Databases

TDA357/DIT622– LP3 2024

Lecture 3

Ana Bove

(much of the material is based on material from
both Thomas Hallgren and Jonas Duregård)

January 18th 2024

Recall Last Lecture

- **SQL** DDL (*data definition language*): Create, delete and alter database tables;
- Types, constraints and patterns;
- Relational schemas;
- **SQL** DML (*data manipulation language*): Insert, update and delete data in tables;
- Database queries involving one table;
- Group-by and aggregations.

Overview of Today's Lecture

- More **SQL**:
  - Local definitions;
  - Views;
  - Set operations (unions, intersections, excepts);
  - Databases with several tables;
  - Database queries involving several tables;
  - Cross products and joins;
  - In and not in, Exists and not exists;
  - Dealing with and avoiding empty values.
- We leave Foreign keys for next lecture.

Our Running Database

- Relation schema:

  Countries (<u>name</u>, abbr, capital, area, population, continent, currency)
  Unique abbr
  population $\geqslant 0$

- **SQL**:

```
CREATE TABLE Countries (
  name TEXT PRIMARY KEY,
  abbr CHAR(2) NOT NULL UNIQUE,
  capital TEXT NOT NULL,
  area FLOAT NOT NULL,
  population INT NOT NULL CHECK (population >= 0),
  continent CHAR(2) NOT NULL,
  currency CHAR(3) );
```

Local Definitions

- Creating a table on the fly:

```
SELECT *
FROM (SELECT name, CEIL(population/area) AS density
      FROM Countries) AS Densities
ORDER BY density DESC
LIMIT 5;
```

- Using WITH statement:

```
WITH Densities AS
     (SELECT name, FLOOR(population/area) AS density
     FROM Countries)
SELECT *
FROM Densities
ORDER BY density DESC
LIMIT 5;
```

Views

We can give names to a query and then use it as if it *were* a table.

- Creates/replaces a view:

```
CREATE VIEW Densities AS
  (SELECT name, ROUND(population/area) AS density FROM Countries);
```

```
CREATE OR REPLACE VIEW Densities AS
  (SELECT name, ROUND(population/area) AS density, abbr FROM Countries);
```

- Uses the view as if it were a table:

```
SELECT name FROM Densities
ORDER BY density DESC LIMIT 3;
```

```
SELECT name, abbr FROM Densities
ORDER BY density ASC LIMIT 3;
```

- Removes a view:

```
DROP VIEW Densities;
```

Materialized Views

- Creates a special kind of table with the result of a query reflecting only the data that is in the original table when the view was created:

```
CREATE MATERIALIZED VIEW MDensities AS
  (SELECT name, ROUND(population/area) AS density FROM Countries);
```

- When the data in the original table changes, the data in the view needs to be updated; materialized views are NOT automatically updated:

```
REFRESH MATERIALIZED VIEW MDensities;
```

- This might require to recompute the whole query when a table is updated. If updates are more frequent than selections, the materialized view will be less efficient than a virtual view.
- Removes a materialized view:

```
DROP MATERIALIZED VIEW MDensities;
```

Set Operations in Relational Databases: UNION

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

The tables need to have comparable types!

- Selects currencies in America and Europe (set union!):

```
SELECT currency FROM Countries WHERE continent = 'AM'
UNION
SELECT currency FROM Countries WHERE continent = 'EU'
```

- Same query with (possible) repetition in the result:

```
SELECT currency FROM Countries WHERE continent = 'AM'
UNION ALL
SELECT currency FROM Countries WHERE continent = 'EU'
```

- Compare with (resp.):

```
SELECT DISTINCT currency FROM Countries
  WHERE continent IN ('AM', 'EU');
SELECT currency FROM Countries WHERE continent IN ('AM', 'EU');
```

More Examples

- Selects the maximum and minimum population:

```
SELECT MAX(population) AS max_min_pop FROM Countries
UNION
SELECT MIN(population) AS max_min_pop FROM Countries;
```

- Classifies countries into small and big:

```
SELECT name, 'small' AS size FROM Countries WHERE area < 300000
UNION
SELECT name, 'big' AS size FROM Countries WHERE area >= 300000;
```

Set Operations in Relational Databases: INTERSECT

$$A \cap B = \{x \mid x \in A \text{ and } x \in B\} \qquad \text{OBS: } A \cap B \subseteq A, B$$

The tables need to have comparable types!

- Selects common sets of country names and capitals (set intersection!):

```
SELECT name AS place FROM Countries
INTERSECT
SELECT capital AS place FROM Countries;
```

Compare with this query, are they the same?

```
SELECT name FROM Countries WHERE name = capital;
```

- Keeps duplicates (result of intersect should be included in each subquery!):

```
SELECT name AS place FROM Countries
INTERSECT ALL
SELECT capital AS place FROM Countries;
```

Set Operations in Relational Databases: EXCEPT

$$S - A = \{x \mid x \in S \text{ and } x \notin A\}$$

The selections need to have comparable types!

- Selects all currencies except those that are not Euro (set diference!):

```
SELECT currency FROM Countries
EXCEPT
SELECT currency FROM Countries WHERE continent = AM;
```

- Keeps duplicates:

```
SELECT currency FROM Countries
EXCEPT ALL
SELECT currency FROM Countries WHERE continent = 'AM';
```

- Compare with (resp.):

```
SELECT DISTINCT currency FROM Countries WHERE continent != 'AM';
SELECT currency FROM Countries WHERE continent != 'AM';
```

Example: Complex Query

```
/* This query makes not much sense!
But it is quite complex! :)
*/

WITH
   ManyPeople AS
      (SELECT name, area, 'many' AS size FROM Countries
       WHERE population >= 10000000),
   FewPeople AS
      (SELECT name, area, 'few' AS size FROM Countries
       WHERE population < 6000000)
(-- Removes American countries from FewPeople
 SELECT name, size FROM FewPeople
 EXCEPT
 SELECT name, 'few' FROM Countries WHERE continent = 'AM')
UNION -- puts the results together
(-- Keeps countries with big area from ManyPeople
 SELECT name, size FROM ManyPeople
 INTERSECT
 SELECT name, 'many' FROM Countries WHERE area > 2500000);
```

Adding a New Table to Our Database

Let us now add the table Currencies (and some values) to our database:

Code	Name	Value
SEK	Swedish Krona	1
DKK	Danish Krone	1.36
EUR	Euro	10.17
ARS	Peso Argentino	0.1
UYU	Peso Uruguayo	0.2
USD	Dollar	8.28
BTC	Bitcoin	85634.34

Relation schema and **SQL** definition:

Currencies (<u>code</u>, name, value)

```
CREATE TABLE Currencies (
  code CHAR(3) PRIMARY KEY,
  name TEXT NOT NULL,
  value FLOAT );
```

Queries Involving Several Tables

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries, Currencies;
```

Note: Observe the qualified selections Countries.name and Currencies.name to disambiguate.

Alternatively,

```
SELECT Co.name, code, Cu.name, value
FROM Countries AS Co, Currencies AS Cu;
```

Under the hood:

- The full cartesian product (*cross product*) of both tables is generated (size $|\text{Countries}| \times |\text{Currencies}|$);
- The desired columns are selected.

Queries Involving Several Tables: New Attempt

```
SELECT Co.name AS country, code, Cu.name AS currency, value
FROM Countries AS Co, Currencies AS Cu
WHERE currency = code;
```

```
SELECT Co.name AS country, code, Cu.name AS currency, value
FROM Countries Co, Currencies Cu
WHERE currency = code;
```

Under the hood:

- The full cartesian product (*cross product*) of both tables is generated (size $|\text{Countries}| \times |\text{Currencies}|$);
- The rows satisfying the condition "currency = code" are kept;
- The desired columns are selected.

Cross Product same as Inner Join

These queries are equivalent.

We obtain only the rows with "matching currencies".

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries, Currencies
WHERE currency = code;
```

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries CROSS JOIN Currencies
WHERE currency = code;
```

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries JOIN Currencies ON currency = code;
```

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries INNER JOIN Currencies ON currency = code;
```

Outer Join: RIGHT, LEFT, FULL

- Includes all currencies, even if no countries are using them:

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries RIGHT OUTER JOIN Currencies ON currency = code;
```

```
SELECT Countries.name, currency, Currencies.name, value
FROM Countries RIGHT OUTER JOIN Currencies ON currency = code;
```

- Includes all countries, even if their currencies are not in Currencies:

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries LEFT OUTER JOIN Currencies ON currency = code;
```

```
SELECT Countries.name, currency, Currencies.name, value
FROM Countries LEFT OUTER JOIN Currencies ON currency = code;
```

- Includes info from all rows in both tables:

```
SELECT Countries.name, code, Currencies.name, value
FROM Countries FULL OUTER JOIN Currencies ON currency = code;
```

```
SELECT Countries.name, currency, Currencies.name, value
FROM Countries FULL OUTER JOIN Currencies ON currency = code;
```

Some Simple New Tables

Assume these tables:

```
CREATE TABLE Capitals (
   country TEXT PRIMARY KEY,
   capital TEXT );

CREATE TABLE CurrencyCodes (
   country TEXT PRIMARY KEY,
   currency CHAR(3) );
```

with the following entries:

country	capital
Sweden	Stockholm
Norway	Oslo
France	Paris

country	currency
Norway	NOK
Germany	EUR
Sweden	SEK

Note: Observe that the attribute/column containing the name of the country has the same name and type in both tables.

Natural Join

The joins are based on the columns with the same name.

- Inner joins: only countries that appear in both tables.
  These queries are (almost) equivalent:

```
SELECT * FROM Capitals, CurrencyCodes
WHERE Capitals.country = CurrencyCodes.country;

SELECT * FROM Capitals JOIN CurrencyCodes
ON Capitals.country = CurrencyCodes.country;

SELECT * FROM Capitals JOIN CurrencyCodes USING (country);

SELECT * FROM Capitals NATURAL INNER JOIN CurrencyCodes;

SELECT * FROM Capitals NATURAL JOIN CurrencyCodes;
```

- Outer joins: all capitals vs. all currencies vs. all info.
  The word OUTER is not needed:

```
SELECT * FROM Capitals NATURAL LEFT JOIN CurrencyCodes;

SELECT * FROM Capitals NATURAL RIGHT JOIN CurrencyCodes;

SELECT * FROM Capitals NATURAL FULL JOIN CurrencyCodes;
```

Playing with Outer Join

The following queries give slightly different results:

- Compare

```
SELECT *
FROM Capitals LEFT/RIGHT/FULL OUTER JOIN CurrencyCodes
ON (Capitals.country = CurrencyCodes.country);
```

  with

```
SELECT *
FROM Capitals LEFT/RIGHT/FULL OUTER JOIN CurrencyCodes
USING (country);
```

- Compare

```
SELECT Capitals.country, capital, currency
FROM Capitals FULL OUTER JOIN CurrencyCodes USING (country);
```

  with

```
SELECT CurrencyCodes.country, capital, currency
FROM Capitals FULL OUTER JOIN CurrencyCodes USING (country);
```

IN and NOT IN vs. EXISTS and NOT EXISTS

The following queries are equivalent:

- Selects all currencies used in some country:

```
SELECT code, Currencies.name FROM Currencies
WHERE code IN (SELECT currency FROM Countries);
```

```
SELECT code, Currencies.name FROM Currencies
WHERE EXISTS
    (SELECT * FROM Countries WHERE currency = code);
```

- Selects all currencies not used in any country:

```
SELECT code, Currencies.name FROM Currencies
WHERE code NOT IN (SELECT currency FROM Countries);
```

```
SELECT code, Currencies.name FROM Currencies
WHERE NOT EXISTS
    (SELECT * FROM Countries WHERE currency = code);
```

Something to Take into Account about Empty Values

What is the result of

```
SELECT country FROM CurrencyCodes WHERE currency = 'EUR'
```

```
SELECT country FROM CurrencyCodes WHERE currency = currency
```

when currency is empty? (has value NULL)

Answer: Neither TRUE or FALSE but UNKNOWN!
UNKNOWN is excluded in conditions and treated as FALSE.

Use "x IS NULL" or "x IS NOT NULL" to check if x is empty or not (they always give TRUE or FALSE)!

Three-valued Logic: TRUE, FALSE, UNKNOWN

The truth-table has 9 rows:

p	q	NOT p	p AND q	p OR q
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
TRUE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	UNKNOWN	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN	TRUE
UNKNOWN	FALSE	UNKNOWN	FALSE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN

In general, good to try to avoid empty values but sometimes they might be needed.

Getting Rid of Empty Values in Queries

Note that even if we do not allow empty values in our tables, the result of a query can contain empty values (for example with OUTER JOIN).

COALESCE takes a list of values and returns the first non-empty value.

Useful to replace null values with constants of the appropriate type.

```
SELECT country, COALESCE (capital, 'no capital'),
               COALESCE (currency, 'no currency')
FROM Capitals NATURAL FULL JOIN CurrencyCodes;
```

We can also give a better name to the column:

```
SELECT country, COALESCE (capital, 'no capital') AS capital,
               COALESCE (currency, 'no currency') AS currency
FROM Capitals NATURAL FULL JOIN CurrencyCodes;
```

Overview of Next Lecture

- Foreign keys;
- More about consistency:
  - Policies on referencial contraints;
  - Assertions;
- Summary of **SQL**;
- Summary of relational schemas;
- Example.

Reading:

Book: chapters 2, 6.1–6.5, 7.1–7.4, 8.1–8.2 and 8.5

Notes: chapter 2, 7.4.1–7.4.3 and 4.9