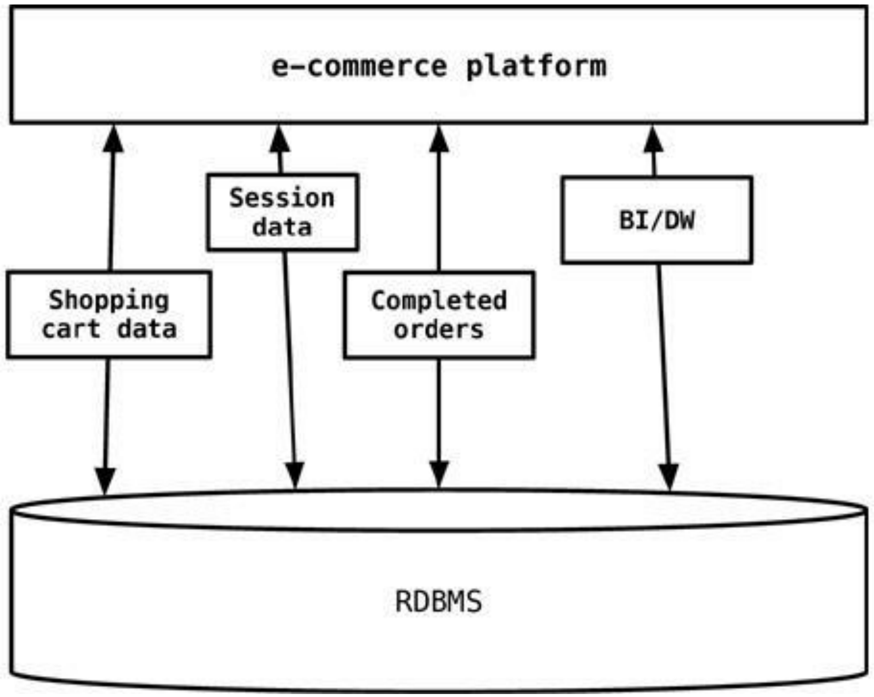
Using a single DBMS to handle everything (the *one size fits all*)…

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

… usually lead to inefficient solutions

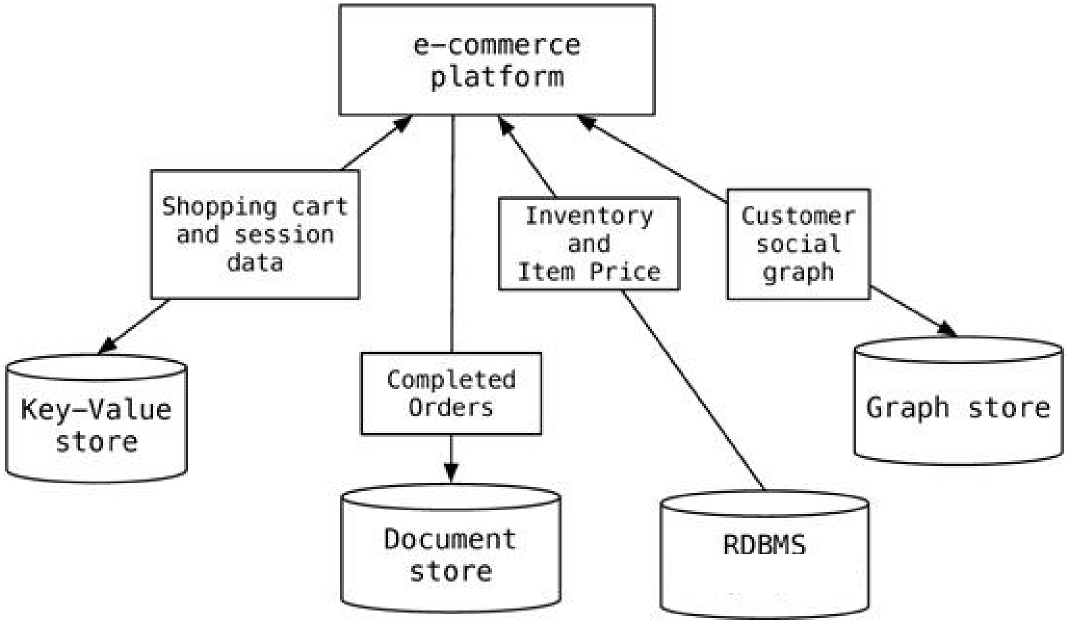
Each activity has its own requirements (availability, consistency, fault tolerance, etc.)

One size fits all approach



Polyglot data management

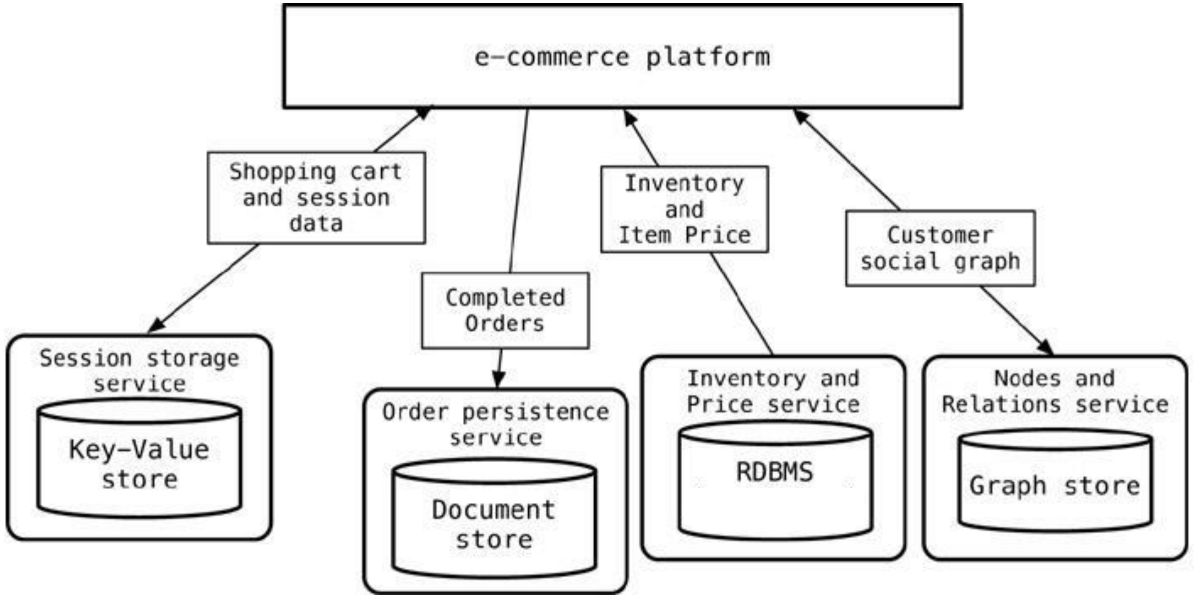
Beware: this leads to harder cross-database consistency management



Polyglot data management

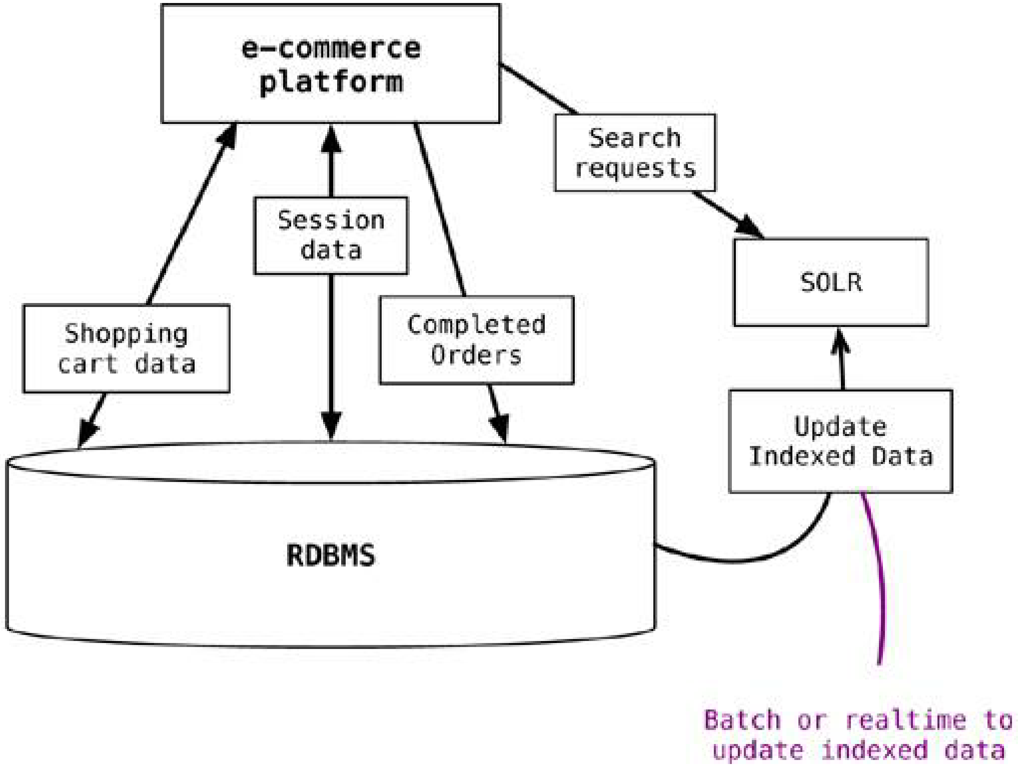
Each DB should be "embedded" within services

- Data access and manipulation enabled with API services



Supporting existing technologies

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



Beyond NoSQL

NewSQL systems

- Combine the benefits from both relational and NoSQL worlds
- Ensure scalability without compromising consistency…
- …but by compromising some availability!

Multi-model systems

- Support multiple data models
- The barrier between distributed and centralized DBMSs still remains

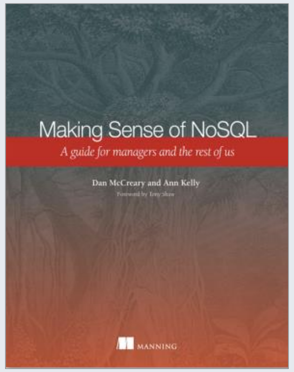
Database-as-a-service

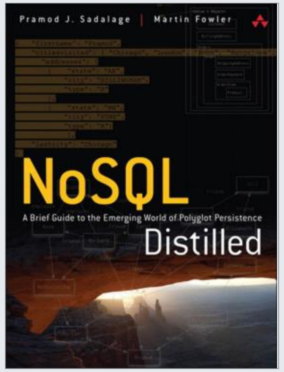
- All cloud providers offer storage services supporting all data models

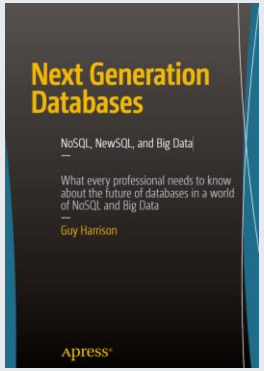
NoSQL Databases

Complementary to RDBMSs

- Different data models
- Better scaling
- Trade absolute consistency for latency
- NewSQL and multi-model systems also exist







Want to know more?

← Suggested readings

- [Distributed algorithms in NoSQL databases]
- Email me at m.francia@unibo.it
- See you next year!