

# Database 2 course notes

Vittorio Romeo

# Contents

<b>1</b>	<b>DBMS types</b>	<b>2</b>
1.1	Relational DBMSs . . . . .	2
1.1.1	Disadvantages . . . . .	2
1.2	Object-oriented DBMSs . . . . .	2
1.2.1	Disadvantages . . . . .	3
1.2.2	Advantages . . . . .	3
1.3	Object-relational DBMSs . . . . .	3
<b>2</b>	<b>Distributed systems</b>	<b>4</b>
2.1	General information . . . . .	4
2.1.1	Transparency . . . . .	4
2.1.2	Openness . . . . .	5
2.1.3	Scalability . . . . .	5
2.2	Types . . . . .	5
2.2.1	Distributed Computing Systems . . . . .	5
2.2.2	Distributed Information Systems . . . . .	6
2.2.3	Distributed Pervasive Systems . . . . .	6
2.3	Architectures . . . . .	6
2.3.1	Styles and models . . . . .	6
2.3.2	Centralized architectures . . . . .	7
2.3.3	Decentralized architectures . . . . .	7
2.3.4	Architectures versus middleware . . . . .	8
2.3.5	Self-managing distributed systems . . . . .	8
<b>3</b>	<b>Distributed architectures</b>	<b>9</b>
3.1	Distributed DBMSs . . . . .	9
3.1.1	Basics and data fragmentation . . . . .	9
3.1.2	Transparency levels . . . . .	9
3.1.3	Transaction classification . . . . .	9
3.2	Distributed DBMSs technology . . . . .	10
3.2.1	Consistency and persistency . . . . .	10
3.2.2	Optimization . . . . .	10
3.2.3	Concurrency control . . . . .	10
3.3	Distributed transaction atomicity . . . . .	11
3.3.1	2-phase commit protocol . . . . .	11

# Chapter 1

## DBMS types

### 1.1 Relational DBMSs

- Formally introduced by **Codd** in 1970.
- ANSI standard: **SQL**.
- Composed of many relations in form of **2D tables**, containing **tuples**.
  - **Logical view**: data organized in tables.
  - **Internal view**: stored data.
  - Rows (*tuples*) are **records**.
  - Columns (*fields*) are **attributes**.
    - \* They have specific **data types**.
- **Constraints** are used to restrict stored data.
- **SQL** is divided in **DDL** and **DML**.

#### 1.1.1 Disadvantages

- Lack of flexibility: all processing is based on values in fields of records.
- Inability to handle complex types and complex interrelationships.

### 1.2 Object-oriented DBMSs

- Integrated with an OOP language.
- Supports:
  - Complex data types.
  - Type inheritance.

- Object behavior.
- Objects have an **OID** (*object identifier*).
- **ADTs** (*abstract data types*) are used for **encapsulation**.
- OODBMSs were standardized by **ODMG** (*Object Data Management Group*).
  - Object model, **ODL**, **OQL** and OOP language bindings.
- **OQL** resembles **SQL**, with additional features (*object identity, complex types, inheritance, polymorphism, ...*).

### 1.2.1 Disadvantages

- Poor performance. Queries are hard to optimize.
- Poor scalability.
- Problematic change of schema.
- Dependence from OOP language.

### 1.2.2 Advantages

- Composite objects and relations.
- Easily manageable class hierarchies.
- Dynamic data model
- No primary key management.

## 1.3 Object-relational DBMSs

- Hybrid solution, expected to perform well.
- Features:
  - Base datatype extension (*inheritance*).
  - Complex objects.
  - Rule systems.

# Chapter 2

## Distributed systems

### 2.1 General information

- A distributed system is a **software** that makes a **collection of independent machines** appear as a **single coherent system**.
  - Achieved thanks to a **middleware**.
- Goals:
  - Making resource available.
  - Distribution **transparency**.
  - **Openness** and **scalability**.

#### 2.1.1 Transparency

Type	Description
Access	Hides data access
Location	Hides data locality
Migration	Hides ability of a system to change object location
Relocation	Hides system ability to move object bound to client
Replication	Hides object replication
Concurrency	Hides coordination between objects
Failure	Hides failure and recovery

- Hard to fully achieve.
  - Users may live in different continents.
  - Networks are unreliable.
  - Full transparency is costly.

### 2.1.2 Openness

- Conformance to well-defined interfaces.
- Portability and interoperability.
- Heterogeneity of underlying environments.
- Requires support for **policies**.
- Provides **mechanisms** to fulfill policies.

### 2.1.3 Scalability

- **Size**: number of users/processes.
- **Geographical**: maximum distance between nodes.
- **Administrative**: number of administrative domains.
- Techniques to achieve scalability:
  - Hide communication latencies.
    - \* Use **asynchronous** communication.
    - \* Use **separate response handlers**.
  - Distribution.
    - \* Decentralized **DNS** and information systems.
    - \* Try to compute as much as possible on clients.
  - Replication/caching.
- Issue: **inconsistency** and **global synchronization**.

## 2.2 Types

### 2.2.1 Distributed Computing Systems

- **HPC** (*high-performance computing*).
- Cluster computing:
  - **Homogeneous** LAN-connected machines.
    - \* Master node + compute nodes.
- Grid computing:
  - **Heterogeneous** WAN-connected machines.
  - Usually divided in **virtual organizations**.

## 2.2.2 Distributed Information Systems

- **Transaction-based systems.**
  - **Atomicity.**
  - **Consistency.**
  - **Isolation:** no interference between concurrent transaction.
  - **Durability:** changes are permanent.
- **TP Monitors** (*transaction processing monitors*) coordinate execution of a distributed transaction.
  - **Communication middleware** is required to separate applications from databases.
    - \* **RPC** (*remote procedure call*).
    - \* **MOM** (*message-oriented middleware*).

## 2.2.3 Distributed Pervasive Systems

- Small nodes, often **mobile** or **embedded**.
- Requirements:
  - **Contextual change.**
  - **Ad-hoc composition.**
  - **Sharing by default.**
- Examples:
  - Home systems.
  - Electronic health systems.
  - Sensor networks.

## 2.3 Architectures

### 2.3.1 Styles and models

- Architectural styles:
  - **Layered:** used for client-server systems.
  - **Object-based:** used for distributed systems.
- Decoupling models:
  - **Publish/subscribe:** uses **event bus**, decoupled in space.

- **Shared dataspace**: used shared persistent data space, decoupled both in space and time.

### 2.3.2 Centralized architectures

- Client-server.
- Three-layered view:
  - User-interface layer.
  - Processing layer.
  - Data layer.
- Multi-tiered architecture:
  - Single-tiered: dumb terminal/mainframe.
  - Two-tiered: client-server.
  - Three-tiered: each layer on separate machine.

### 2.3.3 Decentralized architectures

- **P2P** (*peer-to-peer*):
  - P2P architectures are **overlay networks**: application-level multicasting.
  - **Structured**: nodes follow a specific data structure.
    - \* Example: ring, kd-tree.
  - **Unstructured**: nodes choose random neighbors.
    - \* Example: random graph.
      - Each node has a **partial view** of the network which is shared with random nodes selected periodically, along with data.
  - **Hybrid**: some nodes are special (*and structured*).
- Topology management:
  - 2 layers: structured and random.
    - \* Promote some nodes depending on their services.
    - \* Torus construction: create  $N * N$  grid, keep only **nearest neighbors** via distance formula.
    - \* **Superpeers**: few specific nodes.
      - Examples: indexing, coordination, connection setup.
- Hybrid architectures (*P2P + client-server*):
  - **CDNs**: edge-server architectures.



- **BitTorrent**: tracker and peers.

### 2.3.4 Architectures versus middleware

- Sometimes the middleware needs to **dyamically adapt its behavior** to distributed application/systems.
  - **Interceptors** can be used.
  - **Adaptive middleware**:
    - \* Separation of concerns.
    - \* Computational reflection (*self runtime inspection*).
    - \* Component-based design.

### 2.3.5 Self-managing distributed systems

- Self-*x* operations:
  - Configuration.
  - Management.
  - Healing.
  - Optimization.
- **Feedback control model**.
  - Example: globule (*collaborative CDN driven by cost model*).

# Chapter 3

## Distributed architectures

### 3.1 Distributed DBMSs

#### 3.1.1 Basics and data fragmentation

- Based on **autonomy** and **cooperation**.
- Data **fragmentation** and **allocation**:
  - A relation  $R$  is split in  $R_i$  fragments.
  - **Horizontal** fragmentation:
    - \*  $R_i$ : set of tuples with same schema as  $R$ .
    - \* Like the **where** SQL clause.
  - **Vertical** fragmentation:
    - \*  $R_i$ : set of tuples with subschema of  $R$ .
    - \* Like the **select** SQL clause.

#### 3.1.2 Transparency levels

- **Fragmentation** transparency: independence of a query from data fragmentation and allocation.
- **Allocation** transparency: fragment structure must be specified in a query, but not location.
- **Language** transparency: both fragment structure and location have to be specified in a query.

#### 3.1.3 Transaction classification

- **Remote request**: readonly (*select*) transactions towards a **single** DBMS.

- **Remote transaction:** general transactions towards a **single** DBMS.
- **Distributed transaction:** towards multiple DBMSs, but every SQL operation targets a single DBMS.
- **Distributed request:** arbitrary transaction, language-level transparency.

## 3.2 Distributed DBMSs technology

### 3.2.1 Consistency and persistency

- **Consistency:** does not depend on data distribution. Constraints are only properties local to a specific DBMS. This is a limitation of DBMSs.
- **Persistency:** does not depend on data distribution. Every system guarantees persistency thanks to dumps and backups.

### 3.2.2 Optimization

- **Global optimization** is performed through a cost analysis.
  - A tree of possible alternatives is examined.
  - **IO**, **CPU** and **bandwidth** costs are taken into account.

### 3.2.3 Concurrency control

- *Problem:* two transactions  $t_1$  and  $t_2$  can be composed of subtransactions whose execution is in conflict.
  - The transactions are **locally serializable**.
  - The transactions are **not globally serializable**.
- **Global serializability:** two transactions are globally serializable if  $\exists S$  (*serial schedule*) that is equivalent to every local schedule  $S_i$ .
  - For every node  $i$ , the projection  $S[i]$  of  $S$  needs to be equivalent to  $S_i$
  - This property can be fulfilled using **2-phase locking** or **timestamping**.

#### 3.2.3.1 Lamport's method for timestamping

- Every transaction needs a timestamp of the time instant where it needs to be synchronized with other transactions.
- A timestamp is composed by two numbers: **node ID** and **event ID**.
- Nodes have a local counter that helps ordering transactions.

### 3.2.3.2 Distributed deadlock detection

- Two subtransactions may be waiting for one another in the same or in different DBMSs.
- A **waiting sequence** can be built for every transaction.
- Algorithm:
  1. DBMSs share their waiting sequences.
  2. Waiting sequences are composed in a **local waiting graph**.
  3. Deadlocks are detected locally and solved by aborting transactions.
  4. Updated waiting sequences are sent to other DBMSs.

## 3.3 Distributed transaction atomicity

### 3.3.1 2-phase commit protocol

- Conceptually similar to marriage.
- Servers are called **RM**s (*resource managers*).
- A coordinator is called **TM** (*transaction manager*).
- Both RMs and the TM have **local logs**.
- TM log records:
  - **prepare**: contains RMs identities.
  - **global commit/abort**: atomic and persistent decision regarding **the entire transaction**.
  - **complete**: conclusion of the protocol.
- RM log records:
  - **ready**: signals availability of the node.
- Algorithm (*ideal situation*):
  - Phase one (*preparation*):
    1. TM sends **prepare**, sets a **timeout** for RM responses.
    2. RMs wait for **prepare** messages. On arrival, they send **ready**. If an RM is in a bad state, **not-ready** is sent instead, terminating the protocol (*global abort*).
    3. TM collects RM messages. On success, sends **global commit**.
  - Phase two:
    1. TM sends global decision, setting a **timeout**.
    2. Ready RMs wait for the decision. On arrival, they either log **commit** or **abort**, and send an **ack** to the TM.

3. TM collects all **ack** messages. If all of them arrived, **complete** is set. If an **ack** is missing, a new **timeout** is set and transmissions are repeated.
- The period between **ready** and **commit/abort** is called **uncertainty interval** - the protocol tries to minimize its length.

### 3.3.1.1 Recovery protocols

- RM drops:
  - If last record was **abort**, actions will be undone.
  - If last record was **commit**, actions will be repeated.
  - If last record was **ready**, we are in a **doubtful situation**.
    - \* Information needs to be requested from TM.
- TM drops:
  - If last record as **prepare**, some RMs may be locked.
    - \* **global abort** will be sent, or the first phase will be repeated.
  - If last record was **global commit/abort**, the second phase needs to be repeated.
  - If last record was **complete**, everything is fine.
- Message loss: handled by timeouts, which cause a **global abort** in the first phase, or a retransmission in the second phase.

### 3.3.1.2 Optimizations

- **Presumed abort protocol**: if in doubt during a RM recovery, and TM has no information, **abort** is returned.
  - Some synchronous record writes can be avoided.
- **Read-only optimization**: if an RM only needs to read, it will not influence the transaction's result - it can be ignored during second phase.
- **TODO**: other commits, replication, cooperation