NoSQL Databases

Donato Summa

Eurostat

NoSQL: Definition

- NoSQL databases is an approach to data management that is useful for very large sets of distributed data
  - **NoSQL should not be misleading: the approach does not prohibit Structured Query Language (SQL)**
  - **And indeed they are commonly referred to as "NotOnlySQL"**

NoSQL: Main Features

- Non relational/Schema Free: little or no pre-defined schema, in contrast to Relational Database Management Systems
- Distributed
- Horizontally scalable: able to manage large volume of data also with availability guarantees
    - Transparent failover and recovery using mirror copies

Classification of NoSQL DB

- Key-values
- Document store
- Column Oriented
- Graph databases

The first three share a common characteristic of their data models which we will call aggregate orientation.
An aggregate is a collection of related objects that we wish to treat as a unit

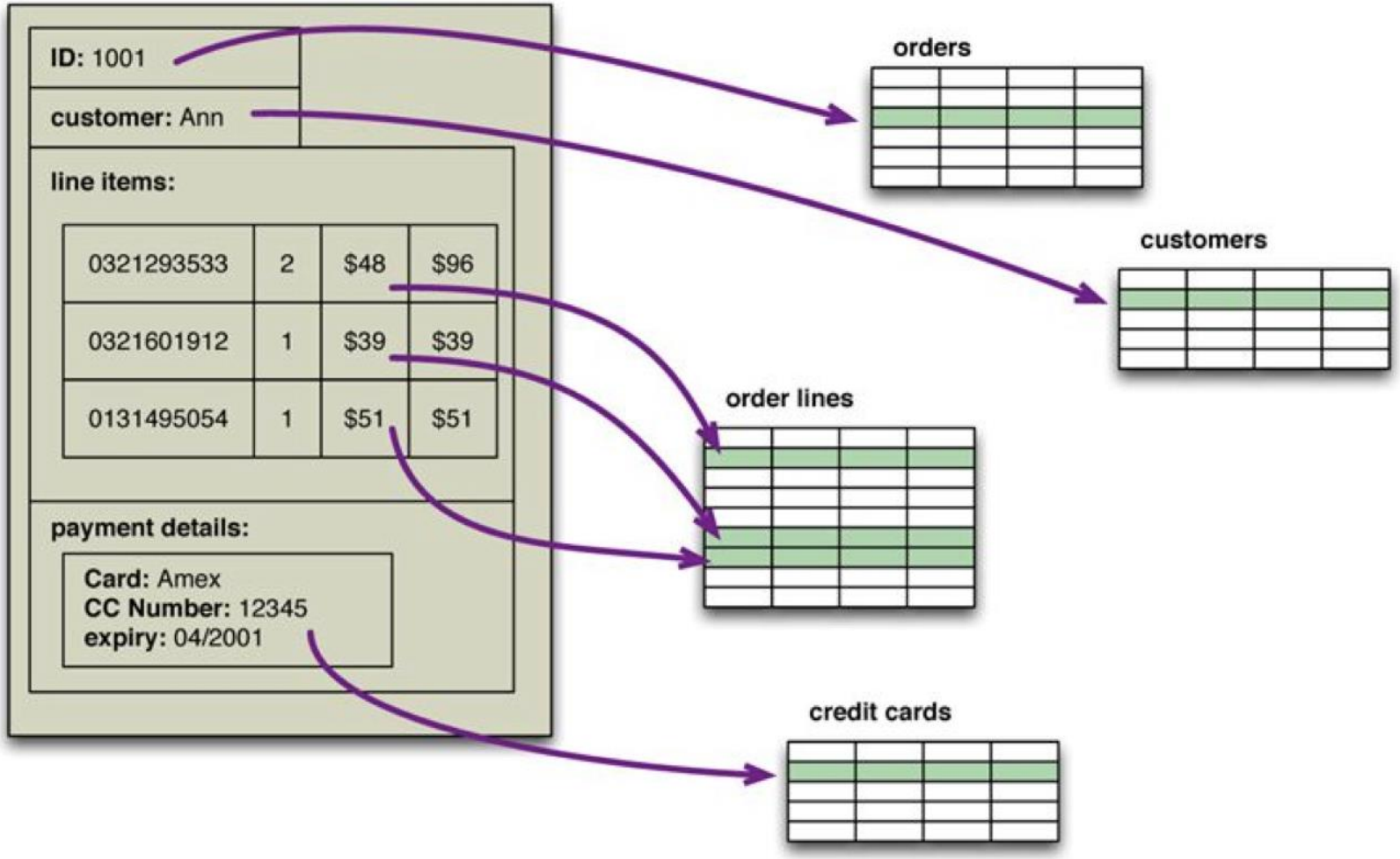
NoSQL: Aggregate - 1

- **The relational model divides the information** that we want to store **into tuples** (rows): this is a very simple structure for data
- **Aggregate orientation** takes a different approach. It recognizes that often you want to operate on data in units that have a more complex structure
- It can be handy to think in terms of a complex record that allows lists and other record structures to be nested inside it

NoSQL: Aggregate - 2

- As we will see, key-value, document, and column-family databases all make use of this more complex record
- However, there is no common term for this complex record; we use here the term **aggregate**

NoSQL: Aggregate



NoSQL: BASE Consistency

- Eventually consistent/not ACID
  - **Informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value**
  - **BASE (Basically Available, Soft state, Eventual consistency) semantics in contrast to**
  - **traditional ACID (Atomicity, Consistency, Isolation, Durability) guarantees**

NoSQL: BASE Consistency

- ACID (Strong Consistency)
  - **Atomicity: every transaction is executed in "all-or-nothing" fashion**
  - **Coherence (or consistency): every transaction preserves the coherence with constraints on data (i.e., at the end of the transaction constraints are satisfied by data)**
  - **Isolation: transaction does not interfere. Every transaction is executed as if it was the only one in the system (every seralization of concurrent transactions is accepted)**
  - **Durability: after a commit, the updates made are permanent regardless possible failures**

NoSQL: BASE Consistency

- Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux
  - **Basically Available:** The availability of BASE is achieved through supporting partial failures without total system failure.
  - **Soft state:** data are "volatile" in the sense that their persistence is in the hand of the user that must take care of refresh them
  - **Eventual Consistency:** the system eventually converge to a consistent state

NoSQL: BASE Consistency

- A common claim we hear is that NoSQL databases don’t support transactions and thus can’t be consistent
- Any statement about lack of transactions usually only applies to some NoSQL databases, in particular the aggregate-oriented ones, whereas graph databases tend to support ACID transactions

Key-Values Databases

- Key–value databases allow applications to store data in a schema-less way
- The data could be stored in a datatype of a programming language or an object
  - **No fixed data model**
- Example of tools:
  - **Riak, Redis, Amazon Dynamo DB**

Key-Values Databases: Example

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

Ref: Domenico Lembo, Master Course on Big Data Management

Key-Values Databases: Issues

- You can store whatever you like in Values
  - **It is the responsibility of the application to understand what was stored**

Document Store

- Differently from key-values, there are some limits on what we can place in it:
  - **Definition of allowable structures and types**
  - **More flexibility when accessing data:** For example, you may want a query that retrieves all the documents with a certain field set to a certain value.
- Examples: Mongo DB, Apache SOLR, Elasticsearch

Document Store: Example (JSON)

Key:"employee_1"

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

Key:"office_1"

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

Document Store: Issues

- Indexing: necessary for speed-up accesses
  - **Indexes can be very big**
- Semantics problem still there:
  - **Need for semantic extraction techniques**

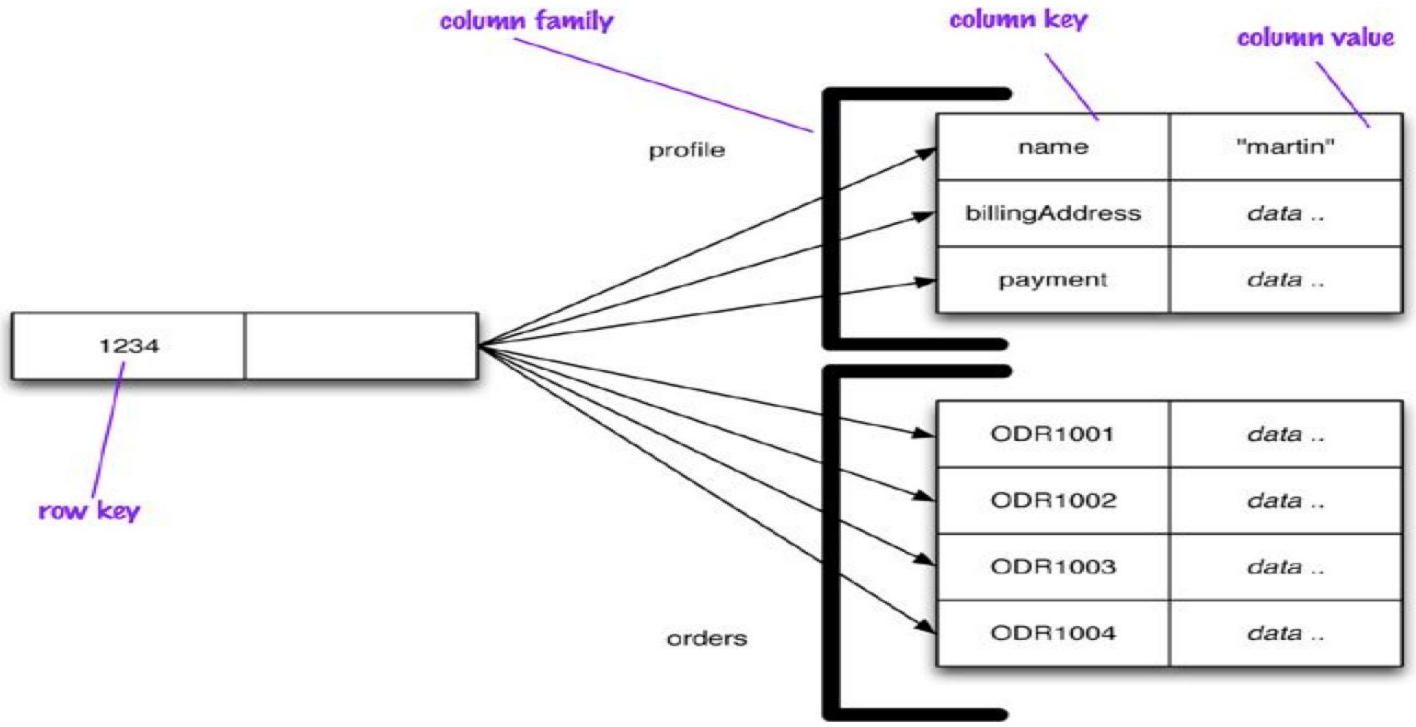
Column-oriented DB

- 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

Column-oriented DB

- 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**
  - **This row aggregate is itself formed of a map of more detailed values. These second-level values are referred to as columns, each being a key-value pair**
  - **Columns can be organized into column families**

Column-oriented DB: Example



Ref: Domenico Lembo, Master Course on Big Data Management

Column-oriented DB: Structure - 1

- **Row-oriented**
  - Each row is an aggregate (for example, customer with the ID of 1234) of values
  - column families are useful chunks of data (profile, order history) within that aggregate

Column-oriented DB: Structure - 2

- **Column-oriented:**
  - Each column family defines a record type (e.g., customer profiles) with rows for each of the records. 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 (but a Column Family can have different columns for each row it contains)
- **Example: Google's BigTable, Cassandra, HBase**

Graph Database

- A graph database is a database that uses graph structures with nodes, edges, and properties to represent and store data
- A management systems for graph databases offers Create, Read, Update, and Delete (CRUD) methods to access and manipulate data
- Graph databases can be used for both OLAP (since are naturally multidimensional structures) and OLTP
- Systems tailored to OLTP (e.g., Neo4j) are generally optimized for *transactional performance*, and tend to guarantee ACID properties

Graph Database: Relationships

- Obviously, graph databases are particulary suited to model situations in which the information is somehow “natively” in the form of a graph

- The real world provide us with a lot of application domains: social networks, recommendation systems, geospatial applications, computer network and data center management, authorization and access control, etc.

Graph Database: Relationships

- The key for the success of graph databases in these contexts is the fact that they provide native means to represent relationships

- Relational databases instead lack relationships: they have to be simulated through the help of foreign keys, thus adding additional development and maintenance overhead, and "discover" them require costly join operations

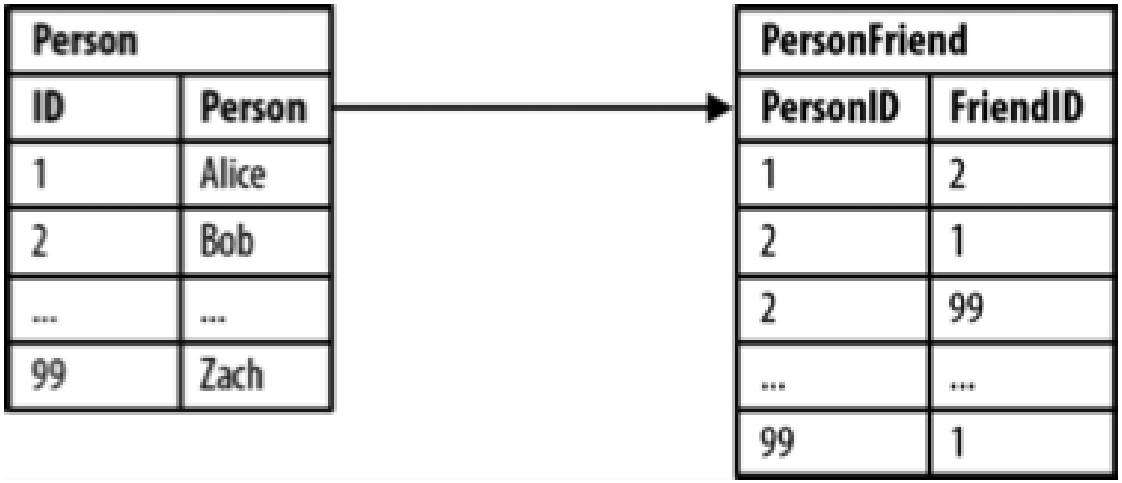
Graph Database: Querying

- Querying = traversing the graph, i.e., following paths/relationships
- Navigational paradigm: online discovery of resources

Graph DBs vs Relational DBs- Example

- Modeling friends and friends-of-friends in a relational database

- Notice that PersonFriend does not have to be considered symmetric: Bob may consider Zach as friend, but the converse does not necessarily hold



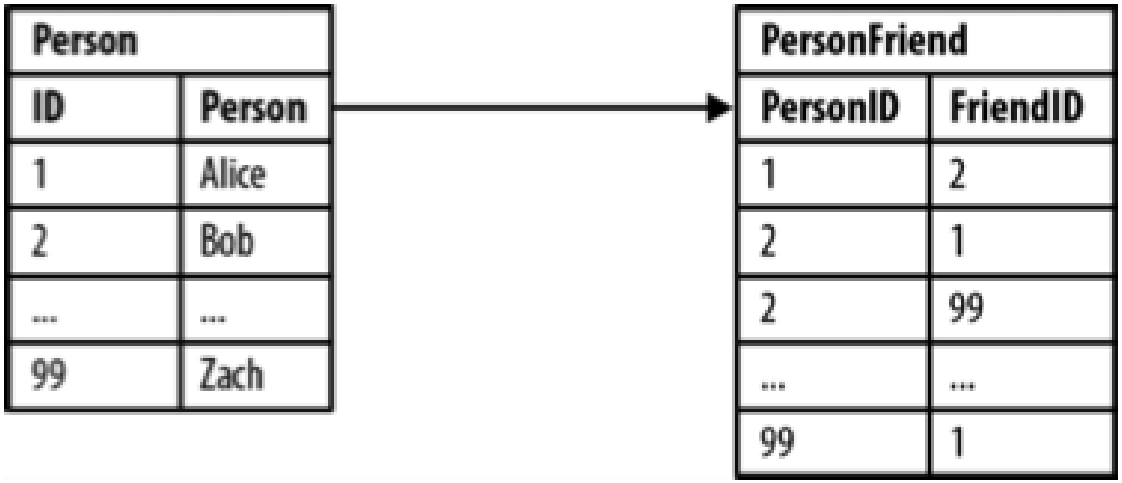
Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Graph DBs vs Relational DBs- Example

- Asking "who are Bob's friends?" (i.e., those that Bob considers as friend) is easy

```
SELECT p1.Person
  FROM Person p1 JOIN PersonFriend ON
       PersonFriend.FriendID = p1.ID JOIN Person p2
  ON
       PersonFriend.PersonID = p2.ID
```



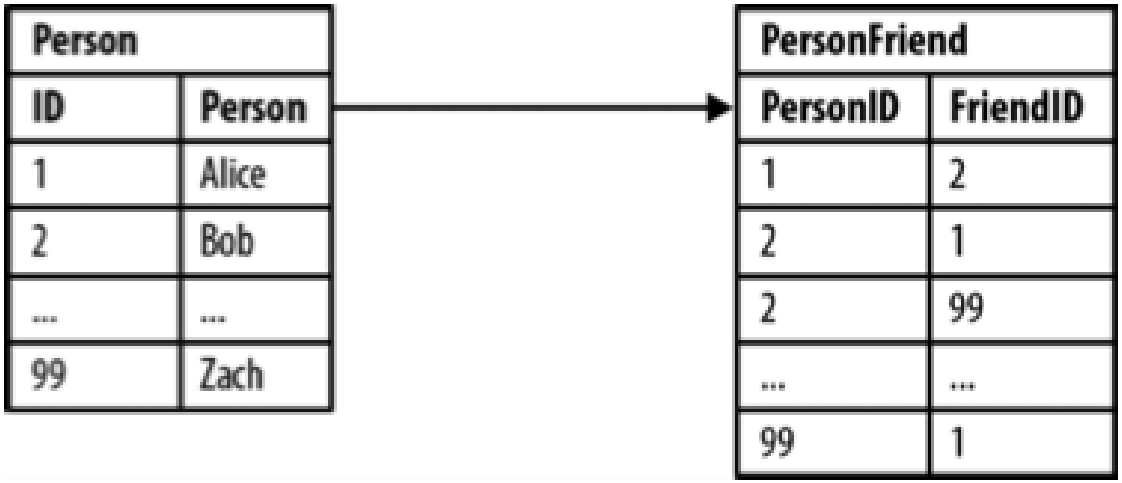
Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Graph DBs vs Relational DBs- Example

- Things become more problematic when we ask, "who are *the Alice's* friends-of-friends?"

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND FROM
PersonFriend pf1 JOIN Person p1 ON
     pf1.PersonID = p1.ID JOIN PersonFriend pf2 ON
     pf2.PersonID = pf1.FriendID JOIN Person p2 ON
     pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```



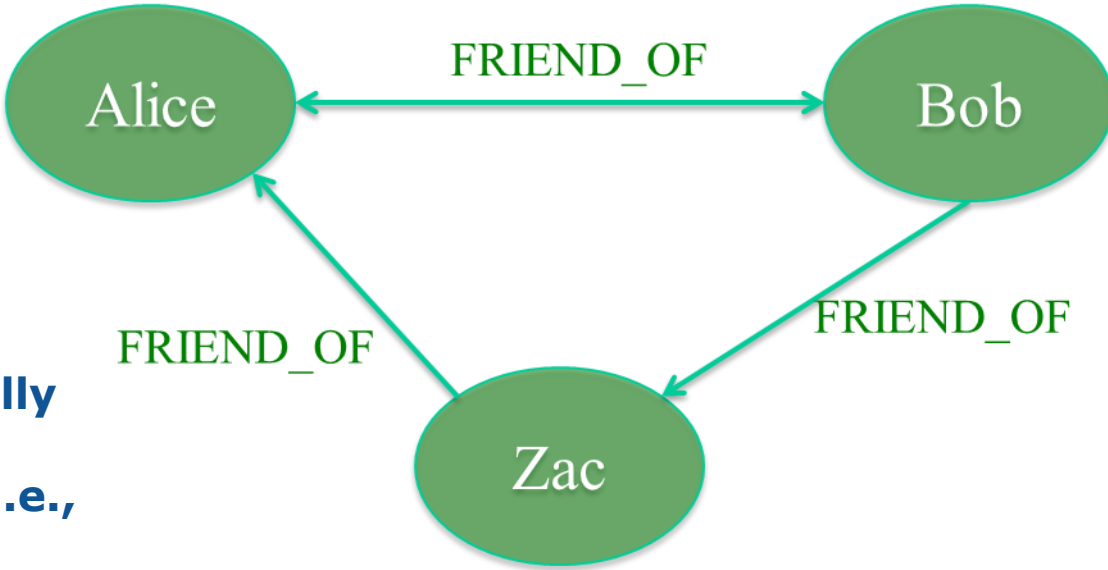
Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Performances highly deteriorate when we go more in depth into the network of friends

Graph DBs vs Relational DBs- Example

- Modeling friends and friends-of-friends in a graph database



Relationships in a graph naturally form paths. Querying means actually traversing the graph, i.e., following paths. Because of the fundamentally path-oriented nature of the data model, the majority of path-based graph database operations are extremely efficient.

Schemaless Databases - 1

- NoSQL databases are schemaless:
  - **A key-value store allows you to store any data you like under a key**
  - **A document database effectively does the same thing, since it makes no restrictions on the structure of the documents you store**
  - **Column-family databases allow you to store any data under any column you like**
  - **Graph databases allow you to freely add new edges and freely add properties to nodes and edges as you wish**

Schemaless Databases - 2

- This has various advantages:
  - **Without a schema binding you, you can easily store whatever you need, and change your data storage as you learn more about your project**
  - **You can easily add new things as you discover them**
  - **A schemaless store also makes it easier to deal with non-uniform data: data where each record has a different set of fields**

Schemaless Databases - 3

- And also some 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 remains ignorant of the schema: it cannot use the schema to support the decision on how to store and retrieve data efficiently.
  - Also, it cannot impose integrity constaints to maintain information coherent