Aggregate data models
Document-based DBs

Roberto Maria Delfino
delfino@diag.uniroma1.it

Sapienza – University of Rome
Department of Computer, Control and Management Engineering

Slides adapted from those of the Large-Scale Data Management course
held by prof. Domenico Lembo

Bibliographic References

- The main bibliographic reference for this part is: [SaFo13] ***NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence***. Pramod J. Sadalage & Martin Fowler. Addison Wesley. 2013. The following slides rework part of the book content.

- The formal part on JSON is adapted from the article

  Pierre Bourhis, Juan L. Reutter, Fernando Suárez and Domagoj Vrgoč. ***JSON: data model, query languages and schema specification***. In Proc. of PODS 2017. available at http://dl.acm.org/authorize?N37868

Quick recap

Relational databases have represented the default choice for data storage for several decades.

Some reasons:

- Data kept in storage in a **structured** way
- Simple and **general** data model
- Efficient concurrency management
- Standardized query language

Quick recap

Relational databases have represented the default choice for data storage for several decades.

Some reasons:

- Data kept in storage in a **structured** way
- Simple and **general** data model
- Efficient concurrency management
- Standardized query language

Bucolic relational landscape

Good things don’t last forever

Some objections have been raised to the relational model:

- Advent of Big Data required a shift towards distributed computing (volume, velocity), which the relational model was not originally designed for
- Impedence mismatch (OOP vs in-memory data structures)
- Costly query processing
- Rigid schema does not fit well with variety

These and other reasons led to the birth of the **NoSQL** movement.

A useful naming confusion

No agreement on what NoSQL meant
(even in the NoSQL movement itself).

$$\begin{tikzcd}
\text{SQL} \arrow[d] \\
\text{Not SQL!} \arrow[d] \\
\text{Not only SQL} \arrow[d] \\
\text{No, SQL!}
\end{tikzcd}$$

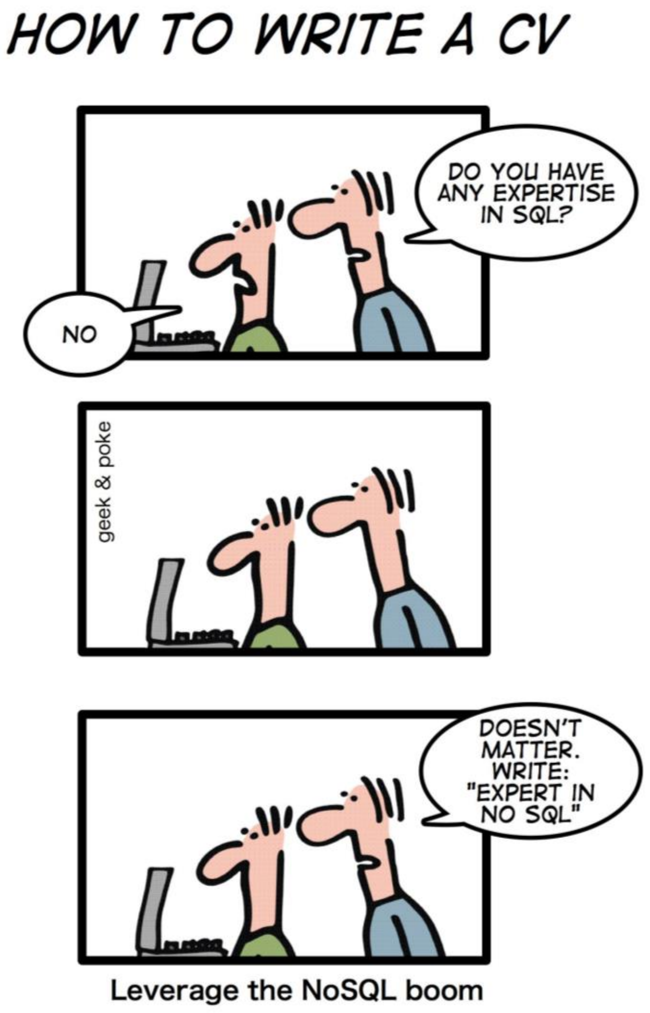
Nowadays Not only SQL is the
accepted meaning.

A useful naming confusion

No agreement on what NoSQL meant (even in the NoSQL movement itself).

$$\begin{tikzcd}
\text{SQL} \arrow[d] \\
\text{Not SQL!} \arrow[d] \\
\text{Not only SQL} \arrow[d] \\
\text{No, SQL!}
\end{tikzcd}$$

Nowadays Not only SQL is the accepted meaning.


HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom

Scaling up vs Scaling out

The need to manage larger and larger data made the increasing of the machines’ computational powers necessary. Two main approaches exist.

Scaling up: increasing computational power of a single machine (more CPU, more storage, more RAM, etc.). This solution can be very expensive and it is prone to physical limitations.

Scaling out: increasing the number of less powerful machines organized in clusters. Cheaper solution which also provides more resilience to failures of single nodes.

Sharding

Sharding is the technique used to partition data among clusters of machines. Each portion is called **shard** and is a subset of the whole database.

Sharding can improve performances by divinding the workload.

Problems:

- How to define the **granularity** of shards?
- How to keep track of where each part of the database is located in the cluster?
- What about **referential integrity** and **consistency management**?

Impedance mismatch

The problem of impedance mismatch arises when two cooperating systems adopt different data models, different data structures or communication interfaces.

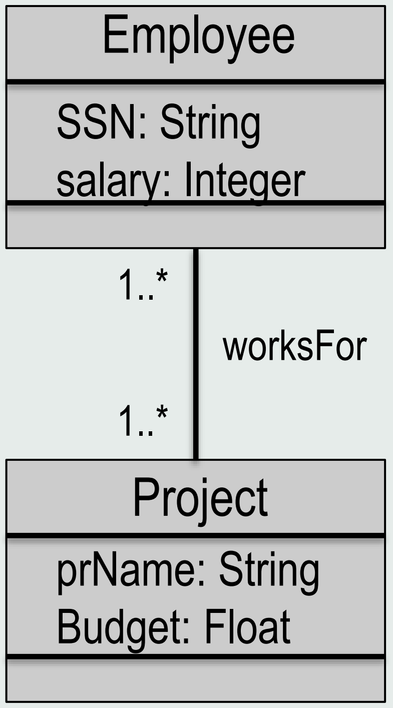
In Object-Oriented Programming data is represented as objects with **complex structures**, while the very same data in a relational model is represented by means of **tables** and **records**.

This potentially yields unwanted issues related to the complexity of mapping data from one representation to the other when executing queries.

Possible solutions: Object-Oriented DBs; Object-Relational Mapping frameworks; **different data models**

Impedance mismatch - Example

in-memory data structure



Actual data is stored in a DB:

D1[*Code*: Int, *Salary*: Int, *SSN*: String]
Employee's Code with salary and SSN

D2[*Code*: Int, *PrName*: String]
Employees and Projects they work for

D3[*PrName*: String, *Budget*: Float]
Project's name and budget

Conceptually:
– An Employee is identified by her/his *SSN*.
– A Project is identified by its *name*.
Thus,
(i) an employee should be created from her *SSN*;
(ii) a project should be created from its *PrName*

Avoiding JOINS

Avoiding JOINS

WHY DO NOSQL DEVELOPERS EAT LUNCH ALONE?
I DON'T KNOW, WHY?
THEY DON'T KNOW HOW TO JOIN TABLES
imgflip.com

Main NoSQL characteristics

The main characteristics of NoSQL systems are:

- They are **schemaless**
- They do not rely on (just) SQL
- They are generally driven by the need to run on clusters (graph databases do not typically fall in this class)
- They generally not handling consistency through ACID transactions (but notice that graph databases instead support transactions)

For the aforementioned reasons, NoSQL had to be based on a data models which were different from the relational one.

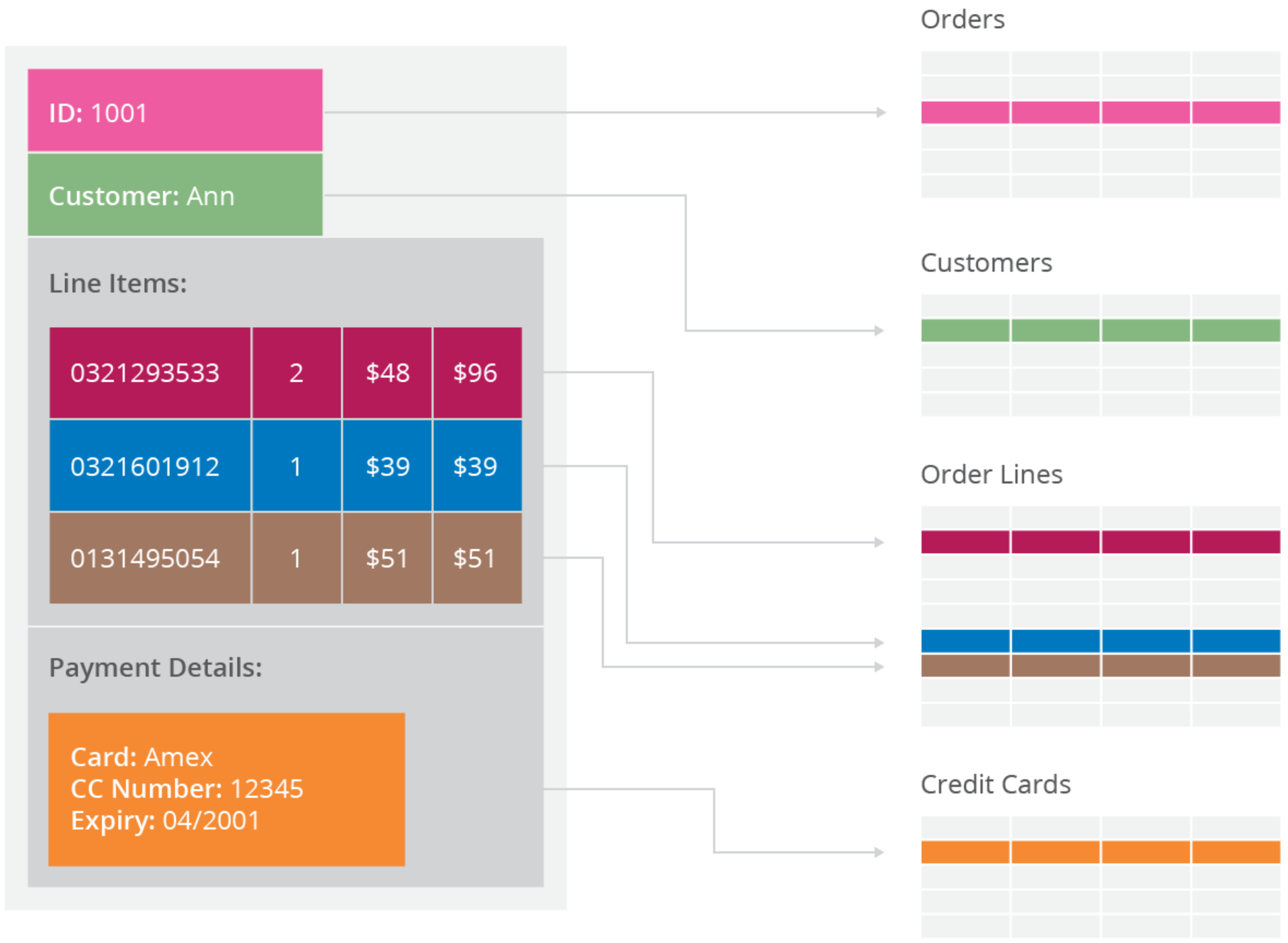
From tuples to aggregates

The relational model organizes information into tuples, which are very simple mathematical structures, e.g., the value of an attribute cannot be a list, or a tuple itself.

The concept of **aggregate** was introduced as a data representation alternative aiming at providing more complex data structures. Intuitively, an aggregate can be seen as a record allowing for lists and for nested records inside it, even though no universally formalized definition of aggregate exists.

Generally speaking, the aggregate is a complex structure used to collect and group data which we want to treat as a unit.

Example of aggregate



Consequences of Aggregate Orientation

The fact that an order consists of order items, a shipping address, and a payment can be expressed in the relational model in terms of foreign key relationships but there is nothing to distinguish relationships that represent aggregations from those that don’t. As a result, the database cannot use a knowledge of aggregate structure to help it store and distribute the data.

Aggregation is not a logical data property: it is all about how the data is being used by applications - a concern that is often outside the boundary of data modeling.

Also, an aggregate structure may help with some data interactions but be an obstacle for others (in our example, to get the product sales history, you’ll have to dig into every aggregate in the database).

Consequences of Aggregate Orientation

The clinching reason for aggregate orientation is that it helps greatly with running on a cluster.

Aggregate orientation fits well with scaling out because the aggregate is a **natural unit to use for distribution**.

On the other hand, slicing aggregates for more fine grained access to them may become very difficult.

Consequences of Aggregate Orientation

Aggregates have an important consequence for **transactions**.

Relational DBs allow you to manipulate combinations of rows from any tables in a single (**ACID**) transaction (i.e., Atomic, Consistent, Isolated, and Durable).

It is often said that NoSQL databases don't support ACID transactions and thus sacrifice **consistency** (this is however not true for graph databases, which are, as relational DBs, aggregate-agnostic).

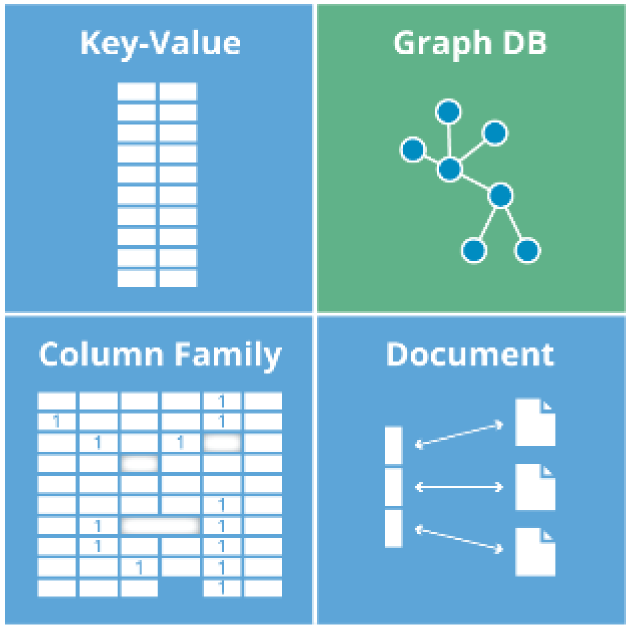
In general, aggregate-oriented databases don't have ACID transactions that span multiple aggregates. Instead, they support **atomic manipulation of a single aggregate at a time**. Thus, if we need to manipulate multiple aggregates in an atomic way, we have to manage that in the application code!

In practice, we find that most of the time we have to keep our atomicity needs within a single aggregate; indeed, that is part of the consideration for deciding how to divide up our data into aggregates.

Aggregate Data Models

Among the NoSQL data models, three of them are based on aggregates and they have different characteristics (and share some similarities):

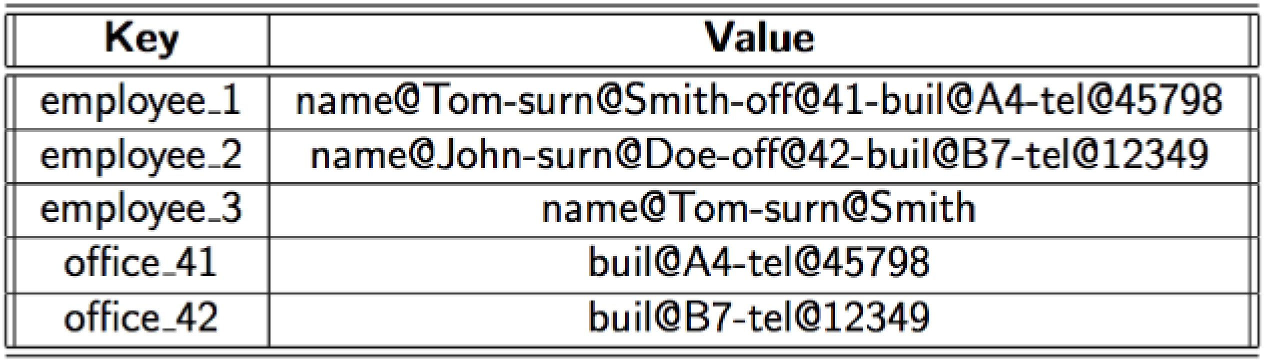
- **Key-Value Store**
- **Column-Family**
- **Document-based**



Key-Value Data Model

In a key-value database, the aggregate is just a **blob of bits**. Thus, we can store whatever we like in the aggregate. The aggregate is **accessed and managed as a whole**. It is not possible to retrieve single portions of data constituting the aggregate. That is why it is responsibility of the application to manage the aggregate in order to understand what is stored inside it and access specific data.

Key-value stores always use **primary-key access**, and thus are very performant. They essentially act like large, distributed hashmap data structures.



Key	Value
employee_1	name@Tom-surn@Smith-off@41-buil@A4-tel@45798
employee_2	name@John-surn@Doe-off@42-buil@B7-tel@12349
employee_3	name@Tom-surn@Smith
office_41	buil@A4-tel@45798
office_42	buil@B7-tel@12349

Column-Family Data Model

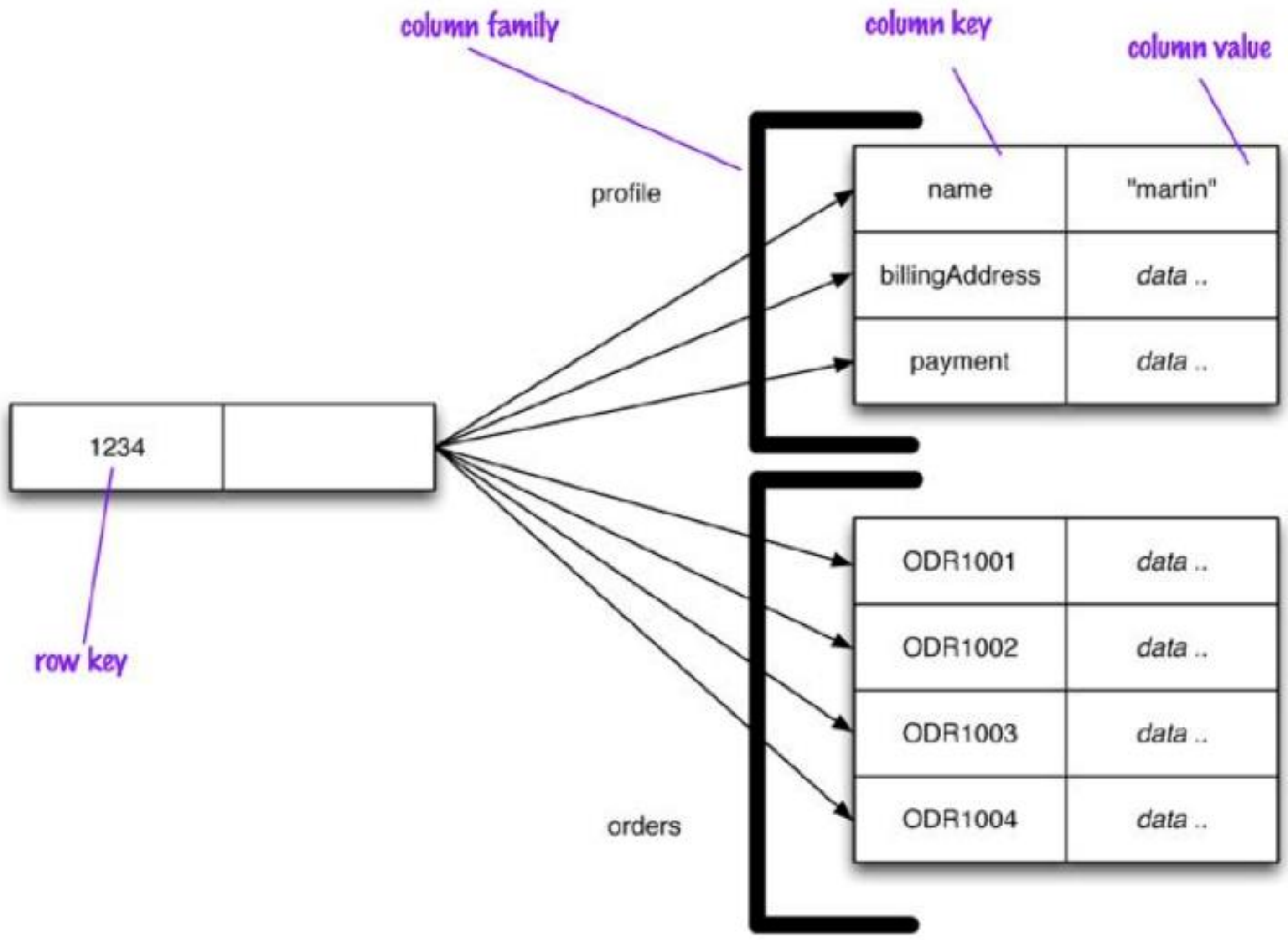
The column-family model can be seen as a two-level aggregate structure.

As with key-value stores, the first key is often described as a **row identifier**, picking up the aggregate of interest.

The corresponding row aggregate is itself is made up of a map of more detailed values. These second-level values are referred to as **columns**, each being a **key-value pair**.

Columns are organized into **column-families**. These are groups of related data typically accessed together. Each column has to be part of a single column-family.

Column-Family Data Model



Column-Family Data Model

Two ways to think about how the data are structured exist:

- **Row-oriented**: Each row is an aggregate (for example, customer with the ID 1234) with column-families representing useful portions of data (e.g., profile, orders within that aggregate). Under this perspective, a row plays the same role as a document in document databases (see next slides).

- **Column-oriented**: Each column-family defines a record type (e.g., profile). You then think of a row as the join of records in all column families. Column-families can be then to some extent considered as tables in RDBMSs. Unlike table in RDBMSs, a Column-family can have different columns for each row it contains.

Supercolumn-Family

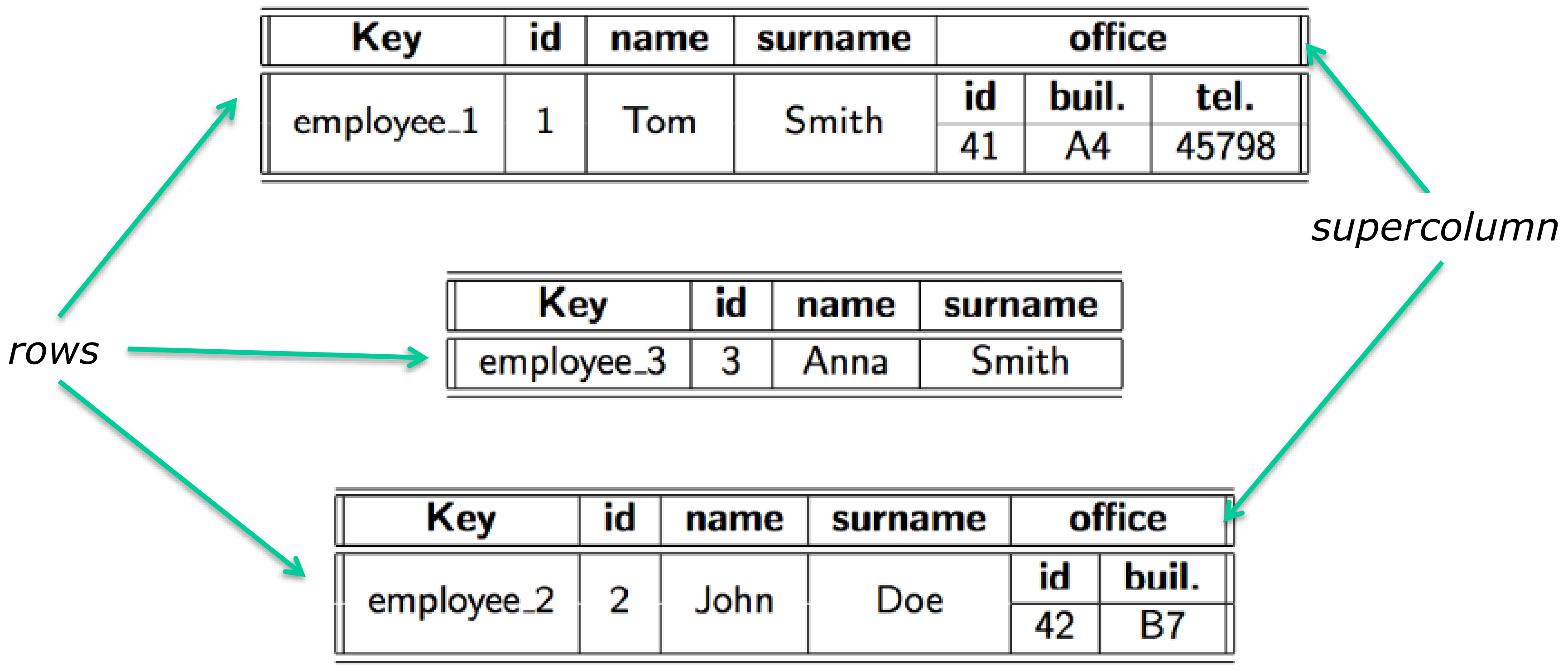
In some systems, a column can also consist of (a name and) a map (that is, a dictionary) of other columns. In this case, it is called **supercolumn**.

If a column-family contains a supercolumn, it is also called supercolumn-family

It has to be noted however that the use of supercolumns may seriously slow down query performances. Indeed, all sub-columns of a supercolumn family must be de-serialized to read a single sub-column family. Thus, they should be used with care.

Supercolumn-Family Example

SuperColumn-family: *Employees*


supercolumn
rows

Key	id	name	surname	office		
				id	buil.	tel.
employee_1	1	Tom	Smith	41	A4	45798

Key	id	name	surname
employee_3	3	Anna	Smith

Key	id	name	surname	office	
				id	buil.
employee_2	2	John	Doe	42	B7

Document-based Data Model

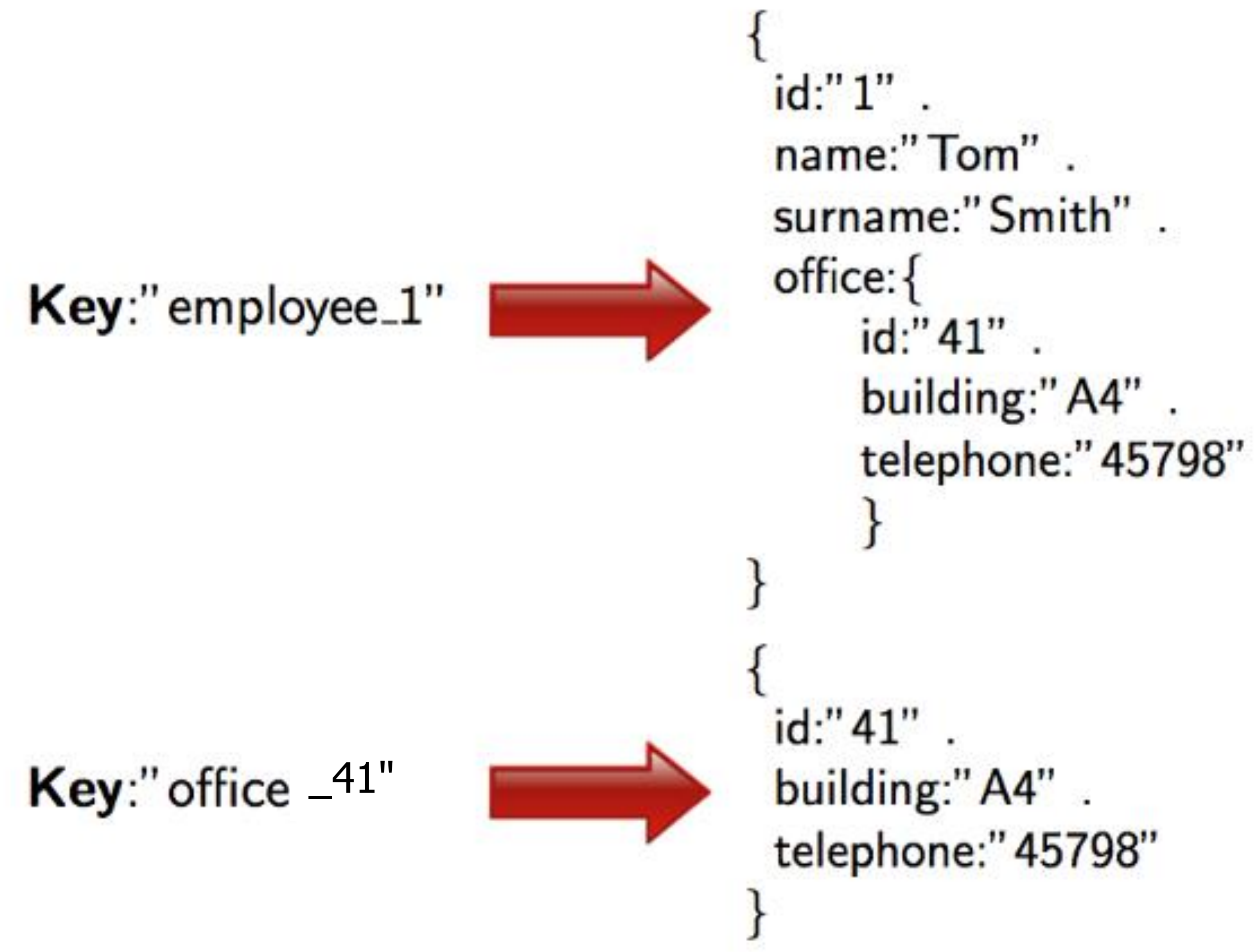
Document-based systems rely on aggregate structures (called **documents**) providing a higher level of complexity with respect to both the key-value and the column-family data models.

A document is structure consisting of a **set of key-value pairs**, where values can be complex structures themselves.

In terms of structure, a document can be seen as an object in Object-Oriented Programming, allowing for an arbitrary level of complexity, in that it allows for complex structures to be used as values of key-value pairs, thus providing nesting capabilities.

Documents are typically represented through the **JSON** format.

Documents in JSON format



Key:"employee_1"

```
{
 id:"1" .
 name:"Tom" .
 surname:"Smith" .
 office:{
       id:"41" .
       building:"A4" .
       telephone:"45798"
       }
}
```

Key:"office _41"

```
{
 id:"41" .
 building:"A4" .
 telephone:"45798"
}
```

JavaScript Object Notation (JSON)

JSON documents are **dictionaries** consisting of key-value pairs, where **the value can again be a JSON document**, thus allowing an arbitrary level of nesting.

Example:

```
{
"name": {
      "first": "John",
      "last": "Doe"
     },
"age": 32,
"hobbies": ["fishing","yoga"]
}
```

As we can see JSON supports arrays and atomic types, such as integers and strings

JSON formalization (simplified)

Let Σ be the set of all unicode characters. JSON values are defined as follows:

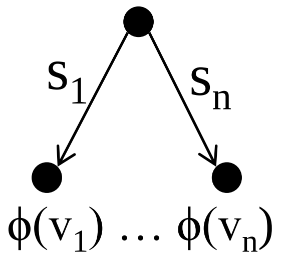
- any (signed real) number n is a JSON value, called **number**.
- if s is a string in Σ , then "s" is a JSON value, called **string**.
- the special symbols true and false are a JSON values, called **Booleans**.
- the special symbol null is a JSON value
- if $v_1,..,v_n$ are JSON values and $s_1,..,s_n$ are pairwise distinct string values, then $\{s_1:v_1,...,s_n:v_n\}$ is a JSON value, called **object** (or document). Each $s_i:v_i$ is called a *key-value pair* of this object. No object can have two (or more) pairs with the same key. If n=0, we have the empty object {}
- if $v_1,..,v_n$ are JSON values then $[v_1,..,v_n]$ is a JSON value called **array**. In this case $v_1,..,v_n$ are called the *elements of the array*.

JSON trees

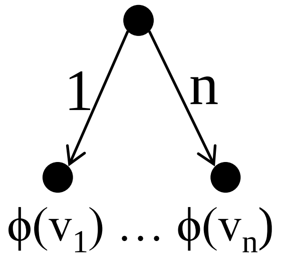
We define the transformation ϕ for JSON documents such that

- $\{\} \xrightarrow{\phi} \bullet$
- If v is a JSON number, string, true/false, or null, then

  $v \xrightarrow{\phi} \underset{v}{\bullet}$
- $\{s_1{:}v_1,\ldots,s_n{:}v_n\} \xrightarrow{\phi}$

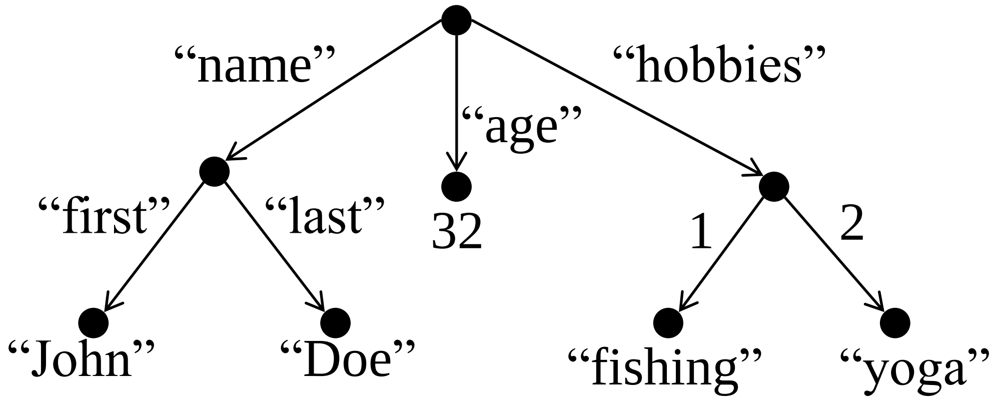


- $[v_1,\ldots,v_n] \xrightarrow{\phi}$



JSON tree - Example

```
{"name": {
           "first": "John",
           "last": "Doe"
          },
 "age": 32,
 "hobbies": ["fishing","yoga"]}
```



Key-value vs Document based comparison

With a key-value store, we can only access an aggregate by **lookup** based on its key.

In document databases, at the simplest level, documents can be stored and retrieved by ID (as key-values stores). However, in general, we can submit queries to the database referring to the fields in the aggregate, i.e., **we can retrieve part of the aggregate rather than the whole thing**, and the database can create indexes based on the contents of the aggregate. In general, **indexes** are used to retrieve sets of related documents from the store.

As usual, indexes speed up read acceesses but slow down write accesses, thus should be designed carefully.

Key-value vs Document based comparison

In practice, the line between key-value and document databases gets a bit blurry.

People often use a document database to do a simple key-value style look-up.

Databases classified as key-value databases may allow you some structures for data.

For example:

- Redis allows you to break down the aggregate into lists or sets
- Riak allows you to put aggregates into buckets. Others support querying by search tools.

Aggregate DBs: wrapping up

All aggregate data models are based on the notion of an aggregate indexed by a key that you can use for lookup. Within a cluser, all the data for an aggregate should be stored together on one node. The aggregate also acts as the atomic unit for management.

- The **key-value** data model treats the aggregate as an opaque whole (no access to portion of an aggregate is allowed). Great performances are allowed but the aggregate has to be understood at the application level.
- The **document-based** model makes the aggregate transparent, allowing you to do queries and partial retrievals. However, since the document has no schema, the database cannot act much on the structure of the document to optimize the storage.
- **Column-family** models divide the aggregate into column-families, allowing the database to treat them as units of data within the row aggregate. This imposes some structure on the aggregate and allows the database to take advantage of that structure to improve its accessibility

Implications of schemaless models

While the schemaless nature of NoSQL systems offers great flexibility, it can introduce undesired problems.

Indeed, whenever we write a program that accesses data, that program almost always relies on some form of **implicit schema**: it will assume that certain field names are present and carry data with a certain meaning, and assume something about the type of data stored within that field.

Having the implicit schema in the application means that in order to understand what data is present you have to dig into the application code. Furthermore, the database cannot use the schema to support the decision on how to store and retrieve data efficiently. Also, it cannot impose integrity constraints to maintain information coherent.

Data Modeling

Despite several NoSQL tools have been developed in the last years, and various technical solutions have been proposed so far, to date no methodolgies have been developed to guide the database designer in the modeling of a NoSQL database.

This contrasts with the well established methodologies available for the design of a relational database.

This is however justified by the fact that NoSQL data models and technologies are more recent. Also, the schemaless nature of NoSQL databases makes the notion of database design inherently blurry.

Methodologies need to be devised to both (i) model data (e.g., decide the form of aggregates in aggregate DBs), and (ii) distribute data on a cluster.

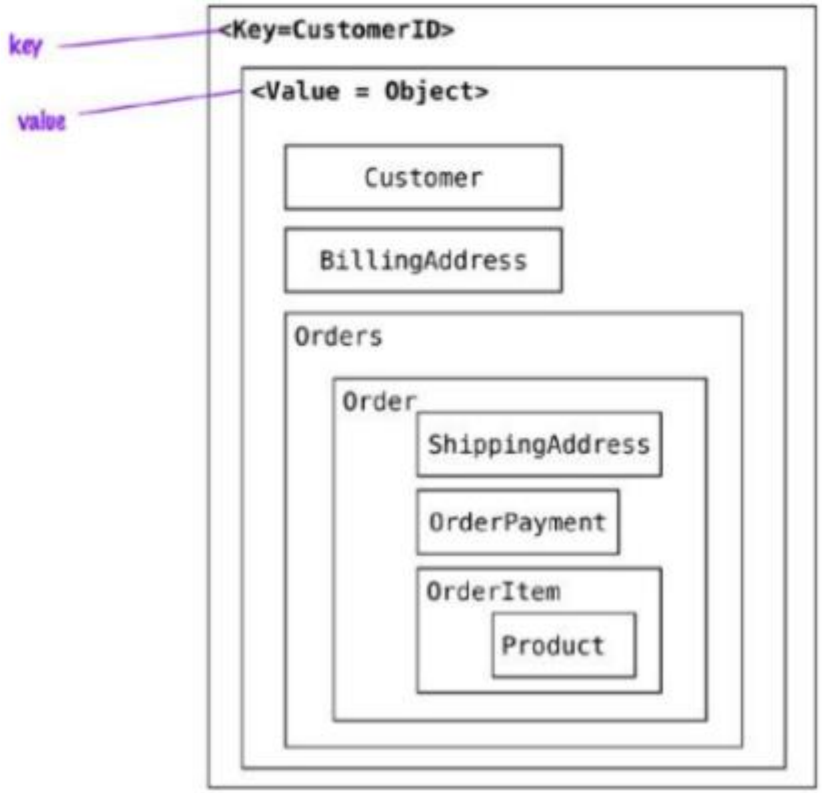
Data Modeling

When modeling data aggregates we need to consider **how the data is going to be accessed** (and what are the side effects with the chosen aggregates)

Example: Let's start with the model where all the data for the customer is embedded using a key-value store.

- The application can read the customer's information and all the related data by using the key.

- To read the orders or the products sold in each order, the whole object has to be read and then parsed on the client side.

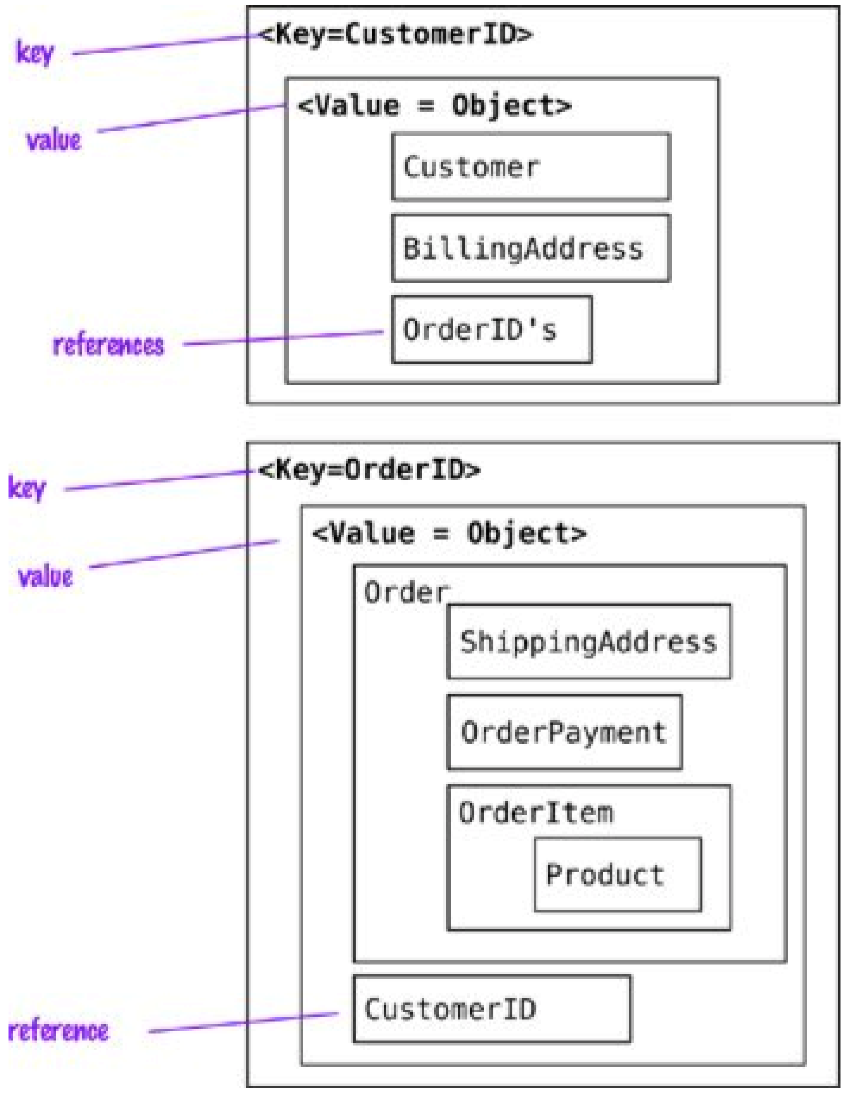


Data Modeling

It might be convenient to split the value object into Customer and Order objects and then maintain these objects reference.

We can now find the orders independently from the Customer, and access then the customer using the CustomerID reference in the Order, whereas with the OrderId in the Customer we can find all Orders for the Customer. In a key-value store this cannot be done on the server side, and it is the client that reads the references and makes further accesses to retrieve the referred objects.

Using aggregates this way allows for read optimization, but we have to push the OrderId reference into Customer for every new Order (which again has to be done at the client-side).



Data Modeling – Document-based DB

In document stores, since we can query inside documents, we can find all Orders for the Customer even removing references to Orders from the Customer object. This change allows us to not update the Customer object when orders are placed by the Customer.

```
# Customer object
{
"customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}],
"payment": [
  {"type": "debit",
  "ccinfo": "1000-1000-1000-1000"}
  ]
}

# Order object
{
"orderId": 99,
"customerId": 1,
"orderDate":"Nov-20-2011",
"orderItems":[{"productId":27, "price": 32.45}],
"orderPayment":[{"ccinfo":"1000-1000-1000-1000",
        "txnId":"abelif879rft"}],
"shippingAddress":{"city":"Chicago"}
}
```

Key-Value store tools

Some popular key-value stores are Riak, Redis (also referred to as Data Structure Server store), Memcached DB, Oracle Berkeley DB (embedded), Amazon DynamoDB (not open-source), Project Voldemort (an open-source implementation of Amazon DynamoDB)

Column-Family tools

Column family stores are modeled on Google's BigTable. The data model is based on a sparsely populated table whose rows can contain arbitrary columns.

Google BigTable

cassandra

HYPERTABLE INC

APACHE HBASE

aws

Amazon SimpleDB

Note: These databases are often referred to as column stores, but this name has been around for a while to describe a different object: DBMSs like MonetDB or Vertica, adopted SQL and the relational model. The thing that made them different was the way in which they physically stored data, based on columns rather than on rows as a unit for storage (this storage system is particularly suited to speed up read accesses)

Document DB tools

Some document stores are MongoDB, Couchbase, CouchDB, Terrastore, OrientDB (which is also a graph DBMS), RavenDB, but also Lotus Notes, which adopts document storage.