Introduction to Databases

Gianluca Quercini

Laboratoire de Recherche en Informatique
CentraleSupélec

2019 – 2020

Using a Relational Database Management System

Which DBMS?

Two types of DBMS:

1. **Client-server DBMS.**
    - The DBMS is a **server** application.
    - A **client** application connects to the DBMS server to access the databases under the control of the DBMS.
        - The client may not run on the same machine as the server.
2. **Embedded DBMS.**
    - The DBMS is a **software library**.
    - The client application uses the functions of the software library.
    - The actual database is just a file.

Which DBMS?

- When you should use a client-server DBMS:
  - Data is separated from the application by a network.
  - Many applications write to the database concurrently.
  - Big Data (in the order of the terabytes).
- Other cases: either choice is fine.
- Examples of **client/server DBMS**:
  - MySQL, Oracle, PostgreSQL, Microsoft SQL Server.
- Examples of **embedded databases**:
  - SQLite, HyperSQL Database (hybrid), Apache Derby (hybrid).
- Interacting with a relational DBMS to access databases: the **SQL** language.

MySQL

- MySQL is one of the most used client/server DBMS.
- Community edition (free) and commercial editions.
- Ways to “talk” to the DBMS:
  - Command-line **shell**.
  - **MySQL Workbench**: a client with a graphical interface.
  - **Custom application**.
    - Any major DBMS provides software libraries (called **drivers**) that allows applications to interact with the DBMS.
    - Python applications need the MySQL Connector/Python.

The SQL Language

- **Structured Query Language** (SQL) is the language 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.
- Several **data types** (INTEGER, FLOAT, VARCHAR...).
- Defines **functions** to create, modify and delete tables.
- Defines **functions** to **C**reate, **R**ead, **U**pdate and **D**elete data (**CRUD** operations).
- Defines **integrity constraints**:
    - **Key constraints** (UNIQUE and PRIMARY KEY).
    - **NOT NULL** constraints.
    - **Foreign key** constraints (FOREIGN KEY).
    - **CHECK** constraints.

Data Types

String Data Types.

- CHAR(M). **Fixed-length** strings with M characters.
  - $0 \leq M \leq 255$.
  - Storage space: M bytes, independently of the actual length of the stored value.
- VARCHAR(M). **Variable-length** strings with M characters.
  - $0 \leq M \leq 65,535$
  - Storage space: $1 + s$ bytes (if $M \leq 255$), or $2 + s$ bytes (if $M > 255$). s is the length of the stored string.
- TEXT. Fixed max size of 65,535 characters.
  - $0 \leq M \leq 65,535$
  - Storage space: $2 + s$ bytes. s is the length of the stored string.

Data Types

- TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT
- FLOAT, DOUBLE
- BIT
- DATE, TIME, DATETIME, YEAR
- Spatial data types: GEOMETRY, POINT, POLYGON...

Create a Database

```
CREATE DATABASE (IF NOT EXISTS) company;
```

To connect to the database:

```
USE company;
```

After executing the command USE, any command is directed to the selected database (`company` in our case).

Create a Table

```
CREATE TABLE Department
(
    codeD INT,
    nameD VARCHAR(50),
    budget FLOAT
)
```

Create a Table — Integrity Constraints

```
CREATE TABLE Department
(
    codeD INTEGER PRIMARY KEY,
    nameD VARCHAR(50) UNIQUE,
    budget FLOAT NOT NULL
)
```

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    first VARCHAR(50),
    last VARCHAR(50),
    position VARCHAR(50),
    salary FLOAT DEFAULT 30000,
    codeD INTEGER
)
```

Create a Table — Integrity Constraints

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    first VARCHAR(50),
    last VARCHAR(50),
    position VARCHAR(50),
    salary FLOAT DEFAULT 30000 CHECK(salary >= 30000),
    codeD INTEGER
)
```

A Zoom on Foreign Keys

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    first VARCHAR(50),
    last VARCHAR(50),
    position VARCHAR(50),
    salary FLOAT DEFAULT 30000,
    codeD INTEGER,
    FOREIGN KEY(codeD) REFERENCES Department (codeD)
)
```

A Zoom on Foreign Keys

Option ON DELETE (UPDATE) SET NULL.

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    last VARCHAR(50),
    position VARCHAR(50),
    salary FLOAT DEFAULT 30000,
    codeD INTEGER,
    FOREIGN KEY(codeD) REFERENCES Department (codeD) ON DELETE SET NULL
)
```

A Zoom on Foreign Keys

Option ON DELETE (UPDATE) SET DEFAULT.

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    first VARCHAR(50),
    last VARCHAR(50),
    position VARCHAR(50),
    salary FLOAT DEFAULT 30000,
    codeD INTEGER,
    FOREIGN KEY(codeD) REFERENCES Department (codeD) ON DELETE
                                                       SET DEFAULT
)
```

A Zoom on Foreign Keys

Option ON DELETE (UPDATE) CASCADE.

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    first VARCHAR(50),
    last VARCHAR(50),
    position VARCHAR(50),
    salary FLOAT DEFAULT 30000,
    codeD INTEGER,
    FOREIGN KEY(codeD) REFERENCES Department (codeD) ON DELETE CASCADE
)
```

A Zoom on Foreign Keys

Option ON DELETE (UPDATE) NO ACTION.

```
CREATE TABLE Employee
(
    codeE INTEGER PRIMARY KEY,
    first VARCHAR(50),
    last VARCHAR(50),
    position VARCHAR(50),
    salary FLOAT DEFAULT 30000,
    codeD INTEGER,
    FOREIGN KEY(codeD) REFERENCES Department (codeD) ON DELETE NO ACTION
)
```

SQL Queries

Query in SQL

SELECT C_1, C_2, ..., C_n
FROM R_1, R_2, ..., R_k
WHERE P

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

SELECT . . . FROM

- SELECT * returns all columns.

Select all employees

```
SELECT *
FROM Employee
```

- Alternatively, we can just select specific columns.

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

SELECT . . . FROM . . . WHERE

Select all employees whose salary is higher than 60,000

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

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

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

SELECT . . . FROM . . . WHERE

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
```

Aggregating Functions

- Perform computations on a **group** of rows.
- If the clause GROUP BY is not specified, the group of rows is the whole table.

Return the maximum salary.

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

Return the minimum salary.

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

Aggregating Functions

- Perform computations on a **group** of rows.
- If the clause GROUP BY is not specified, the group of rows is the whole table.

Return the average salary.

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

Return the sum of the salaries.

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

Aggregating Functions

- Perform computations on a **group** of rows.
- If the clause GROUP BY is not specified, the group of rows is the whole table.

Counts the number of employees.

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

Return the number of family names in the table Employee, ignoring the duplicates

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

Aggregating Functions — GROUP BY

- Partitions the table into **non-overlapping groups**.
- The aggregating 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
```

Aggregating Functions — HAVING

Return the average salary of an employee 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
```

- The clause HAVING specifies search conditions on groups of tuples.
- A combination of Boolean predicates can be used in the HAVING clause.
  - Each predicate **must** use an aggregating function.

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 counselour	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 (usually on the foreign key).
- Costly operation when the two tables have a lot of rows.

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.
- The syntax can become cumbersome.

```
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
```

Left Outer Join

- The following query will only return employees who are member 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 **left** table of the join operation.

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

Right Outer Join

- The following query will only return departments that have at least one employee.
- What about the others?

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

- A **right outer join** will complete the table with all the data in the **right** table of the join operation.

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

Full Outer Join

- The union of a left outer join with a right outer join is a **full outer join**.

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

- MySQL does not support FULL OUTER JOIN, but we can emulate it with the operator UNION.

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

Subqueries

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

```
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 FROM clause.

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

Subqueries

```
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 some 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 in a table.

```
UPDATE department
SET nameD="new department"
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
```