Information Retrieval
WS 2016 / 2017

Lecture 13, Tuesday January 31st, 2017
(Knowledge Bases, SPARQL, Translation to SQL)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg
Overview of this lecture

■ Organizational
  - Your experiences with ES12 linear classification proof

■ Content
  - Knowledge bases + SPARQL explanation + examples
  - Databases + SQL explanation + examples
  - SQLite a lightweight database
  - SPARQL to SQL algorithm + example
  - Performance joins and join order
  - **ES13:** Implement SPARQL → SQL translation and use to process SPARQL queries with Python+SQLite
Experiences with ES12  1/3

Summary / excerpts

- Most of you found the exercise sheet quite easy
  "This was the shortest ex. sheet anyone ever gave to me"
- In particular, the hint provided was quite extensive
- Some of you doubted whether you understood it correctly, because it went so relatively easily
  "It was so easy that we felt unchallenged and didn't solve it"
- For some it was still tricky because they don't like proofs
  "I'm not that good in proving things in a formal way"
- Almost everybody participated in the evaluation!
Electromagnetic waves, wavelengths and characteristics

- γ-rays \(< 10\text{pm}\) decay of atomic nuclei
- X-rays \(10\text{pm} .. 10\text{nm}\) braking radiation fr. electrons
- Ultraviolet \(10\text{nm} .. 400\text{nm}\) 5% of sunlight (energy-wise)
- Visible \(400\text{nm}..700\text{nm}\) 45% of sunlight
- Infrared \(700\text{nm} .. 1\text{mm}\) thermal radiation, 50% of sun
- Microwave \(1\text{mm} .. 1\text{m}\) line-of-sight, narrow beams
- Radio waves \(1\text{m} .. 100\text{km}\) broadcasting, ground waves

- Atmosphere absorbs γ- and X-rays, and most of UV and infrared

Main absorbers: **water vapor**, carbon dioxide, ozone

Note: lower wavelength → higher frequency → more energy
Experiences with ES12  3/3

- Electromagnetic waves
  - So, in a nutshell, our "visible range" is more or less that part of the sunlight, which is not filtered by the atmosphere
  - Understand that what we "see" is actually just "reflection data" (of electromagnetic waves from the 400nm..700nm range)
    It is as weird, as if we could see WiFi → question on ES13
  - Some animals can also see (low wavelength) infrared
    For example, they can then also "see" temperature directly
    However, water is a strong absorber of (low wavelength) infrared, so you can't use a biological organ like the eye
Knowledge Bases and SPARQL 1/7

What is a knowledge base

- A knowledge base is a database of statements about entities and their relations

  Critical: unique identifiers for each entity and relation

- A common format / schema is to express all statements as subject predicate object triples:

  Brad Pitt    acted in    Mr. and Mrs. Smith
  Brad Pitt    acted in    Burn After Reading
  Angelina Jolie acted in  Mr. and Mrs. Smith
  Joel Cohen    directed  Burn After Reading
  Ethan Cohen    directed  Burn After Reading
  Brad Pitt    married to  Angelina Jolie
Freebase and WikiData

- Freebase is the largest open general-purpose KB to date
  
  Started by Metaweb in 2007, acquired by Google in 2010

  Current size: \( \approx \text{3 billion} \) triples on \( \approx \text{60 million} \) entities

  Freebase has become read-only in March 2015 and most of its data will eventually be merged into WikiData

- WikiData is the soon-to-become largest open general-purpose knowledge base to data

  WikiData is the "Wikipedia" among the knowledge bases

  Current size: \( \approx \text{135 million} \) triples on \( \approx \text{25 million} \) entities
Reification

- Restriction to triples is no real restriction: n-ary relationships can also be represented as triples:

  m/0jy6xg film Finding Nemo
  m/0jy6xg actor Ellen DeGeneres
  m/0jy6xg character Dory
  m/0jy6xg type Voice

  m/0jy6xg is an entity name from Freebase

In the example above, it's a so-called mediator, which serves as a link between the entities it connects

The dataset for ES13 has no mediators
Relation to the "Semantic Web"

- The Semantic Web initiative is concerned with making knowledge base data **explicitly** available on the web

  **Variant 1**: semantic mark-up in normal web pages
  
  Typical format: Microdata or JSON-LD

  **Variant 2**: web pages containing only structured data
  
  Typical format: RDF

- No rules that enforce consistent entity or relation names
  
  The hope is that people adhere to standards nevertheless, and that machines can resolve the remaining heterogeneity

**Anyway: this is not** the topic of this lecture / course
What is SPARQL

- The standard query language for knowledge bases

\[ \text{SPARQL} = \text{SPARQL Protocol And RDF Query Language} \]

- Example query in natural language: actors who are married and played together in at least one movie

- The same query expressed in (simplified) SPARQL

```
SELECT ?person1 ?person2 ?film WHERE {
  ?person1 acted_in ?film .
  ?person1 married_to ?person2
}
```
SPARQL syntax

- In the lecture today, we use a simplified syntax

  In "real" SPARQL, names of subjects / predicates / objects may contain whitespace and are surrounded by <...>

- The actual SPARQL syntax is slightly more complicated and has many more features

  In particular, it involves namespace prefixes, so that names can be made globally unambiguous

  See the Wikipedia page or the W3C specification if you are interested

  Not relevant for our lecture today
SPARQL queries as subgraphs

- One can view a knowledge base as a graph, where the nodes are the entities, and the edges are the relations.
- A SPARQL query is then a sub-graph with variables at some or all of the nodes.
- Solving the query then amounts to finding all matches of the subgraph in the (large) knowledge base graph.
Databases and SQL  1/4

- Introduction
  - Data from a knowledge base can also be stored in an ordinary database
    This is also what we do in the lecture and for ES13
  - The standard query language for databases is SQL

  **SQL = Structured Query Language**

  - On the following slides, let us recap the basics from databases and SQL via a few examples
What is a database

- For this lecture, a database is a collection of tables, where each table has a fixed number of columns.
- For example, we could have one table for each predicate from our knowledge base, with two columns each.

Table for "acted in" predicate

\[
\begin{array}{ll}
\text{actor} & \text{film} \\
\hline
\text{Brad Pitt} & \text{Burn after Reading} \\
\text{Angelina Jolie} & \text{Mr. and Mrs. Smith} \\
\ldots & \ldots
\end{array}
\]

Table for "married to" predicate

\[
\begin{array}{ll}
\text{person1} & \text{person2} \\
\hline
\text{Brad Pitt} & \text{Angelina Jolie} \\
\text{Ellen DeGeneres} & \text{Portia de Rossi} \\
\ldots & \ldots
\end{array}
\]

For ES13, you should work with one table (for the whole database) with three columns (subject, predicate, object).
SQL example 1

- Example query FROM **one** table

```sql
SELECT actor
FROM acted_in
WHERE film = "Burn After Reading";
```

**In words:** all actors from movie "Burn After Reading"

**Principle:** select those rows from the specified table which satisfy properties specified in WHERE clause
SQL example 2

- Example query FROM multiple tables

```
SELECT married_to.person1, married_to.person2
FROM married_to, acted_in AS acted1, acted_in AS acted2
WHERE married_to.person1 = acted1.actor
AND married_to.person2 = acted2.actor
AND acted1.film = acted2.film;
```

In words: all couples which acted in the same movie

- Principle: selects items from cross-product $T_1 \times \cdots \times T_k$
  which satisfy properties specified in WHERE clause

- Syntax: use AS for unique names of copies of same table;
  use table.column to refer to that column from that table
A full-fledged database, easy to install and use

- On Debian/Ubuntu install with: `sudo apt-get install sqlite3`
- Two types of commands ... examples on next slides
  
  **SQL** commands: must end with a semicolon
  
  **SQLite** commands: start with a dot, no semicolon at end

- Two modes to start SQLite:

  `sqlite3` will work on an in-memory database
  
  `sqlite3 <name>.db` create database in that file, and if file exists, use database from that file

Let's read our example tables in SQLite using the commands from the next two slides ... it's easy
Some useful **SQLite** commands by example

- Specifies the column separator used for input and output
  `.separator " " `use Ctrl+V TAB for TAB`

- Read table from TSV (tab-separated values) file
  `.import film.tsv film`

- Execute commands from script file (typical suffix is `.sql`)
  `.read <file with commands>`

- Show execution time of every command
  `.timer on`
Some useful SQL commands by example

- Create a table with a given schema
  ```sql
  CREATE TABLE acted_in(actor TEXT, film TEXT);
  ```
- Create an index for a column of a table
  ```sql
  CREATE INDEX acted_in_index ON acted_in(actor);
  ```
- Extract / combine data from tables
  ```sql
  SELECT * FROM acted_in WHERE ... LIMIT 100;
  ```
- Delete table / index (without error msg if it's not there)
  ```sql
  DROP TABLE IF EXISTS acted_in;
  DROP INDEX IF EXISTS acted_in_index;
  ```
Python interface to SQLite

- Executing SQL commands to a SQLite database from within Python is very easy:

  ```python
  import sqlite3
db = sqlite3.connect("example.db")
cursor = db.cursor()
cursor.execute("SELECT * FROM table")
for row in cursor.fetchall():
    print("\t".join(row))
  ```

Beware: the SQLite commands (starting with a dot) cannot be executed from within Python, you need SQLite for those
Motivation

- We want to translate a given SPARQL query to a SQL query that gives the desired results on a given database.
- In the following example, we use one table per relation.

```sql
CREATE TABLE acted_in(actor TEXT, film TEXT)
CREATE TABLE married_to(person1 TEXT, person2 TEXT)
```

Note: all elements from one table are from one relation, so we don't need to store the relation name in the table.

For ES13, use **one big table** for all the data, with three columns named **subject, predicate, object**.

This is deliberately different from how we did it in the lecture, so that you have to do some thinking yourself.
Example

- **SPARQL query**

  ```sparql
  SELECT ?p1 ?p2 ?f WHERE {
    ?p1 acted_in ?f .
    ?p1 married_to ?p2
  }
  ```

- **SQL query**

  ```sql
  SELECT married_to.person1, married_to.person2, acted1.film
  FROM married_to, acted_in AS acted1, acted_in AS acted2
  WHERE married_to.person1 = acted1.actor
  AND married_to.person2 = acted2.actor
  AND acted1.film = acted2.film;
  ```
Algorithm

- It is up to you in ES13, to design a generic algorithm that works for arbitrary basic SPARQL queries of the form SELECT <vars> { <triples> }

- The algorithm is not difficult, but requires understanding of how the data is stored and how SPARQL and SQL work

  So perfect exercise to understand the basics!

- On the next slide we give you some valuable advice.
Algorithm, advice for ES13

- If there are $k$ query triples in the SPARQL query, have $k$ entries in the FROM clause of the SQL query

  FROM freebase as f1, freebase as f2, ..., freebase as fk

- In your code, for each variable from the SPARQL query, build an array of all its occurrences in the query, e.g.

  ?x: f1.subject, f2.object, f5.object

- Then, when building the SQL query, add the corresponding equalities to the WHERE clause, e.g.

  WHERE f1.subject = f2.object AND f2.object = f5.object

Note: if ?x occurs $m$ times, $m - 1$ equalities are enough
Performance 1/4

- Cross product of tables

  - Recall that, conceptually, an SQL statement like

    \[
    \text{SELECT ... FROM } T_1, T_2, ..., T_k \text{ WHERE ...}
    \]

    selects elements from the **cross-product**

    \[
    T_1 \times \cdots \times T_k \quad \text{(which has } |T_1| \cdot \cdots \cdot |T_k| \text{ elements)}
    \]

    (where some or all of the $T_i$ can be the same table)
Joining of tables

- Each ... = ... in the WHERE clause effectively ask for a JOIN operation between two tables
- Algorithmically, a JOIN requires a list intersection
- If we CREATE an index for the respective tables on the respective join attributes, this list intersection gets fast

E.g., by sorting (a copy of) the table by that attribute
Join ordering

- Typical SQL-from-SPARQL queries require multiple joins
- Order of joins can make a huge performance difference
- For our example query, the \textit{acted\_in} table (actors – films) is more than ten times larger than the \textit{married\_to} table
- \textbf{Join order 1}: look at all pairs of actors who played in the same film, and for each check whether they are married
  materialized all pairs of actors from same film (large)
- \textbf{Join order 2}: look at all married couples and for each get their films and check whether they overlap
  materializes list of films of all married people (small)
Join ordering, continued

Without further ado, SQLite seems to take the order of the tables in the FROM clause as its join order:

```sql
SELECT married_to.person1, married_to.person2
FROM acted_in as acted1, acted_in as acted2, married_to
WHERE married_to.person1 = film1.actor
  AND married_to.person2 = film2.actor
  AND acted1.film = acted2.film;
```

Alternatives: (note that there are 6 possible orderings)

```sql
FROM married_to, acted_in as acted1, acted_in as acted2
FROM married_to, acted_in as acted2, acted_in as acted1
```
References

- **Textbook**
  - Nothing about this topic in the text book by Manning, Raghavan, and Schütze

- **Wikipedia**