Complex Systems Theory and Practice

Lecture 1

Piotr Syga

2021-10-08

Course Outline

Main goal: Show how to select the manner in which the information is stored and how to deliver important data do the app

Selected topics:

- SQL and relational data bases
- Designing database systems
- Database normalization
- NoSQL
- Security and privacy of the information

Course Goal

- Showing the value of information and gains of a proper information processing
- Showing the practical side of the storing data, data processing and analysis
- Pointing out the main differences between relational and non-relational databases
- Allowing the students to produce a system with large amount of data operations – from the design to the implementation and testing

Literature

- J.D. Ullman, J. Widom, A First Course in Database Systems.
- J.D. Ullman, Principles of Database Systems.
- G. Harrison, Next Generation Databases: NoSQL and Big Data.
- E. Redmont, Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement.
- A. Petrov, Database Internals: A deep-dive into how distributed data systems work.
- M.J. Hernandez, Database Design For Mere Mortals A Hand-on Guide to Relational Database Design.
- H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems The Complete Book.
- D. Sullivan, NoSQL for Mere Mortals.
- T. Hills, NoSQL and SQL Data Modeling: Bringing Together Data, Semantics, and Software.
- S. Chellappan, D. Ganesan, MongoDB Security.
- C. Dwork, A. Roth, The Algorithmic Foundations of Differential Privacy.

Grading

default – approx. 4 lists + designing and implementing small DB app (individually or in pairs). All lists are graded, with a -0.5 penalty for each (started) fortnight after the deadline. To pass the class one must solve all the lists (even if after the deadline).
The final grade is a (rounded) mean of all the grades.

Grading

project – full-semester project of designing and implementing a DB app (in groups of at most 4 people) consisting of large number of records, the app has to face a selected problem (e.g., distributed database with failing nodes, handling large number of simultaneous users, efficient operations during large batch inputs/selects/..., efficacy on severely constrained devices). 3 milestones: at the time of the deadline of the first list (m_1) – the design of the application, main goals, functionalities, the stated problem and ideas how to overcome it, selected technology; midterm (m_2) – verifying the progress, testing part of the application on mock data, verifying interactions between the tables, coping with the problem; final presentation (m_3) – deadline like for the small apps.
The final grade: $\mathrm{Rnd}(0.2m_1 + 0.3m_2 + 0.5m_3)$.

The motivation

- the importance of the information
- how to store the data
- how to (effectively) process the data

Storing the data

We need

- data representation – e.g., data as a set
- the operations for extracting specific piece of information both: selection of specific objects from the set and specific properties of the objects
- a manner of comparing and aggregating the knowledge
- rules describing the order and constraining the possibilities of the operations

Storing the data



$\Rightarrow$

(white, 400.0, 48)

(yellow and black, 65.0, 100)

(brown, 1.5, 57)

(red, 23000.0, 160)

$\Rightarrow$

color	weight	speed
string	float	int

Operations

Having a set $\mathcal{A}$ we want to define such operations $\mathtt{Op}_1(\cdot)$, $\mathtt{Op}_2(\cdot)$,... that applied to $\mathcal{A}$ produce some sets $\mathcal{A}'$, $\mathcal{A}''$, ...

- mathematical model – Relational algebra, e.g., $\sigma_{\text{weight}>200}(\mathcal{A})$, $\pi_{\text{color}}(\mathcal{A})$, $\pi_{\text{speed}}(\sigma_{\text{weight}<3 \vee \text{weight}>150}(\mathcal{A}))$
- SQL:
  - a typed (all types are ordered), non-procedural programming language
  - included RA-based optimizer
  - memory management
  - additional procedural extensions and further functionalities in DBMS

Base of a query

Schema of a simple query

Select *columns*
From *table*
Where *condition*

Base of a query

Schema of a simple query

Select *columns*
From *table*
Where *condition*

Example

```
Select  Speed
From    A
Where   weight>150;
```

Base of a query

Schema of a simple query

Select *columns*
From *table*
Where *condition*

Example

```
Select Distinct  speed
From             A
Where            weight>150;
```

What we can SELECT?

'Displaying' the result

- entire row (Select *)
- a single column (Select speed)
- a group of columns (Select speed, color)
- a constant (Select 1)
- arithmetic operations on columns (Select 3*speed+weight-4)
- aggregations (Select MAX(speed))
- few more (string operations, 'custom' functions, . . .)

FROM where do we select?

Data sources

- a table or group of joined tables
- a derived table/a subquery
- a view

FROM where do we select?

JOINing tables

`SELECT` *columns* `FROM A JOIN B (ON` *condition*`)`

JOIN types:

- `CROSS JOIN` (,) — Cartesian product of all the tuples
- `INNER JOIN` — joining only the tuples that follow the condition (both in A and B)
- `LEFT` (outer) `JOIN` — all tuples from A are extended by B's as in `INNER JOIN` or by `NULL`
- `RIGHT` (outer) `JOIN` — all tuples from B are extended by A's as in `INNER JOIN` or by `NULL`
- `FULL JOIN` — records look like in `LEFT` and `RIGHT`, joining only the tuples that meet the condition

Conditions

Logical operators

- Basic comparison: `=`, `<>` (`!=`), `>`, `<`, `<=`, `>=`
- Conjunctions `AND`, `OR`, `NOT`
- `BETWEEN` a `AND` b
- `LIKE` '%*pattern*_'
- (`NOT`) `IN` *set*
- `IS` (`NOT`) `NULL`
- additional (dialect dependent) features (e.g., REGEX)

Conditions

Logical operators

- Basic comparison: `=`, `<>` (`!=`), `>`, `<`, `<=`, `>=`
- Conjunctions `AND`, `OR`, `NOT`
- `BETWEEN` a `AND` b
- `LIKE` '%*pattern*_'
- (`NOT`) `IN` *set*
- `IS` (`NOT`) `NULL`
- additional (dialect dependent) features (e.g., REGEX)

e.g., `SELECT` speed `FROM` A `WHERE` color `LIKE` '%and%';

Set operations and merging results

Set operations

- `UNION` — $\cup$
- `INTERSECT` — $\cap$
- `EXCEPT` — $\setminus$ (dialect dependent, some require `Where` column `NOT IN`)

Important: the schemas (the number, type and names of the columns) have to be the same

Subqueries

We can merge results with subqueries and derived tables:

```
SELECT color FROM (
SELECT color, speed FROM A WHERE weight <> 400 ) TAB1 WHERE speed
<100;
SELECT color FROM A WHERE weight <> (
SELECT weight FROM A WHERE speed=48);
```

Aliases

- We can alias a selected column to change its name (for presentation reasons or to agree on a common scheme)
- We can alias a table to change its name (e.g., when using two copies of the same table)
- We **have to** alias a derived table (the entity from which we select has to be named)

```
SELECT T1.speed AS s, T2.color
FROM A (AS) T1 JOIN A (AS) T2 ON T1.color=T2.color;
```

Views

The result of `SELECT` queries are not 'saved' by default, neither are the aliases. If we want to reuse the query we have to use some way of preserving the derived tables we may use CTE or a View.

- Views simplify the structure of a query
- We can treat it as an interface – backwards compability of the code even when the structure of the tables changes.
- Views may be easily used to limit the records or the columns, the user has access to.

```
CREATE (algorithm) VIEW name (columns) AS
SELECT ...
```

Important: There are some constraints on the things we may use in a view (subqueries, variables, dynamic queries, triggers, some functions and procedures).

More than a set.. – Ordering and result manipulation

As a general rule we want to interpret a table as a set, yet it may not be true in some specific cases, in particular after a query involving projection. Hence the result of the query is not a set:

sorting — `ORDER BY` *column*(, *column2*, . . .) (`DESC`)

grouping — `GROUP BY` *column*(, *column2*, . . .) – we use aggregation functions so that columns not being grouped may be replaced by a single entry

filtering — `HAVING` *condition* – allows filtering **after** the query, hence filtering by aggregated column is possible

row cap — `LIMIT` (`TOP`, `ROWNUM`< number) – **Important:** the place of a cap in the query differs for each capping keyword

Aggregation

- `MIN(`*column*`)`, `MAX(`*column*`)`, `SUM(`*column*`)`
- `AVG(`*column*`)`, `VARIANCE(`*column*`)`, `STD(`*column*`)`
- `COUNT(`*column*`)`, `COUNT(`*column, column2*`)`, `COUNT(*)`
- `GROUP_CONCAT(`*column*`)`
- bit operations (logical and shifts), file operations, ...