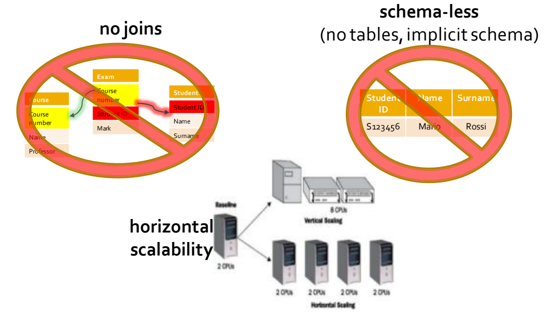
Relational and Non-relational databases for Big Data

Relational vs Non-relational Databases

«NoSQL» birth

- In 1998 Carlo Strozzi's lightweight, open-source relational database that did not expose the standard SQL interface
- In 2009 Johan Oskarsson's (Last.fm) organizes an event to discuss recent advances on non-relational databases. A new, unique, short hashtag to promote the event on Twitter was needed: #NoSQL

NoSQL main features



Comparison

Relational databases	Non-Relational databases
Table-based, each record is a structured row	**Specialized storage solutions**, e.g, document-based, key-value pairs, graph databases, columnar storage
Predefined **schema** for each table, changes allowed but usually blocking (expensive in distributed and live environments)	**Schema-less**, schema-free, schema change is dynamic for each document, suitable for semi-structured or **un-structured data**
Vertically scalable, i.e., typically scaled by increasing the power of the hardware	**Horizontally** scalable, NoSQL databases are scaled by increasing the databases servers in the pool of resources to reduce the load

Comparison

Relational databases	Non-Relational databases
Use **SQL** (Structured Query Language) for defining and manipulating the data, very powerful	**Custom query** languages, focused on collection of documents, graphs, and other specialized data structures
Suitable for **complex queries**, based on data **joins**	**No standard** interfaces to perform complex queries, **no joins**
Suitable for **flat** and structured data storage	Suitable for complex (e.g., **hierarchical**) data, similar to JSON and XML
Examples: MySQL, Oracle, Sqlite, Postgres and Microsoft SQL Server	Examples: MongoDB, BigTable, Redis, Cassandra, HBase and CouchDB

Relational DBMSs

- Pros
  - Work with structured data
  - Support strict ACID transactional consistency
  - Support joins
  - Built-in data integrity
  - Large eco-system
  - Relationships via constraints
  - Limitless indexing
  - Strong SQL
  - OLTP and OLAP
  - Most off-the-shelf applications run on RDBMS

Relational DBMSs

- Cons
  - Do not scale out horizontally (concurrency and data size) – only vertically, unless use sharding
  - Data is normalized, meaning lots of joins, affecting speed
  - Difficulty in working with semi-structured data
  - Schema-on-write

Non-relational/NoSQL DBMSs

- Pros
  - Work with semi-structured data (JSON, XML)
  - Scale out (horizontal scaling – parallel query performance, replication)
  - High concurrency, high volume random reads and writes
  - Massive data stores
  - Schema-free, schema-on-read
  - Support records/documents with different fields
  - High availability
  - Speed, due to not having to join tables

Non-relational/NoSQL DBMSs

- Cons
  - Do not support strict ACID transactional consistency
  - Data is denormalized, requiring mass updates (e.g., product name change)
  - Do not have built-in data integrity (must do in code)
  - No relationship enforcement
  - Limited indexing
  - Weak SQL
  - Slow mass updates
  - Use 10-50 more space (replication, denormalized, documents)
  - Difficulty tracking schema changes over time

Data Models

- (Logical) Data model
  - It is a set of constructs for representing the information
- Storage model
  - How the DBMS stores and manipulates the data internally
- A data model is usually independent of the storage model
  - In practice we need at least some insight to achieve good performances

Data Models

- Data model for relational systems
  - Relational model
    - tables, columns and rows
- Data models for NoSQL systems
  - Aggregate models
    - key-value based model
    - Document based model
    - column-family based model
  - Graph-based models

Relational Model

- The dominant data model of the last decades was the relational data model
- Relational data model
  - It can be represented as a set of tables
  - Each table has rows, with each row representing an object of interest
    - We describe objects through columns
  - A column may refer to another row in the same or different table (relationship)

13

Relational Model

- The relational model takes the information that we want to store and divides it into tables and tuples (rows)
- However, a tuple is a limited data structure
  - It captures a set of values
  - We can't nest one tuple within another to get nested records
  - Nor we can put a list of values or tuple within another

14

Aggregate Models

- Data are modeled as units that have a complex structure
  - A more complex structure than just a set of tuples
  - Complex records with
    - Simple fields
    - Lists
    - Maps
    - Records nested inside other records

15

Aggregate Models

- Aggregate is a term coming from Domain-Driven Design
  - An aggregate is a collection of related objects that we wish to treat as a unit for data manipulation, management, and consistency
- We work with data in terms of aggregates
- We like to update aggregates with atomic operations

16

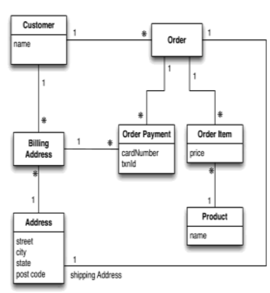
Aggregate Models

- With aggregates we can easier work on a cluster
  - They are "independent" units
- Aggregates are also easier for application programmer to work since solve the impedance mismatch problem of relational databases
  - There is a strict "matching" between the objects used inside programs and the "units/complex records" stored in the databases

17

Example

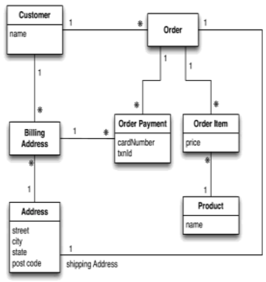
- We are building an e-commerce website
- Stored information
  - Users
  - Products
  - Orders
  - Shipping addresses
  - Billing addresses
  - Payment data



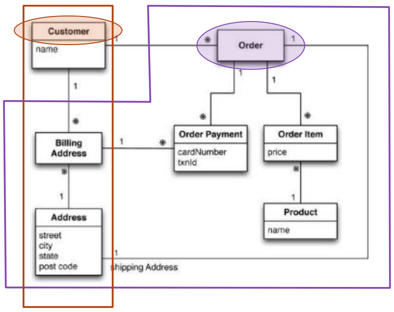
18

Example of Relational Model

- Relational model
  - Everything is normalized
  - No data is repeated in multiple tables
  - We have referential integrity

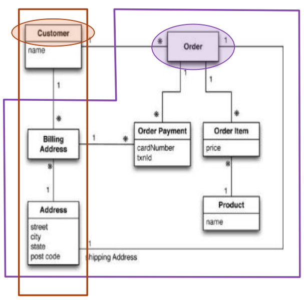


Example of Aggregate Model



Example of Aggregate Model

- We have two aggregates in this example model
  - Customers and
  - Orders



Aggregate implementation

```
// (Single) Customer
{
  "id": 1,
  "name": "Fabio",
  "billingAddresses": [
    {
      "city": "Bari"
    }
  ]
}
```

```
//(Single) Order
{
  "id": 99,
  "customerId": 1,
  "orderItems": [
    {
      "productId": 27,
      "price": 34,
      "productName": "Scala in Action"
    }],
  "shippingAddress": [{"city": "Bari"}],
  "orderPayment": [
    {"ccinfo": "100-432423-545-134",
     "txnId": "afdfsdfsd",
     "billingAddress": [{"city": "Bari" }]
    }]
}
```

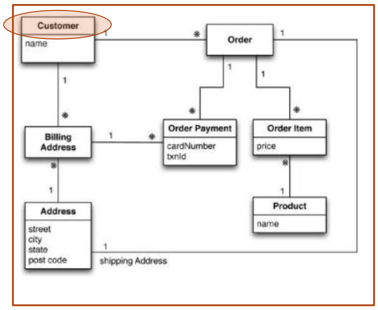

Aggregate implementation

- In the example aggregate model there are two “complex types” of records
  - Customer
    - Each customer record contains the customer profile, including his/her billing addresses
  - Order
    - Each order record contains all the data about one order
- Data are denormalized and some information is replicated

Aggregate implementation

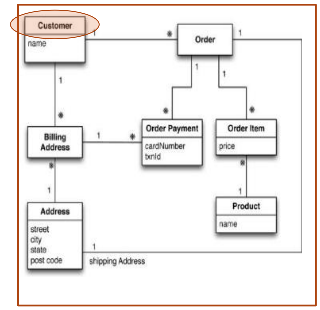
- The solution (data model) is domain-driven
  - The aggregates are related to the expected usage of the data
- In the reported example we suppose to frequently read/write
  - Customer profiles (including shipping addresses)
  - Orders, with all the related information

Another possible aggregation



Another possible aggregation

- We have one aggregate in this model
  - Customers



Another possible aggregation - implementation

```
// (Single) Customer
{
  "id": 1,
  "name": "Fabio",
  "billingAddresses": [
    {
      "city": "Bari"
    }
  ]
  "orders": [
  {
    "id": 99,
    "orderItems": [
      {"productId": 27,
        "price": 34,
        "productName": "Scala in Action"
      }],
     "shippingAddress": [{"city": "Bari"}],
     "orderPayment": [
      {"ccinfo": "100-432423-545-134",
       "txnId": "afdfsdfsd",
       "billingAddress": [{"city": "Bari" }]
      }]
  }]
}
```


Design strategy

- No universal answer for how to draw aggregate boundaries
- It depends entirely on how you tend to manipulate data
  - Accesses on a single order at a time and a single customer at a time
    - First solution
  - Accesses on one customer at a time with all her orders
    - Second solution
- Context-specific
  - Some applications will prefer one or the other

Aggregate Model

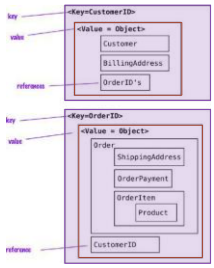
- The focus is on the unit(s) of interaction with the data storage
- Pros:
  - It helps greatly when running on a cluster of nodes
    - The data of each "complex record" will be manipulated together, and thus should the stored on the same node
- Cons:
  - An aggregate structure may help with some data interactions but be an obstacle for others

Solutions-based on Aggregate models

- Key-value model
- Column-family based model
- Document-based model

Key-Value model

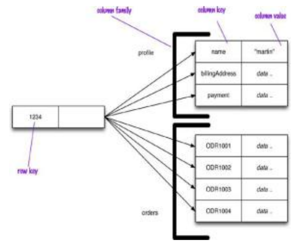
- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model:
  - A set of <key,value> pairs
  - Value: an aggregate instance
- The aggregate is opaque to the database
  - Just a big blob of mostly meaningless bit
- Access to an aggregate
  - Lookup based on its key



31

Column-Family model

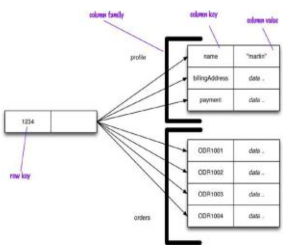
- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model: a two-level map structure:
  - A set of <row-key, aggregate> pairs
  - Each aggregate is a group of pairs <column-key,value>



32

Column-Family model

- Columns can be organized in families
  - Columns of the same family are usually accessed together
- Access to an aggregate
  - Accessing the row as a whole
  - Picking out particular columns (of the same family)



33

Properties of Column-Family model

- Operations also allow picking out a particular column
  - get('1234', 'name')
- Each column
  - Has to be part of a single column family
  - Acts as unit for access
- You can add any column to any row, and rows can have very different columns
- You can model a list of items by making each item a separate column

Properties of Column-Family model

- Two ways to look at data
  - Row-oriented
    - Each row is an aggregate
    - Column families represent useful chunks of data within that aggregate
  - Column-oriented
    - Each column family defines a record type
    - Row as the join of records in all column families

Document-based model

- Strongly aggregate-oriented
  - Lots of aggregates
  - Each aggregate has a key
- Data model:
  - A set of <key,document> pairs
  - Document: an aggregate instance
- Structure of the aggregate visible
  - Limits on what we can place in it
- Access to an aggregate
  - Queries based on the fields in the aggregate

```
# 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"}
}
```

36

Key-Value vs Document-based

- Key-value model
  - A key plus a big blob of mostly meaningless bits
  - We can store whatever we like in the aggregate
  - We can only access an aggregate by lookup based on its key
- Document-based model
  - A key plus a structured aggregate
  - More flexibility in access
    - We can submit queries to the database based on the fields in the aggregate
    - We can retrieve part of the aggregate rather than the whole thing
  - Indexes based on the contents of the aggregate

37

Relationships

- Relationship between different aggregates
  - Put the ID of one aggregate within the data of the other
- Join: **write a program** that uses the ID to link data
  - The database is ignorant of the relationships in the data

38

Key Points

- An aggregate is a collection of data that we interact with as a unit
- Aggregates form the boundaries for ACID operations with the database

39

Key Points

- Aggregates make it easier for the database to manage data storage over clusters
  - Aggregate-oriented databases work best when most data interaction is done with the same aggregate
  - Aggregate-ignorant databases are better when interactions use data organized in many different formations
- Key-value, document, and column-family databases can all be seen as forms of aggregate-oriented database

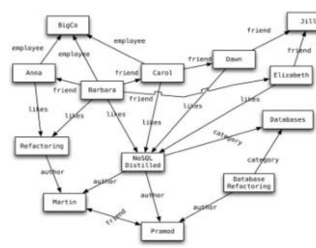
40

Graph Databases

- Graph databases are motivated by a different frustration with relational databases
  - Complex relationships require complex join
- Goal
  - Capture data consisting of complex relationships
  - Data naturally modeled as graphs
  - Examples
    - Social networks, Web data, product preferences

41

A graph database



Query: "find the books in the Database category that are written by someone whom a friend of mine likes."

Data model of graph databases

- Basic characteristic
  - Nodes are connected by edges (also called arcs)
- Beyond this
  - A lot of variation in data models
    - Neo4J stores Java objects as nodes and edges in a schemaless fashion
    - InfiniteGraph stores Java objects, which are subclasses of built-in types, as nodes and edges.
    - FlockDB is simply nodes and edges with no mechanism for additional attributes

43

Data model of graph databases

- Queries
  - Navigation through the network of edges
  - You do need a starting place
  - Nodes can be indexed by an attribute such as ID

44

Graph vs Relational databases

- Relational databases
  - Implement relationships using foreign keys
  - Joins require to navigate around and can get quite expensive
- Graph databases
  - Make traversal along the relationships very cheap
  - Performance is better for highly connected data
  - Shift most of the work from query time to insert time
  - Good when querying performance is more important than insert speed

Graph vs Aggregate-oriented databases

- Very different data models
- Aggregate-oriented databases
  - Distributed across clusters
  - Simple query languages
  - No ACID guarantees
- Graph databases
  - More likely to run on a single server
  - Graph-based query languages
  - Transactions maintain consistency over multiple nodes and edges

46

Some NoSQL databases

- Key-value databases
  - Redis, Riak, Memcached, ..
- Column-family databases
  - Cassandra, HBase, Hypertable, Amazon DynamoDB, ..
- Document databases
  - MongoDB, CouchDB, RavenDB, ..
- Graph databases
  - Neo4J, Infinite Graph, OrientDB, ..

47