Database 2 course notes

Vittorio Romeo
http://vittorioromeo.info

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)*.

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 sistem 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** coss 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 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 diffrerent 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 **RMs** *(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.

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_xxx` 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 atomiticy *(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 $\frac{cost}{tps}$ aas the number of processors grows.

4.2 Cloud computing architectures

- **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**.

4.2.1 Classification

4.2.1.1 Characteristics

- **Remotely hosted**.
- **Ubiquitous**: services/data available from anywhere.
- **Commodified**: pay for what you want/need.
- Massive scale.
- Resilient computing.
- Homogeneity.
- Geographic distribution.
- Virtualization.
- Service-orientation.
- Low-cost.
- Security.

4.2.1.2 Features

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

4.2.1.3 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.
- **Cloud federations**.

4.2.2 Service models

4.2.2.1 Layers

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

4.2.2.2 IaaS

- **IaaS**: clients rent only hardware resources.

4.2.2.2.1 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.

4.2.2.3 PaaS

- **PaaS**: clients rent hardware resources and base software.
- Deploys user-created applications.
- Highly-scalable architecture.

4.2.2.4 SaaS

- **SaaS**: clients rent finished applications.
- Provides applications.
- *Examples:* Facebook apps, Google apps.

4.2.2.4.1 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.

4.2.3 Google ecosystem

4.2.3.1 GFS

- Distributed file system.
- Two node types:
  - **Chunk**: nodes that store files.
    * Every file is 64MB.
    * Every chunk is assigned to a 64bit partition.
    * Chunks are periodically replicated.
  - **Master**: manage chunk metadata, 64bit partition tables, chunk copies locations.

4.2.3.2 MapReduce

- Like Hadoop MapReduce.

4.2.3.3 BigTable

- A **key-value** big data system based on GFS.

4.2.3.4 Chubby

- Manages locks and agreements between nodes.
- A **cell** is a small set of servers *(usually 5)* called replicas.
  - Replicas use the Paxos protocol to elect a master.
- Similar to Apache Zookeeper.

4.2.4 Hadoop ecosystem and MapReduce

- **Apache Hadoop** is a suite of open-source components which serve as the building blocks of large distributed systems.
  - Focus on gradual, horizontal scaling.

4.2.4.1 ZooKeeper

- **ZooKeeper** is a distributed coordination service which is used when nodes in a distributed system need a single source of truth.
  - Similar to **Google Chubby**.
- Implemented as a single moveable master, with n coordinated nodes.
  - A majority $(n/2+1)$ must agree on a write.
  - Reads can be answered by any node.

4.2.4.2 HDFS

- **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**.

4.2.4.3 MapReduce

- **MapReduce**: parallel computation model.
  - **Jobs** are handled by a **job tracker**.
  - Jobs assign **tasks**, which are handled by a **task tracker**.

4.2.4.4 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;`.

4.2.4.5 Apache Hive and Hive QL

- Similar to Pig, but closer to SQL.

Chapter 5

SQL vs NoSQL

5.1 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.

5.2 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.

5.2.1 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-procesing of data.

5.2.2 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.

Chapter 6

Oracle and PL/SQL

- **Oracle Database** is an object-relational database management system *(ORDBMS)*.
- **PL/SQL** is also known as **Embedded SQL**.
- More powerful than pure **SQL**:
  - Has **iteration**, **branching**, **cursors**, **blocks**, **stored procedures**, and more.

6.1 Basic structure

```
DECLARE
    -- ...
BEGIN
    -- ...
EXCEPTION
    -- ...
END;
```

6.1.1 Server output

- Execute `set serveroutput on` before running.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello world!');
END;
```

6.1.2 Example

```
DECLARE
    v_id INTEGER;
    v_empno NUMBER;
```

```
BEGIN
    v_id := 1234567;
    SELECT EMPNO
    INTO v_empno
    FROM EMP
    WHERE empno = v_id;
    DBMS_OUTPUT.PUT_LINE('Value is ' || v_empno);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No record exists');

END;
```

6.2 Variables

- Common data types:
  - NUMBER.
  - DATE.
  - INTEGER.
  - VARCHAR2.
  - CHAR.
  - BOOLEAN.

6.3 SELECT INTO example

```
DECLARE
    v_job emp.job%TYPE;
    v_sal emp.sal%TYPE;
    v_empno emp.empno%TYPE;

BEGIN
    v_empno := 1234567;
    SELECT job, sal
    INTO v_job,v_sal
    FROM emp
    WHERE empno = v_empno;

END;
```

6.4 IF example

```
DECLARE
    -- ...

BEGIN
    -- ...
    IF v_dept = 10 THEN
        v_commision := 5000;
    ELSIF v_dept = 20 THEN
        v_commison := 5500;
    ELSIF v_dept = 30 THEN
        v_commison := 6200;
    ELSE
        v_commision := 7500;
    END IF;
    -- ...

END;
```

6.5 Loops

- LOOP, EXIT WHEN, END LOOP.
- FOR, IN, LOOP, END LOOP.
- WHILE, LOOP, END LOOP.

6.5.1 LOOP example

```
LOOP
    INSERT INTO dept(deptno)
    VALUES(v_deptno);
    v_counter := v_counter + 1;
    v_deptno := v_deptno + 10;
    EXIT WHEN v_counter > 5;
END LOOP;
```

6.5.2 FOR example

```
FOR v_counter IN 1..5 LOOP
    INSERT INTO dept(deptno)
    VALUES(v_deptno);
```

```
    v_deptno := v_deptno + 10;
END LOOP;
```

6.5.3 WHILE example

```
v_counter := 1;
WHILE v_counter <= 5 LOOP
    INSERT INTO dept(deptno)
    VALUES(v_deptno);
    v_deptno := v_deptno + 10;
END LOOP;
```

6.6 Procedures

6.6.1 Syntax

```
CREATE OR REPLACE PROCEDURE /*name*/(/*parameters*/) IS
    -- local variables

BEGIN
    -- ...

EXCEPTION
    -- ...

END;
```

- Parameters can be `IN`, `OUT` or `IN OUT`.

6.6.2 Example

```
CREATE OR REPLACE PROCEDURE proc_test(p_empno IN VARCHAR2) IS
    v_job EMP.job%TYPE;
    v_sal EMP.sal%TYPE;

BEGIN
    SELECT job, sal
    INTO v_job,v_sal
    FROM emp
        WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('job is '||v_job);

EXCEPTION
```

```
    WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('ERROR...');

END;
```

6.7 Functions

6.7.1 Syntax

```
CREATE OR REPLACE FUNCTION /*name*/(/*parameters*/)
RETURN /*datatype*/ IS
    -- local variables

BEGIN
    -- ...

EXCEPTION
    -- ...

END;
```

- Paremeters can only be `IN`.
- Returns a single value.

6.8 Packages

6.8.1 Specification example

```
CREATE OR REPLACE PACKAGE emp_info IS

    v_count INTEGER;

    PROCEDURE insert_record(  p_empno IN NUMBER
                            , p_ename IN VARCHAR2
                            , p_job IN VARCHAR2
                            , p_sal IN NUMBER
                            , p_comm IN NUMBER
                            , p_deptno IN VARCHAR2);

    PROCEDURE delete_record(p_empno IN NUMBER);

    FUNCTION sum_dept_sal( p_deptno IN dept.deptno%TYPE) RETURN is dept.sal%TYPE;
```

```
END emp_info;
```

6.8.2 Body definition syntax

```
CREATE OR REPLACE PACKAGE BODY emp_info IS

    -- define declared procedures and functions

END emp_info;
```

6.9 Triggers

6.9.1 Syntax example

```
CREATE OR REPLACE TRIGGER del_emp( p_empno emp.empno%TYPE)
BEFORE DELETE ON emp
FOR EACH ROW
BEGIN
    INSERT INTO emp_audit
    VALUES(p_empno, USER, sysdate);

END;
```

6.10 Cursors

- A **cursor** is a pointer to a row.

6.10.1 Syntax example

```
DECLARE
    CURSOR c_emp IS
    SELECT empno, ename, job
    FROM emp
    WHERE deptno = 20;

BEGIN
    FOR v_c IN c_emp LOOP
        DBMS_OUTPUT.PUT_LINE(v_c.ename);
    END LOOP;
```

```
END;
```

6.11 Dynamic SQL

```
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE tt(id NUMBER(3)'
END;
```

Chapter 7

NoSQL and NoSQL types

7.1 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**: failures do not affect the entire system.
    * **Soft state**: data copies may be inconsistent.
    * **Eventually consistent**: consistency is obtained over time.
  - 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.

7.2 Motivation

- Explosion of social media sites with huge data needs.
- Explosion of storage needs and cloud-based solutions such as AWS.
- Shift to more dynamic data with frequent schema changes.

7.2.1 Parallel databases and data stores

- Scaling server applications is easy, but not databases. Possible approaches:
    1. `memcache` or similar caching mechanisms. Limited in scalability.
    2. Use existing parallel databases. Expensive and most of them do not support **OLTP** *(online transaction processing)*.
    3. Build parallel stores with databases underneath.

7.2.2 Sharding

- Consists in the use of multiple cheap databases.
- **Sharding** can be used to partition and scale RDBMSs.
    - Scales well, but it is **not transparent**.

7.2.3 Parallel key-value data stores

- Distributed and **transparently** partitionable/scalable.
- No support for joins or constraints.

7.2.4 Scalability

- Necessary due to big data growth.
- **Vertical scalability** *(scale-up)*: increasing performance of a single machine.
    - Hard to manage.
    - Possible down times.
- **Horizontal scalability** *(scale-out)*: increase the number of machines.
    - Elastically scalable.
    - Cheaper.
    - Heterogeneity.
- Issue with **NoSQL** and multiple machines: **coordination** between nodes.

7.3 CAP theorem

7.3.1 Network partitions

- A **network partition** occurs when a failure of a node splits the network.

7.3.2 C-A-P

- **Consistency**, **availability** and **partition-resilience**.
- Choose two:
  - **CA**: available and consistent, unless there is a partition.
  - **AP**: a replica provides service even in case of a partition, but can be inconsistent.
  - **CP**: always consistent, but a replica may deny service to prevent inconsistency.

7.3.3 Log-based transactions

- In order to prevent partial transactions from being committed, a **log** is used.
  - After a crash, different actions are taken depending on the data present in the log.
- **Commit protocols** are used to prevent incoherences.

7.4 NoSQL types

- Key-value stores.
- Column NoSQL databases.
- Document-based.
- Graph databases.
- XML databases.

7.4.1 Key-value stores

- Extremely simple interface.
- Data model: **key-value pairs**.
  - No explicit relationships.
  - No queries-by-data.
  - No set operations.
- Operations:

  - `insert(k, v)`.
  - `fetch(k)`.
  - `update(k, v)`.
  - `delete(k)`.
- Implementation:
  - Records distributed to nodes depending on key.
  - Replciation.
  - Single-record transactions *(eventual consistency)*.
    * No multi-operation transactions.
- Examples: SimpleDB, Riak.
- Use for: storing session information, user profiles, shopping carts.

7.4.2 Document stores

- Similar to key-value stores, except that values are **documents**.
- Data model: **key-document pairs**.
  - Document: **JSON**, **XML**, etc…
- Operations: like key-value stores.
- Examples: CouchDB, MongoDB, SimpleDB.
- Use for: event logging, CMSs, analytics, e-commerce.
- *Example:* MongoDB.

7.4.3 Column-oriented

- Data is stored in **column order**.
  - Key-value pairs can be stored and retrieved in massively parallel systems.
- Data model: **families of attributes** defined in a schema.
- Storing principle: **big hashed distributed tables**.
- Properties:
  - Horizontal and vertical partitioning.
  - High availability.
  - Transparency to application.
- *Example:* Cassandra.

7.4.4 Graph database

- Data model: **nodes** and **edges**.
- Interface and query languages vary.
- *Examples:* Neo4j, FlockDB, Prgel.

Chapter 8

Cassandra

8.1 Background

- **Cassandra** is an open-source DBMS.

8.1.1 History

- Created to power **Facebook Inbox Search**.
- Open sourced in 2008 as an Apache Incubator project.

8.1.2 Motivation and function

- Can handle large amounts of data across multiple servers.
- Mimics relational DBMS, using **triggers** and **lightweight** transactions.
- Raw and simple data structures.
- Focus on availability.

8.2 Design

- Emphasis on **performance** over analysis.

8.2.1 Data organization

- Rows are organized into tables.
- First component of a table's primary key is the **partition key**.
- Rows are clustered by the remaining columns of the key.

- Columns may be indexed separately from primary key.
- Tables can be altered at runtime without blocking queries.

8.2.1.1 Elements

- The **keyspace** wraps all keys. Usually the name of the application.
- A **column family** is a structure containing an unlimited number of rows.
- A **column** is a **tuple** with name, value and timestamp.
  - A **super column** contains more columns.d
- A **key** is a name of a record.
- Use for: CMSs, blogging platforms, event logging.

8.2.2 P2P clustering

- Decentralized design.
  - Every node has same role.
  - No single point of failure.
  - No bottlenecking.

8.2.3 Fault tolerance

- Automatic replication and replacement of faulty nodes.
- Distribution over multiple data centers.
- **AP**: availability and partitioning-tolerance.
  - Eventual consistency.

8.3 Data model

8.3.1 Key-value model

- Cassandra is **column-oriented**.
- **Column families**: sets of key-value pairs inside a **keyspace**.
  - Analogies:
    * A column family is like an SQL table.
    * Key-value pairs are like a SQL row.
- A Cassandra **row** is a sequence of key-value pairs.

- Schema is adjusted as new queries are introduced.
  - No joins.

8.4 CQL examples

8.4.1 Keyspaces

- Creation:

  ```
  CREATE KEYSPACE demo
  WITH replication = {‘class’: ’SimpleStrategy’, replication_factor’: 3};
  ```

- Usage:

  ```
  USE demo;
  ```

8.4.2 Tables

```
CREATE TABLE users(
email varchar,
bio varchar,
birthday timestamp,
active boolean,
PRIMARY KEY (email));
```

```
CREATE TABLE tweets(
email varchar,
time_posted timestamp,
tweet varchar,
PRIMARY KEY (email, time_posted));
```

8.4.3 Queries

- Insertion:

  ```
  INSERT INTO users (email, bio, birthday, active)
  VALUES (‘john.doe@bti360.com’, ‘BT360 Teammate’,
  516513600000, true);
  ```

- Selection:

  ```
  SELECT * FROM users;
  SELECT email FROM users WHERE active = true;
  ```

8.4.4 Other

“‘sql CREATE KEYSPACE Excelsior WITH replication = {‘class’: ‘SimpleStrategy’, ‘replication_factor’ : 3};

CREATE KEYSPACE Excalibur WITH replication = {‘class’: ‘NetworkTopologyStrategy’, ‘DC1’ : 1, ‘DC2’ : 3}

ALTER KEYSPACE Excelsior WITH replication = {‘class’: ‘SimpleStrategy’, ‘replication_factor’ : 4};

DROP KEYSPACE Excelsior;

CREATE TABLE timeline (userid uuid,posted_month int, posted_time uuid,body text,posted_by text, PRIMARY KEY (userid, posted_month, posted_time)) WITH compaction = { ‘class’ : ‘LeveledCompactionStrategy’ };

INSERT INTO timeline(userid, posted_month, posted_time, body, posted_by) VALUES (0, 0, 0, ‘mioTesto’, ecc ecc);

SELECT * FROM timeline WHERE userid = 0 AND posted_time = 0; ALTER TABLE timeline ADD gravesite varchar;

UPDATE timeline SET posted_month = posted_month + 2 WHERE userid = 2 AND posted_by = ‘Mario’;

DELETE posted_by FROM timeline WHERE userid IN (3, 4);

DROP TABLE timeline;

CREATE INDEX userIndex ON timeline (userid);

DROP INDEX userIndex; “‘sql

8.5 Architecture

- P2P, distributed.
  - All nodes have he same node.
  - Data partitioned among all nodes in a cluster.
- Custom data replication to ensure fault tolerance.
- Transparent elasticy and scalability.
  - No downtimes.
  - Linear performance increase with addition of nodes.
- High availability.
  - No single point of failure.
  - Multi-geography/zone aware.
    - Supports multiple geographically dispersed datacenters.

- Data redundancy.
- Partitioning.
  - Nodes structured in **ring topology**.
  - Hashed value of key used to assign it to a node.
  - Nodes move around to alleviate loads.
- **Gossip protocols**.
  - Used for node communication. Inspired by real-life gossiping.
  - Periodic, pairwise node-to-node communication.
    * Low cost.
  - Failure detection:
    * Gossiping tracks heartbeats from other nodes.
    * A **suspicion level** variable is used to detect failures.

8.6 Write operations

8.6.1 Stages

1. Log data in commit log.
2. Write data to memtable.
3. Flush data from memtable.
4. Store data on disk in SSTables.

8.6.1.1 Memtable

- Data structure in memory.
- Flushed to disk once a certain size is reached.
- Read operations start looking here.

8.6.1.2 SSTable

- Kept on disk.
- Immutable once written.
- Periodically compacted for performance.

8.6.2 Consistency

- Read consistency:
  - Number of nodes that must agree before read request returns.
  - **One to all**.
- Write consistency:
  - Number of nodes that must be updated before a write is considered successful.
  - **Any to all**.
  - At **an**, a hinted handoff is all that is needed to return.
- **Quorum**:
  - Middle-ground consistency level.
  - Defined as: $(replication_factor/2) + 1$.
- Example queries:
  - `INSERT INTO table (column1, …) VALUES (value1, …) USING CONSISTENCY ONE`
  - `INSERT INTO table (column1, …) VALUES (value1, …) USING CONSISTENCY QUORUM`

8.7 Delete operations

8.7.1 Tombstones

- Deleted data is **marked for deletion**.
- Actual deletion will happen on major compaction or configurable timer.

8.7.2 Compaction

- Runs periodically to merge multiple SSTables.
  - Reclaims space.
  - Creates new index.
  - Merges keys.
  - Combines columns.
  - Discards tombstones.
  - Improves performance.
- Two types:
  1. Major.

2. Read-only.

8.7.3 Anti-entropy

- Ensures synchronization of data across nodes.
- Compares data checksums across neighbors.
- Uses **Merkle trees** *(hash trees)*.
  - Leaves are data, intermediate nodes are hashes.

8.8 Read operations

8.8.1 Read repair

- On read, nodes are queried until a number of nodes matching specified consistency level is reached.
- If consistency level is not met, nodes are updated with most recent value, which is then returned.
- If consistency level is met, value is returned immediately and old nodes are then updated.

8.8.2 Bloom filters

- **Bloom filters** are used to check if a value is in a set.
- A value is hashed with multiple algorithms.
  - Bits of created hashes in a **bit vector** are set to `1`.
- Checking for an element:
  - Hash the element again with same functions, check bits.
    * If the element is not there, it is **certain**.
    * Otherwise, there is a small chance of **false positives**.

8.9 Conclusion

8.9.1 Advantages

- High performance.
- Decentralization.

- Linear scalability.
- Replication.
- No single points of failure.
- MapReduce support.

8.9.2 Disadvantages

- No referential integrity.
  - No `JOIN`.
- Limited querying options.
- Sorting data is a design decision.
  - No `GROUP BY`.
- No support for atomic operations.
- *"First think about queries, then data model"*.

8.9.3 Considerations

- Use Cassandra when you have a lot of data spread across multiple servers.
- Write performance is always excellent, read performance depends on write patterns.
  - Schema must be designed for the queries.

Chapter 9

MongoDB

9.1 Background

- **MongoDB** is a document-oriented NoSQL DBMS.
- Uses **BSON** format.
- Schema-less.
- **No transactions** and **no joins**.

9.2 Basics

- A MongoDB **instance** contains **databases**.
- A database contains **collections**.
  - Conceptually similar to tables in SQL.
- A collection contains **documents**.
  - Conceptually similar to records in SQL.
  - Every document has an **unique key**.
- A document contains **fields**.
- **Indexing** support.
  - Uses **B-trees**.

9.3 Examples

9.3.1 Basic

- Documents:

- ```
  user = {
      name: "Z",
      occupation: "A scientist",
      location: "New York"
      }
  ```

- Collections:
  - ```
    {
        "_id": ObjectId("4efa8d2b7d284dad101e4bc9"),
        "Last Name": "DUMONT",
        "First Name": "Jean",
        "Date of Birth": "01-22-1963"
    }
    ```
  - ```
    {
        "_id": ObjectId("4efa8d2b7d284dad101e4bc7"),
        "Last Name": "PELLERIN",
        "First Name": "Franck",
        "Date of Birth": "09-19-1983",
        "Address": "1 chemin des Loges",
        "City": "VERSAILLES"
    }
    ```

- Queries:
  - `db.users.find( {last_name: 'Smith'} )`
  - `db.users.find( {age: {$gte: 23} } )`
  - `db.users.find( {age: {$in: [23,25]} } )`

9.3.2 Complex

```
db.createCollection(miaCollection, options)
db.COLLECTION_NAME.drop()

db.miaCollection.insert({name: Mario, sesso:'M', peso: 450})
db.miaCollection.find({sesso: 'm',peso: {$gt: 700}})
db.miaCollection.update({name: 'Mario'}, {$set: {peso: 590}})
db.miaCollection.find().sort({peso: -1})
db.miaCollection.count({peso: {$gt: 50}})

db.employees.insert({
    _id: ObjectId("4d85c7039ab0fd70a117d734"),
    name: 'Ghanima',
    scores:[],
    latlong: [40.0,70.0],
    family:
```
```

```
        {mother: 'Chani',
        father: 'Paul',
        brother: ObjectId("4d85c7039ab0fd70a117d730")
        }
    })
db.employees.find({'family.mother': 'Chani'})
db.employees.update({ _id: 1 }, { $push: { scores: 89 } }
db.employees.find({latlong:{$near: { [40,70], $minDistance:
    1000,$maxDistance: 5000 }}})
```

Chapter 10

HBase

10.1 Overview

10.1.1 History

- Developed for massive natural language data search.
- Open-source implementation of Google BigTable.
  - Semi-structured data.
  - Cheap, horizontal scalability.
  - Integration with MapReduce.
- Developed as part of Hadoop, on top of HDFS.

10.1.2 Characteristics

- Non-relational, distributed.
- Column-oriented.
- Multi-dimensional.
- High availability and performance.

10.2 Data model

- **Sparse**, **multi-dimensional**, **sorted** map.
  - {row, column, timestamp} -> cell
- Rows are **lexicographically sorted** on row key.
- **Region**: contiguous set of sorted rows.

10.2.1 Operators

- Operations are based on **row keys**.
- Single-row operations:
  - Put.
  - Get.
  - Scan.
- Multi-row operations:
  - Scan.
  - MutiPut.
- No joins - use MapReduce.

10.3 Physical structures

- **Region**: unit of distribution and availability.
  - Split when grown too large.
  - Max size is a tuning parameter.
- Row keys are **plain byte arrays**.
- No support for secondary indexes.
  - Create new table with index and exploit sorting for complex queries.
  - Use libraries such as **Lily**.

10.4 System architecture

10.4.1 Components

- The **HMaster** talks to n **HRegionServer** instances.
- HRegionServers contain **HRegion** instances.
- HRegions contain **HLog** and multiple **memstores**.
- The memstores contain **StoreFiles** which are **HFiles** that interact with Hadoop.

10.5 ACID properties

- HBase is **not ACID compliant**.
- Guarantees:

- Atomicity:
  * All mutations are atomic within a row.
- Consistency and Isolation:
  * Eventual Consistency.
- Durability:
  * All visible data is durable data.

10.6 Examples

```
create 'impiegato', 'personali', 'professionali'

scan 'impiegato'

drop 'impiegato'

put 'impiegato', 'row1', 'personali:nome', 'mario'

put 'impiegato', 'row1', 'personali:cognome', 'rossi'

put 'impiegato', 'row1', 'personali:eta', '65'

get 'impiegato', 'row1', {COLUMN => ['personali:nome',
    'personali:eta']}
```

Chapter 11

Neo4J

11.1 Graph databases

- Schema-less.
- Efficient storage of semi-structured data.
- No **O/R mismatch**.
  - Natural to map a graph to OOP language.
- Express queries as traversals.
- Express graph-related problems.
  - *Example:* does a path exist between A and B?

11.2 Features

- Both **nodes** and **edges** can contain **properties**.
- Edges are **relationships**:
  - They have a start node and end node.
  - Have a relationship type.
  - Can have properties.
- **ACID**.
  - Transaction support.
- Query language: **Cypher**.
- Bad horizonal scalability:
  - Read-only scalability: all writes go to master, then fan out.

11.3 Examples

```
CREATE (p1:Profilo1)

CREATE (m:Movie:Cinema:Film:Picture)

CREATE (p1:Profilo1)-[relazione1:LIKES]->(p2:Profilo2)

MATCH (emp:Employee) RETURN emp.empid,emp.name,emp.salary,emp.deptno

MATCH (emp:Employee) WHERE emp.name = 'Abc' RETURN emp

MATCH (emp:Employee) WHERE emp.name = 'Abc' OR emp.name = 'Xyz' RETURN emp

MATCH (cust:Customer),(cc:CreditCard)
WHERE cust.id = "1001" AND cc.id= "5001"
CREATE (cust)-[r:DO_SHOPPING_WITH{shopdate:"12/12/2014"}]->(cc)
RETURN r

MATCH (cc:CreditCard)-[r]-(c:Customer)RETURN r

MATCH (cc: CreditCard)-[rel]-(c:Customer) DELETE cc,c,rel

MATCH (e: Employee) DELETE e
```

Chapter 12

XML

- **XML** *(extensible markup language)* is both a markup language and a meta-language to specify markup languages.
- A data model can be defined using **DTD** or **XSD**.
- Queries can be executed with **Xquery** or **XSL**.
- An XML document is **well-formed** when the syntax is valid.
- An XML document is **valid** when the contents respect a data model *(schema)*.
- Namespaces are handled by using prefixes.

12.1 DTD

- Defining subelement occurrences:

```
<!ELEMENT product (description)>
<!ELEMENT product (description?)>

<!ELEMENT product_list (product+)>
<!ELEMENT product_list (product*)>
```

- Attributes/modifiers:
  - `CDATA`: character data.
  - `ID`: identifier.
  - `IDREF`: this value is an ID of anoter element.
  - `ENTITY`: this value is an entity.
  - `NMTOKEN`: this value is a valid XML name.
- Constraints:
  - `#REQUIRED`.
  - `#IMPLIED`: can be absent.

– `#FIXED "x"`: value needs to be `x`.
– `#DEFAULT "x"`.

12.2 XSD

- Another schema definition language.
- Compared to DTD:
  – More extensible and richer.
  – Can manage multiple namespaces.
  – Are XML themselves.

12.2.1 Example

```xml
<xs:element name="Attributo" type="xs:string">
    <xs:attribute name="lang" type="xs:string"
        use="required"/>
</xs:element>

<xs:element name="age">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="120"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<xs:element name="car">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:enumeration value="Audi"/>
            <xs:enumeration value="Golf"/>
            <xs:enumeration value="BMW"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<xs:complexType name="tipoComplessoMio">
    <xs:sequence>
        <xs:element name="firstname" type="xs:string"
            minOccurs="0" maxOccurs= "2"/>
```

```
        <xs:element name="lastname" type="xs:string"
            minOccurs="2"/>
    </xs:sequence>
</xs:complexType>
<xs:element name="employee" type="tipoComplessoMio"/>

<xs:complexType name="tipoComplessoMioESTESO">
    <xs:complexContent>
        <xs:extension base="tipoComplessoMio">
            <xs:sequence>
                <xs:element name="address" type="xs:string"/>
                <xs:element name="city" type="xs:string"/>
                <xs:element name="country" type="xs:integer"/>
            </xs:sequence>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
<xs:element name="amministratore" type="tipoComplessoMioESTESO"/>

<xs:group name="custGroup">
    <xs:sequence>
        <xs:element name="customer" type="xs:string"/>
        <xs:element name="orderdetails" type="xs:string"/>
    </xs:sequence>
</xs:group>

<xs:complexType name="ordertype"> Riuso di "custGroup"
    <xs:group ref="custGroup"/>
    <xs:attribute name="status" type="xs:string"/>
</xs:complexType>
<xs:element name="esempioGRUPPO" type="ordertype"/>
```

12.3 XSL

- **Extensible stylesheet language**.
- Specifies how XML output is reprsented.
- **XSLT** *(XSL transformation)* transforms an XML in another XML or a different type *(like HTML)*.

12.4 Xquery

- Can use **Xpath** expressions to query XML documents.

- Examples:

```
doc("books.xml")/List/Book
doc("books.xml")/List/Book[Editore = 'Bompiani']/Title
doc("books.xml")//Author
doc("books.xml")/List/Book[2]/*
```

- Can use complex **Xquery** expressions combined with Xpath.
  - FOR, LET, WHERE, ORDER BY, RETURN, INSERT, DELETE.
  - Examples:

```
for $book in doc("books.xml")//Book
return $book
```

```
for $book in doc("books.xml")//Book
WHERE $book/Editor = "Bompiani" and $book/@availability = "S"
return $book
```