

# Database 2 course notes

Vittorio Romeo

# Contents

<b>1</b>	<b>DBMS types</b>	<b>3</b>
1.1	Relational DBMSs . . . . .	3
1.1.1	Disadvantages . . . . .	3
1.2	Object-oriented DBMSs . . . . .	3
1.2.1	Disadvantages . . . . .	4
1.2.2	Advantages . . . . .	4
1.3	Object-relational DBMSs . . . . .	4
<b>2</b>	<b>Distributed systems</b>	<b>5</b>
2.1	General information . . . . .	5
2.1.1	Transparency . . . . .	5
2.1.2	Openness . . . . .	6
2.1.3	Scalability . . . . .	6
2.2	Types . . . . .	6
2.2.1	Distributed Computing Systems . . . . .	6
2.2.2	Distributed Information Systems . . . . .	7
2.2.3	Distributed Pervasive Systems . . . . .	7
2.3	Architectures . . . . .	7
2.3.1	Styles and models . . . . .	7
2.3.2	Centralized architectures . . . . .	8
2.3.3	Decentralized architectures . . . . .	8
2.3.4	Architectures versus middleware . . . . .	9
2.3.5	Self-managing distributed systems . . . . .	9
<b>3</b>	<b>Distributed architectures</b>	<b>10</b>
3.1	Distributed DBMSs . . . . .	10
3.1.1	Basics and data fragmentation . . . . .	10
3.1.2	Transparency levels . . . . .	10
3.1.3	Transaction classification . . . . .	10
3.2	Distributed DBMSs technology . . . . .	11
3.2.1	Consistency and persistency . . . . .	11
3.2.2	Optimization . . . . .	11
3.2.3	Concurrency control . . . . .	11
3.2.3.1	Lamport's method for timestamping . . . . .	11
3.2.3.2	Distributed deadlock detection . . . . .	12

3.3	Distributed transaction atomicity . . . . .	12
3.3.1	2-phase commit protocol . . . . .	12
3.3.1.1	Recovery protocols . . . . .	13
3.3.1.2	Optimizations . . . . .	13
3.3.2	Other commit protocols . . . . .	13
3.3.2.1	4-phase commit protocol . . . . .	14
3.3.2.2	3-phase commit protocol . . . . .	14
3.3.2.3	Paxos commit . . . . .	14
3.3.2.4	X-Open DTP . . . . .	14
3.4	DBMS replication . . . . .	15
<b>4</b>	<b>Parallel DBMSs and cloud architectures</b>	<b>16</b>
4.1	Parallelism . . . . .	16
4.1.1	Relationship with data fragmentation . . . . .	16
4.1.2	Speed-up and scale-up . . . . .	16
4.2	Cloud computing architectures . . . . .	17
4.2.1	Classification . . . . .	17
4.2.1.1	Features . . . . .	17
4.2.1.2	Types . . . . .	17
4.2.1.3	Service models . . . . .	17
4.2.2	Hadoop and MapReduce . . . . .	17
4.2.3	Apache Pig and Pig Latin . . . . .	18
4.2.4	Apache Hive and Hive QL . . . . .	18
<b>5</b>	<b>Cloud computing</b>	<b>19</b>
5.1	Definitions . . . . .	19
5.2	Service models . . . . .	20
5.2.1	Layers . . . . .	20
5.2.2	IaaS . . . . .	20
5.2.2.1	Virtualization . . . . .	20
5.2.3	PaaS . . . . .	21
5.2.4	SaaS . . . . .	21
5.2.4.1	Maturity model . . . . .	21
<b>6</b>	<b>SQL vs NoSQL</b>	<b>22</b>
6.1	SQL characteristics . . . . .	22
6.2	Big data . . . . .	22
6.2.1	3-layer processing architecture . . . . .	22
6.2.2	Lambda architecture . . . . .	23
6.3	NoSQL . . . . .	23

# 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 **dynamically 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

## 3.3.2 Other commit protocols

- The biggest issue with the 2-phase protocol is that an RM can become stuck if the TM drops.
  - The following protocols don't have this issue but are less performant.

### 3.3.2.1 4-phase commit protocol

- The TM process can be replicated by a **backup process** on a different node.
  - On every phase, the TM first communicates with the backup, then with the RMs.

### 3.3.2.2 3-phase commit protocol

- After receiving **ready** from every RM, the TM has an additional **pre-commit** state.
  - If the TM drops during that state, any RM can become the TM, because every RM has to be **ready**.
- Unusable in practice due to widened uncertainty interval and atomicity issues in case of network partitioning.

### 3.3.2.3 Paxos commit

- More general goal: have nodes “agree” on a specific value in case of malfunction.
- Three node categories:
  - Proponent.
  - Acceptor.
  - Receiver.
- Three phases:
  1. Election of a coordinator.
  2. Acceptors agree on a value.
  3. The value is propagated to receivers.
- Algorithm:
  1. The coordinator sends  $n$  **prepare** messages to participants.
  2. Every participant sends **ready** to coordinator and to  $f$  acceptors.
  3. Every acceptor sends its state using  $f$  messages.
  4. Coordinator and acceptors are  $f + 1$  nodes that know the state of the transaction.  
Any malfunction in  $f$  is not a problem.

### 3.3.2.4 X-Open DTP

- Guarantees interoperability of transactions on different DBMSs.
- Two main interfaces:
  1. **TM-interface**: between client and TM.
    - `tm_XXX` functions.

2. **XA-interface**: between TM and RM.

- Database vendors must guarantee XA-interface availability.
- `xa_***` functions.
- Features:
  - RMs are passive. All control is in TM, which uses RPCs to enable RM functions.
  - Uses 2-phase commit with aforementioned optimizations.
  - **Heuristical decisions** are taken, which can harm atomicity (*notifying clients*).

## 3.4 DBMS replication

- A **data replicator** handles replication and **synchronization** between copies.
  - Copies are updated asynchronously (*no commit protocols*).
- Replication data can be **batched** and reconciled with the copies all at once.
- **Multidatabase systems**: tree hierarchies of **dispatchers** and multiple DBs behind a single interface.



# Chapter 4

## Parallel DBMSs and cloud architectures

### 4.1 Parallelism

- Ideally speeds up computation by a factor of  $1/n$ .
- Two types:
  1. **Inter-query**: different queries ran in parallel.
  2. **Intra-query**: parts of the same query (*subqueries*) ran in parallel.

#### 4.1.1 Relationship with data fragmentation

- Data fragments are in different locations, which can be associated to different processors.

#### 4.1.2 Speed-up and scale-up

- **Speed-up**: only related to inter-query parallelism. Measures *tps* as the number of processors grows.
- **Scale-up**: related to both parallelism types. Measures  $\frac{cost}{tps}$  as the number of processors grows.

## 4.2 Cloud computing architectures

### 4.2.1 Classification

#### 4.2.1.1 Features

- **On-demand self-service:** architecture elements can be defined depending on current needs through web interfaces.
- **Remote access.**
- **Service measuring:** architectural resources are rented using costs depending on use.
- **Elasticity.**
- **Resource sharing.**

#### 4.2.1.2 Types

- **Private cloud:** of an organization/institution.
- **Community cloud:** of a community of organizations/institutions.
- **Public cloud:** like AWS or Azure.
- **Hybrid cloud:** private cloud that use public services when needed.

#### 4.2.1.3 Service models

- **SaaS:** clients rent finished applications.
- **PaaS:** clients rent hardware resources and base software.
- **IaaS:** clients rent only hardware resources.

### 4.2.2 Hadoop and MapReduce

- **HDFS:** distributed filesystem developed in Java.
  - Uses TCP/IP for communication.
  - Files are fragmented in separate nodes and are replicated.
  - The main node is called **NameNode**, others are called **workers**.
- **MapReduce:** parallel computation model.
  - **Jobs** are handled by a **job tracker**.
  - Jobs assign **tasks**, which are handled by a **task tracker**.

### 4.2.3 Apache Pig and Pig Latin

- Query system based on Hadoop.
- Data model is similar to OODBMSs, but does not support inheritance.
  - Data is organized in relationships.
  - Relations can contain duplicated elements (*tuple bags*).
  - There is no explicit primary key.
- Example query: `FOREACH table GENERATE attribute0 attribute1;`

### 4.2.4 Apache Hive and Hive QL

- Similar to Pig, but closer to SQL.

# Chapter 5

## Cloud computing

TODO: merge previous chapter into this one or viceversa?

TODO: add cloud federations to cloud models

### 5.1 Definitions

- **Cloud computing** describes a class of network-based computing:
  - A collection/group of networked hardware, software and infrastructure (*platform*).
  - Uses the Internet for communication/transport, providing hardware and software services to client.
- The complexity of the platforms is hidden behind simple **APIs**.
- Characteristics:
  - **Remotely hosted**.
  - **Ubiquitous**: services/data available from anywhere.
  - **Commodified**: pay for what you want/need.
  - Common characteristics:
    - \* Massive scale.
    - \* Resilient computing.
    - \* Homogeneity.
    - \* Geographic distribution.
    - \* Virtualization.
    - \* Service-orientation.
    - \* Low-cost.
    - \* Security.

- Essential characteristics:
  - \* On-demand self-service.
  - \* Broad network access.
  - \* Elasticity.
  - \* Resource pooling.
  - \* Measured service.

## 5.2 Service models

### 5.2.1 Layers

- From application-focused to infrastructure-focused:
  1. Services.
  2. Application.
  3. Development.
  4. Platform.
  5. Storage.
  6. Hosting.

### 5.2.2 IaaS

- Provides hardware.

#### 5.2.2.1 Virtualization

- The basis of IaaS.
- **Virtual workspaces:** abstraction over the execution environment.
  - Has specific resource quota and software configuration.
- Implemented on **VMs** (*virtual machines*).
  - Abstraction of the physical host.
  - Advantages:
    - \* OS flexibility. Easier deployment.
    - \* Versioning/backups/migrations.
- A **VMM** (*virtual machine monitor, or hypervisor*) is used to manage multiple VMs on a single machine.

### 5.2.3 PaaS

- Deploys user-created applications.
- Highly-scalable architecture.

### 5.2.4 SaaS

- Provides applications.
- *Examples:* Facebook apps, Google apps.

#### 5.2.4.1 Maturity model

- **Level 1:** ad-hoc/custom. One instance per customer.
- **Level 2:** configurable per customer.
- **Level 3:** configurable and multi-tenant-efficient.
- **Level 4:** scalable (*uses load balancer*) level 3.

# Chapter 6

## SQL vs NoSQL

### 6.1 SQL characteristics

- Data is stored in columns and tables.
- Relationships represented by data.
- DML and DDL.
- Transactions.
  - ACID properties.
- Abstraction from physical layer.
  - Declarative language.
  - Query optimization engine.

### 6.2 Big data

- Extremely large datasets.
- Challenges:
  - Analysis, capture, searching, storage, transfer, visualization, querying, security.
- Characteristics: **volume**, **velocity** and **variety**.
- Big data **analytics**: capture and analysis processes aiming to find patterns and correlations in huge heterogeneous datasets.

#### 6.2.1 3-layer processing architecture

1. Online processing:
  - Real-time data capture/processing.

- Deals with **velocity**:
  - Algorithms need to be simple and fast.
- 2. Nearline processing:
  - Database-oriented.
  - Handles data storage and some processing (*slightly more complex than online processing*).
- 3. Offline processing:
  - Batch heavy-processing of data.

### 6.2.2 Lambda architecture

- Principles:
  1. **Human fault-tolerance**: data needs to survive human errors and hardware faults.
  2. **Data immutability**: no updates/deletes.
  3. **Recomputation**: recomputing previous results must always be possible.
- Levels:
  1. **Batch layer**: stores the master dataset and computes **views** (*pre-computing*) using MapReduce algorithms.
  2. **Speed layer**: computes **real-time** views only with new data, not total data. Uses an **incremental model**.
  3. **Serving layer**: output of the batch layer. Handles view indexing and provides views to the query system.
    - The query system uses both batch and speed views.

## 6.3 NoSQL

- Class of non-relational data storage systems.
  - Types:
    - \* Document store. *Example*: **MongoDB**.
    - \* Column based. *Example*: **Cassandra**.
    - \* Graph. *Example*: **Neo4j**.
    - \* Key-value.
- Usually do not require fixed schema and do not use joins.
  - Can be distributed.



- One or more ACID properties are relaxed.
  - **BASE** transactions:
    - \* Basically available.
    - \* Soft state.
    - \* Eventually consistent.
  - Brewer’s **CAP** theorem: a distributed system can support only two of the following:
    - \* Consistency.
    - \* Availability.
    - \* Partition tolerance.
- Compared to SQL: higher scalability and flexibility.