Database 2 course notes

Vittorio Romeo

Contents

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 trasparency 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)*.