

Introduction to Databases

Lecture 3 – Relational database management systems

Gianluca Quercini

gianluca.quercini@centralesupelec.fr

Master DSBA 2020 – 2021



CentraleSupélec

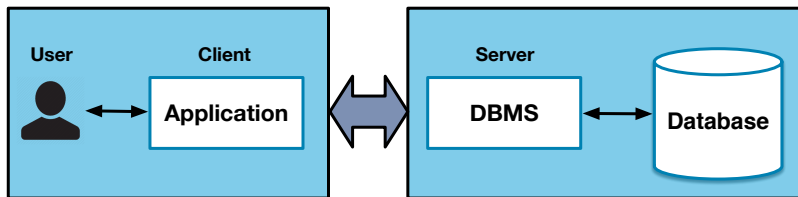
What you will learn

In this lecture you will learn:

- The difference between **client-server** and **embedded** database systems.
- What **SQLite** is and its features.
- How to interact with a relational DBMS with **SQL**.

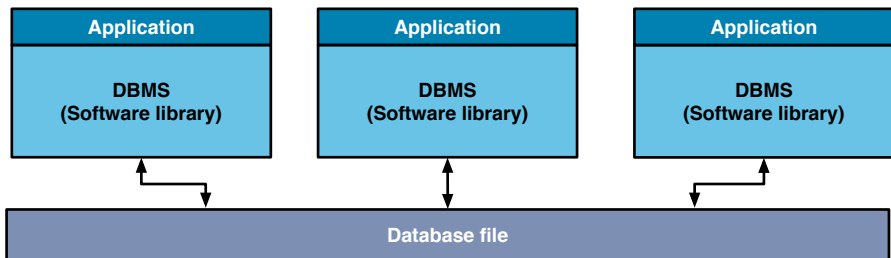
Client-server database systems

- The DBMS is a **process** that *waits* for requests for access to the database (hence, the name **server**).
- A **client** is any **application** that requests data to the DBMS.
- The server may reside on a different machine than the client.
- **Examples:** Oracle, MySQL, Microsoft SQL Server, PostgreSQL, IBM Db2.



Embedded database systems

- The DBMS is a **software library**, a collection of **functions** that can be called by an application to access a database.
- The DBMS is **embedded** (i.e., *integrated*) into the application that accesses the database.
- The **database** is typically a single file stored on disk.
- **Examples:** SQLite, Informix, Firebird, HyperSQL, solidDB.



Client-server vs embedded database systems

Client-server database systems **must be preferred** over embedded systems if one of the following conditions is met:

- **Client-server applications.** Many client applications request for access to a database over a network.
- **High-volume websites, high-concurrency.** Accesses to the website translate to many, possibly concurrent, write operations to the database.
- **Very large datasets.** Big data applications.

► Source



In this course, we're going to use **SQLite**, an embedded database system.

Main features of SQLite

- **Zero-configuration.** No installation, nor configuration needed.
- **Serverless.** No server process that manages the requests for access.
 - Many applications can **read** the database **concurrently**.
 - Only one application can **write** to the database at any given time.
- **Single database file.** The database can easily be transferred to another computer.
- **Stable cross-platform database file.** The file can be read on different machines with different architectures and is **backward compatible**.
- **Manifest typing.** The values in a column may have different types.

[► Source](#)

SQLite data types

- **NULL.** Used to indicate the absence of a value.
- **INTEGER.** Signed integer, stored in 1, 2, 3, 4, 6 or 8 bytes, depending on the magnitude of the value.
 - Boolean values are stored as integers: 0 (false) and 1 (true).
- **REAL.** Floating point value, stored as a 8-byte IEEE floating point number.
- **TEXT.** Text string, stored using the database encoding (UTF-8, UTF-16).
- **BLOB.** Blob of data, stored as it was written.

► Source

Representing date and time

👉 SQLite does not have a datatype for representing date and time. It does provide **date and time functions** capable of storing dates and times as TEXT, REAL or INTEGER values.

- **TEXT.** ISO8601 strings ("YYYY-MM-DD HH:MM:SS.SSS").
- **REAL.** Julian day numbers.
 - The number of days since noon in Greenwich on November 24, 4714 B.C. according to the proleptic Gregorian calendar.
- **INTEGER.** A **timestamp** as **Unix Time**.
 - The number of seconds since 1970-01-01 00:00:00 UTC (a.k.a., the **epoch**).

Accessing a database from an application

- An application accesses a database by using an **Application Programming Interface (API)**.

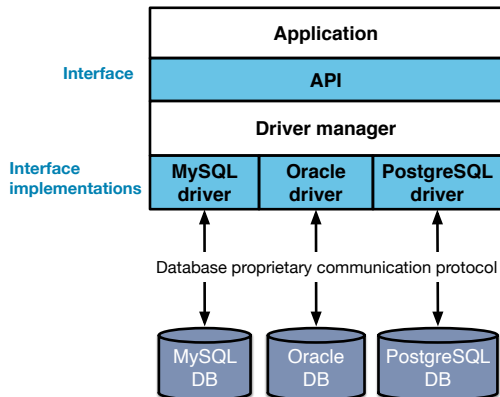
Definition (Application Programming Interface)

An **Application Programming Interface**, or simply **API**, is a collection of **functions** used by an application to interact with another application.

👉 The API defines what the functions are and what they do (**interface**), but not how they do it.

- Each database vendor provides an *implementation* of the API that is called a **driver**.
- The **driver** processes the function calls from the application and translates them into requests for the database.

Accessing a database from an application



The application calls a function provided by the API

The **driver manager** loads the driver associated to the database to access

The **driver** implements the API function to access the underlying database

The **same application** can communicate with **different databases** with the **same API** (using the appropriate driver)

Accessing a SQLite database

- From a Python program: using the **sqlite3** module.
- Using a **graphical front-end**.
- In both cases, **you'll need to use SQL** to read and write data from/to a database.


The SQL Language

- **Structured Query Language (SQL)** is the standard language used to read/write data from/to relational databases.
 - Introduced by Donald D. Chamberlin and Raymond F. Boyce at IBM Research in the early 1970s.
 - Standard since 1986.
- **Functions** to create, modify and delete tables.
- **Functions** to **C**reate, **R**ead, **U**ppdate and **D**elate data (**CRUD**).
- Defines **integrity constraints**:
 - **Key constraints** (UNIQUE, PRIMARY KEY and FOREIGN KEY).
 - **NOT NULL** constraints.
 - **CHECK** constraints.

👉 SQL is a **declarative** language (as opposed to **imperative**): you tell the DBMS what you want, not how to do it.

Create a table

```
CREATE TABLE Department  
(  
    codeD INTEGER,  
    nameD TEXT,  
    budget REAL  
)
```

 The SQL code shown here and in the following slides has been tested in SQLite. Since SQL is a standard language, the same queries should work in other relational DBMS too.

Integrity constraints

```
CREATE TABLE Department
(  
    codeD INTEGER PRIMARY KEY,  
    nameD TEXT UNIQUE,  
    budget REAL NOT NULL  
)
```

```
CREATE TABLE Employee
(  
    codeE INTEGER PRIMARY KEY,  
    first TEXT,  
    last TEXT,  
    position TEXT,  
    salary REAL DEFAULT 30000 CHECK(salary >= 30000),  
    codeD INTEGER  
)
```

Foreign key constraints

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    first TEXT,
    last TEXT,
    position TEXT,
    salary REAL DEFAULT 30000,
    codeD INTEGER,
    FOREIGN KEY(codeD) REFERENCES Department (codeD)
)
```

Foreign constraints

- When we delete/update a department, the DBMS will set the value NULL in the column CodeD of the employees working in that department.

```
CREATE TABLE Employee
(
  codeE INTEGER PRIMARY KEY,
  first TEXT,
  last TEXT,
  position TEXT,
  salary REAL DEFAULT 30000,
  codeD INTEGER,
  FOREIGN KEY(codeD) REFERENCES Department (codeD) ON DELETE SET NULL
  ON UPDATE SET NULL
)
```


Foreign key constraints

- When we delete/update a department, the DBMS will set the default value in the column CodeD of the employees working in that department.

👉 The default value **must exist** and must refer to an **existing department**.

```
CREATE TABLE Employee
(
  codeE INTEGER PRIMARY KEY,
  first TEXT,
  last TEXT,
  position TEXT,
  salary REAL DEFAULT 30000,
  codeD INTEGER DEFAULT 0,
  FOREIGN KEY(codeD) REFERENCES Department (codeD)
    ON DELETE SET DEFAULT ON UPDATE SET DEFAULT )
```

Foreign key constraints

- When we delete/update a department, the DBMS will delete all employees working in that department/update the value of codeD of all employees working in that department.

```
CREATE TABLE Employee
(
  codeE INTEGER PRIMARY KEY,
  first TEXT,
  last TEXT,
  position TEXT,
  salary REAL DEFAULT 30000,
  codeD INTEGER DEFAULT 0,
  FOREIGN KEY(codeD) REFERENCES Department (codeD)
                                ON DELETE CASCADE
                                ON UPDATE CASCADE
)
```

Foreign key constraints

- Deleting/updating a department is not allowed.

👉 If we don't specify the options ON DELETE, ON UPDATE, this is the default choice.

```
CREATE TABLE Employee
(
  codeE INTEGER PRIMARY KEY,
  first TEXT,
  last TEXT,
  position TEXT,
  salary REAL DEFAULT 30000,
  codeD INTEGER DEFAULT 0,
  FOREIGN KEY(codeD) REFERENCES Department (codeD)
                                ON DELETE NO ACTION
                                ON UPDATE NO ACTION
)
```

SQL queries

Definition (Query)

A **query** is a request for data from a database. A SQL query is a sequence of **declarative clauses**.

Query in SQL

```
SELECT  $C_1, C_2, \dots, C_n$   
FROM  $T_1, T_2, \dots, T_k$   
WHERE  $P$ 
```

- Returns the values of columns C_1, C_2, \dots, C_n (clause SELECT)
- of all rows satisfying the predicate P (clause WHERE)
- from the tables T_1, T_2, \dots, T_k (clause FROM).

SQL queries

- `SELECT *` returns all columns.

Select all employees

```
SELECT *  
FROM Employee
```

- Alternatively, we can just select specific columns.

```
SELECT first, last  
FROM Employee
```

SQL queries

Select all employees whose salary is higher than 60,000

```
SELECT *  
FROM Employee  
WHERE salary > 60000
```

Select the family name and the department code of all secretaries making more than 30,000.

```
SELECT last, codeD  
FROM Employee  
WHERE position='Secretary' AND salary > 30000
```

SQL queries

Select the code of the employees that are either secretaries or team leaders.

```
SELECT codeE
FROM Employee
WHERE position = 'secretary' OR position= 'Team Leader';
```

Select the code of the employees that are either secretaries or team leaders.

```
SELECT codeE
FROM Employee
WHERE position IN ('Secretary', 'Team Leader')
```

Sorting

Select the code and the salary of all employees sorted by their salary in **descending order**.

```
SELECT codeE, salary  
FROM Employee  
ORDER BY salary DESC
```

Select the code and the salary of all employees sorted by their salary in **ascending order**.

```
SELECT codeE, salary  
FROM Employee  
ORDER BY salary ASC
```


Aggregate functions

- Perform computations on a **group** of rows.
- A group of rows is specified by using the clause GROUP BY.

👉 If the clause GROUP BY is not specified, the aggregate function is applied on all rows of the table.

Return the maximum salary.

```
SELECT MAX(salary)
FROM Employee
```

Return the minimum salary.

```
SELECT MIN(salary)
FROM Employee
```

Aggregate functions

Return the average salary.

```
SELECT AVG(salary)  
FROM Employee
```

Return the sum of the salaries.

```
SELECT SUM(salary)  
FROM Employee
```

Aggregate functions

Counts the number of employees.

```
SELECT COUNT(*)  
FROM Employee
```

Return the number of positions ignoring the duplicates.

```
SELECT COUNT(DISTINCT position)  
FROM Employee
```

The clause GROUP BY

- Partitions the table into **non-overlapping groups** of rows.
- The aggregate function is applied separately to each group.

Return the average salary for each department.

```
SELECT codeD, AVG(salary)
FROM Employee
GROUP BY codeD
```

Count the number of employees per department and position.

```
SELECT codeD, position, count(*) AS nbEmployees
FROM Employee
GROUP BY codeD, position
```

Using predicates with aggregate functions

Return the average salary by department, but **only for the departments that have at least 3 employees**.

```
SELECT codeD, avg(salary) AS avgSalary
FROM Employee
WHERE COUNT(*) >=3
GROUP BY codeD
```

- This query is not accepted by the DBMS!
- **Question:** Could you tell why?
- **Answer.** The DBMS cannot check any condition on a value (COUNT(*)) that is the output of the query itself.

Using predicates with aggregate functions

- We need to use the clause HAVING.
- The clause HAVING specifies search conditions on groups of rows.
- A combination of Boolean predicates can be used in the HAVING clause.
 - Each predicate in the HAVING clause **must** use an aggregate function.

Return the average salary by department but **only for the departments that have at least 3 employees**.

```
SELECT codeD, avg(salary) AS avgSalary
FROM Employee
GROUP BY codeD
HAVING COUNT(*) >=3
```

Natural join

Employee					
codeE	first	last	position	salary	codeD
1	Joseph	Bennet	Office assistant	55,000	14
2	John	Doe	Budget manager	60,000	62
3	Patricia	Fisher	Secretary	45,000	25
4	Mary	Green	Credit analyst	65,000	62
5	William	Russel	Guidance counsellor	35,000	25
6	Elizabeth	Smith	Accountant	45,000	62
7	Michael	Watson	Team leader	80,000	14
8	Jennifer	Young	Assistant director	120,000	14

Department		
codeD	nameD	budget
14	Administration	300,000
25	Education	150,000
62	Finance	600,000
45	Human Resources	150,000

- Joins two tables based on the value of the columns that have the **same name**.
- Costly operation when the two tables have a lot of rows.

Natural join

Employee NATURAL JOIN Department

codeE	first	last	position	salary	codeD	nameD	budget
1	Joseph	Bennet	Office assistant	55,000	14	Administration	300,000
2	John	Doe	Budget manager	60,000	62	Finance	600,000
3	Patricia	Fisher	Secretary	45,000	25	Education	150,000
4	Mary	Green	Credit analyst	65,000	62	Finance	600,000
5	William	Russel	Guidance counselour	35,000	25	Education	150,000
6	Elizabeth	Smith	Accountant	45,000	62	Finance	600,000
7	Michael	Watson	Team leader	80,000	14	Administration	300,000
8	Jennifer	Young	Assistant director	120,000	14	Administration	300,000

Natural join

- Joins the two tables on the columns that have the **same name**.

```
SELECT *  
FROM Employee NATURAL JOIN Department
```

- The following formulation allows the explicit specification of the **join condition**.

```
SELECT *  
FROM Employee e JOIN Department d ON e.codeD = d.codeD
```

Natural join

- We can join different tables pairwise.

```
SELECT *  
FROM Employee e JOIN Department d JOIN Membership m  
    ON (e.codeD=d.codeD and e.codeE=m.codeE)
```

- Alternative formulation:

```
SELECT *  
FROM Employee e, Department d, Membership m  
WHERE e.codeD=d.codeD and e.codeE=m.codeE
```

Right outer join

- The natural join operation will only include in the result the departments that have at least one employee.
- A **right outer join** will complete the table with all the data in the table that is on the **right** side of the join operation.

```
SELECT *  
FROM Employee e RIGHT OUTER JOIN Department d ON(e.codeD=d.codeD)
```

Right outer join

Question. Which values on the Employee side?

Employee RIGHT OUTER JOIN Department							
codeE	first	last	position	salary	codeD	nameD	budget
1	Joseph	Bennet	Office assistant	55,000	14	Administration	300,000
2	John	Doe	Budget manager	60,000	62	Finance	600,000
3	Patricia	Fisher	Secretary	45,000	25	Education	150,000
4	Mary	Green	Credit analyst	65,000	62	Finance	600,000
5	William	Russel	Guidance counsellor	35,000	25	Education	150,000
6	Elizabeth	Smith	Accountant	45,000	62	Finance	600,000
7	Michael	Watson	Team leader	80,000	14	Administration	300,000
8	Jennifer	Young	Assistant director	120,000	14	Administration	300,000
?	?	?	?	?	45	Human Resources	150,000

Right outer join

Answer. NULL values.

Employee RIGHT OUTER JOIN Department							
codeE	first	last	position	salary	codeD	nameD	budget
1	Joseph	Bennet	Office assistant	55,000	14	Administration	300,000
2	John	Doe	Budget manager	60,000	62	Finance	600,000
3	Patricia	Fisher	Secretary	45,000	25	Education	150,000
4	Mary	Green	Credit analyst	65,000	62	Finance	600,000
5	William	Russel	Guidance counsellor	35,000	25	Education	150,000
6	Elizabeth	Smith	Accountant	45,000	62	Finance	600,000
7	Michael	Watson	Team leader	80,000	14	Administration	300,000
8	Jennifer	Young	Assistant director	120,000	14	Administration	300,000
NULL	NULL	NULL	NULL	NULL	45	Human Resources	150,000

Left outer join

- The following query will only return employees who are members of an association.
- What about the others?

```
SELECT *  
FROM Employee e JOIN Membership m ON(e.codeE=m.codeE)
```

- A **left outer join** will complete the table with all the data in the table that is on the **left** side of the join operation.

```
SELECT *  
FROM Employee e LEFT OUTER JOIN Membership m ON(e.codeE=m.codeE)
```

Full outer join

- The union of a left outer join with a right outer join.

```
SELECT *  
FROM Employee e FULL OUTER JOIN Department d ON(e.codeD=d.codeD)
```

👉 RIGHT OUTER JOIN and FULL OUTER JOIN are currently not supported in SQLite.

Right and full outer join in SQLite

- Employee RIGHT OUTER JOIN Department

```
SELECT *  
FROM Department d LEFT OUTER JOIN Employee e ON(e.codeD=d.codeD)
```

- Employee FULL OUTER JOIN Department

```
SELECT e.codeE, e.first, e.last, e.salary, d.codeD, d.nameD, d.budget  
FROM Employee e LEFT OUTER JOIN Department d  
    ON(e.codeD=d.codeD)  
UNION  
SELECT e.codeE, e.first, e.last, e.salary, d.codeD, d.nameD, d.budget  
FROM Department d LEFT OUTER JOIN Employee e  
    ON(e.codeD=d.codeD)
```


Subqueries

- A **subquery** is a query embedded in another query.
- The subquery can be placed in the clauses FROM, WHERE, HAVING.

```
SELECT first, last
FROM Employee
WHERE codeE in
      (SELECT codeE
       FROM Membership m NATURAL JOIN Association a
       WHERE a.nameA="Soccer")
```

Subqueries

- Example of a subquery in the clause FROM.

```
SELECT first, last
FROM Employee e NATURAL JOIN
    (SELECT codeE
     FROM Membership m NATURAL JOIN Association a
     WHERE a.nameA="Soccer") t
```

Subqueries

Get the name of the department that has the most of employees.

```
SELECT nameD
FROM Department
WHERE codeD IN
    ( SELECT codeD
      FROM Employee
      GROUP BY codeD
      HAVING count(*) =
        ( SELECT MAX(n)
          FROM
            (SELECT count(*) as n
             FROM Employee
             GROUP BY codeD) temp
        )
    )
```

INSERT

- INSERT is used to add rows to a table.

```
INSERT INTO Department VALUES (234, 'New dept', 30000)
```

- You can also specify the values of a subset of the columns.
- The other columns will be given the default value.
- If no default value is specified, the column is given the NULL value.
- If the column is declared with the constraint NOT NULL, an error is returned.

```
INSERT INTO Department (codeD, budget) VALUES (234, 30000)
```

UPDATE

- UPDATE is used to change values in selected rows of a table.

```
UPDATE department  
SET nameD="new name"  
WHERE codeD=234
```

DELETE

- DELETE is used to delete rows from a table.

```
DELETE FROM department  
WHERE codeD=234
```

- If you do not specify the WHERE clause, all rows in the table are deleted!

```
DELETE FROM Department
```