Information Retrieval

WS 2013 / 2014

Lecture 12, Tuesday January 28th, 2014

(Ontologies, SPARQL, Relation to SQL)

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

UNI FREIBURG

Overview of this lecture

- Organizational
  - Your results + experiences with Ex. Sheet 11 (NB vs. SVM)
- Ontologies
  - Ontologies = fact databases ... ask questions like:

    Actors that are married and starred in the same movie
  - The SPARQL ontology query language
  - Translate SPARQL queries to SQL queries on a database
  - Performance issues (join order)
  - **Exercise Sheet 12:** Implement SPARQL → SQL translation and use to process SPARQL queries with the SQLite database

Experiences with ES#11 (NB vs. SVM)

- Summary / excerpts   last checked January 28, 14:00
  - Exercise helped to understand concepts more deeply
  - The theoretical task wasn't as hard as it first looked
  - Ambiguous notation: the $w_1, \ldots, w_m$ denoted both the words from the vocabulary and the m components of the w vector

    Sorry, will be fixed next time I give the course
  - Small mistake on slide 12 of last lecture: $Pr(b|A) \rightarrow Pr(a|B)$
  - Vector representation was different from that for ES#10

    But the same as in all other lectures before that, and anyway, the two representations are equivalent (slide 9 of last lecture)
  - Thanks again for the great feedback from the tutor

Your results for ES#11 (SVM vs. NB)

- Summary
  - **Dataset 1:** actors and politicians, 18,499 documents

    Accuracy of 98% for both SVM and NB
  - **Dataset 2:** singers and songwriters, 8,913 documents
  - Accuracy of 92% for SVM and of 89% for NB
  - On both datasets zero outliers for the SVM and a much larger margin than for NB (which does not care about margin size)

    However, this does not seem to matter much for accuracy
  - Experience shows that NB typically **estimates badly** the Prob(C=c | doc), but nevertheless often **classifies well**

    In practice, one is often not interested in accurate probabilities, but just that the correct class gets the highest probability

Ontologies 1/5

- **Ontology** = a database of **facts** on **entities**
  - With **unique** names / identifiers for each entity
  - Facts are expressed 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

Understand: we can always decompose more complex facts into triples, so triples is all we need

Ontologies 2/5

- Relation to the "Semantic Web"
  - The classical web contains a lot of facts hidden in text

    For example: infos about an actor or a movie on IMDB
  - The Semantic Web (SW) initiative is concerned with making ontology data **explicitly** available on the web
  - The challenges of SW are really about standardization:

    Unique identifiers ... use URIs + namespaces

    Diff. identifiers meaning the same thing ... use owl:sameAs

    Well-defined syntax ... RDF has become common
  - This is **not** the topic of this lecture / course

Ontologies 3/5

- Example 1: the GeoNames ontology
  - Very complete database of geographical features:

    Cities, countries, rivers, mountains, roads, ...
  - Around 10M entities, 250MB compressed
  - Download from http://www.geonames.org
  - RDF endpoint: http://www.geonames.org/ontology

Great dataset, but for this lecture we want something more general-purpose ...

Ontologies 4/5

- Example 2: the YAGO ontology (Yet Another Great Ontology)
  - From Suchanek et al, WWW 2007 & J.Web.Sem 2008
  - General-purpose facts, extracted from Wikipedia + WordNet
  - Original dataset: about 120M facts on 10M entities

    Of those, only about 10M are real "facts" that we as humans would find useful ... this is typical for ontologies
  - Download from http://www.mpi-inf.mpg.de/yago
  - Accuracy is good, but many popular facts are missing, e.g. only very few actors per movie are known

    Nevertheless, small and simple and was hence quite popular with researchers (including us) for a while ...

Ontologies 5/5

- Example 3: the FreeBase Ontology
  - A general-purpose ontology, **community-maintained**
  - Developed by Metaweb, aquired by Google in 2010
  - Freely available: https://developers.google.com/freebase/data

    Currently 2500M facts on 50M entities, 25GB compressed

    Rather complex schema + some inconsistencies
  - Nicer version: http://freebase-easy.cs.uni-freiburg.de

    Around 250M facts on 50M entities, 2.5GB compressed

    The currently most complete and most accurate general-purpose ontology ... we extracted a nice subset for you !

SPARQL 1/5

pronounced "Sparql" ← K

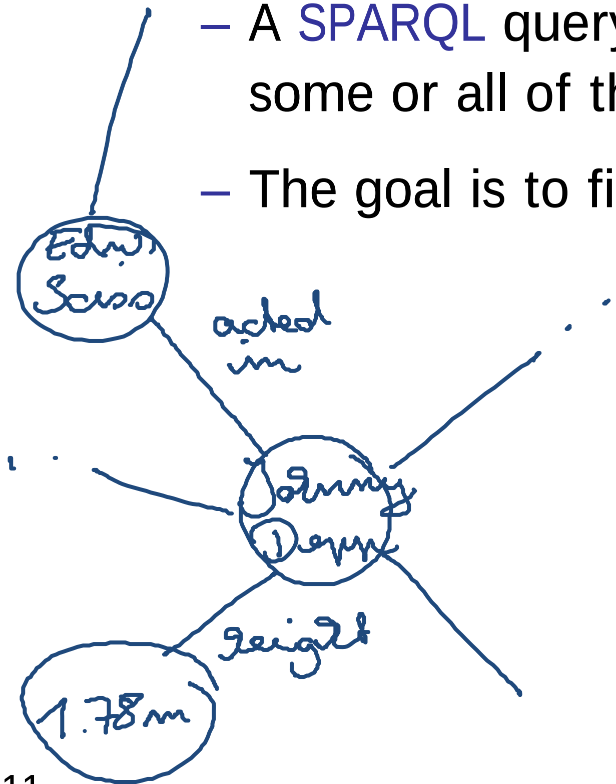
- Structured queries on ontologies
  - Example query in natural language: actors who are married and starred together in at least one movie
  - Difference between ontology search and text search

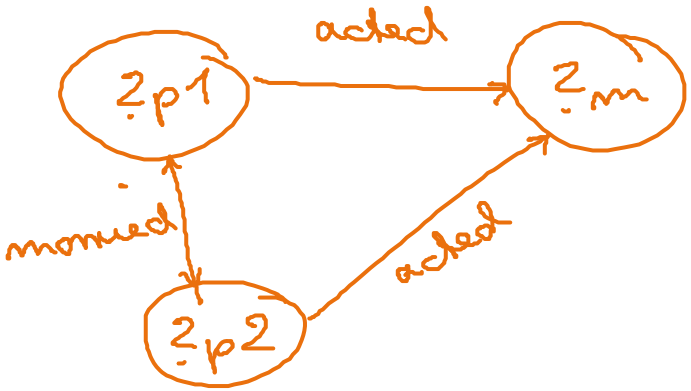
    There is a well-defined result set ... no fuzzy "relevance"
  - **SPARQL** = **S**PARQL **P**rotocol **A**nd **R**DF **Q**uery **L**anguage
  - The standard query language for ontology queries

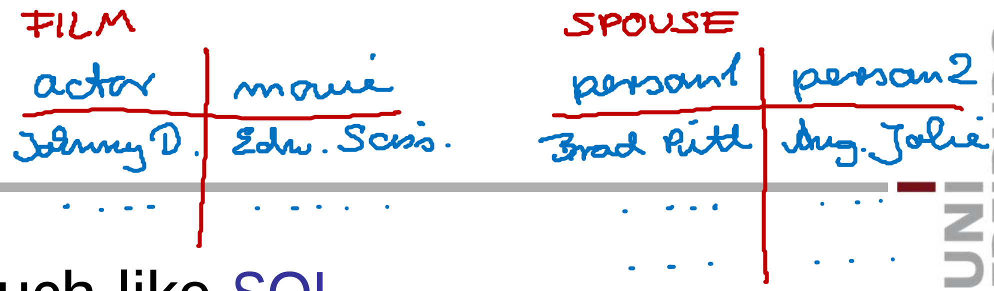
```
SELECT ?person1 ?person2 ?movie WHERE {
  ?person1  acted_in  ?movie  .
  ?person2  acted_in  ?movie  .
  ?person1  married_to  ?person2
}
```

SPARQL 2/5

- Viewing SPARQL queries as subgraphs
  - On can view a (triple) ontology as a **graph**, where the nodes are the entities, and the edges are the facts
  - A SPARQL query is then a sub-graph with variables at some or all of the nodes
  - The goal is to find all matches in the ontology graph







- SPARQL looks very much like SQL
  - Indeed, ontology data is naturally stored in databases
  - The standard query language for **databases** is SQL
  - Assume we have two tables film (with columns actor and movie) and spouse (with columns person1 and person2)

```
SELECT  spouse.person1, spouse.person2
FROM    spouse, film as film1, film as film2
WHERE   spouse.person1 = film1.actor
AND     spouse.person2 = film2.actor
AND     film1.movie = film2.movie;
```

Let's play around a bit with **SQLite** ... see slides 15 - 17 film

subject	object
...	...

spouse

subject	object
...	...

- SPARQL to SQL: generic translation
  - In the following example, we use one table per relation, each with two columns, just named subject and object

    For ES#11, use **one big table** for all the data, with three columns named **subject**, **predicate**, **object**

SPARQL:

```
SELECT ?p1, ?p2, ?m
WHERE {
    ?p1  film    ?m .
    ?p2  film    ?m .
    ?p1  spouse  ?p2
}
```

SQL:

```
SELECT f1.subject, f2.subject, f1.obj.
FROM film as f1, film as f2,
     spouse as s
WHERE
     f1.object = f2.object
AND  f1.subject = s.subject
AND  f2.subject = s.object
```

For ES#11, you also need to specify the predicate in the WHERE clause of the SQL query

SPARQL 5/5

- SPARQL to SQL: implementation advice for ES#11
  - 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

SQLite 1/3

- A full-fledged database, easy to install and use
  - Download from http://www.sqlite.org
  - 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

SQLite 2/3

- 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
  - Show execution time of every command

    .timer on
  - Output to file (use stdout for output to console again)

    .output <file name>
  - Execute commands from script file (typical suffix is .sql)

    .read <file with commands>

SQLite 3/3

- Some useful SQL commands by example
  - Create a table with a given schema

    CREATE TABLE film(actor TEXT, movie TEXT);
  - Create an index for a column of a table

    CREATE INDEX file_index ON film(actor);
  - Extract / combine data from tables

    SELECT * FROM film WHERE ... LIMIT 100;
  - Delete table / index (without error msg if it's not there)

    DROP TABLE IF EXISTS film;

    DROP INDEX IF EXISTS film_index;

Performance 1/4

- Cross product of tables
  - Understand that, conceptually, an SQL statement like

    FROM T_1, T_2, ..., T_k WHERE ... = ... AND ... = ... AND ...

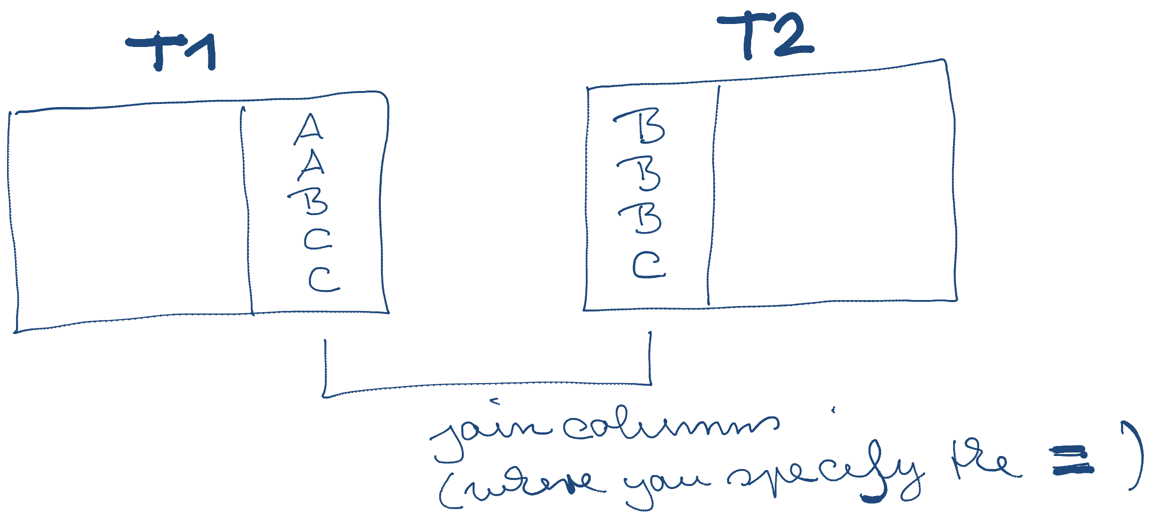
    selects elements from the cross-product

    $T_1 \times \cdots \times T_k$ (which has $|T_1| \cdot \cdots \cdot |T_k|$ elements)

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

Performance 2/4

- Joining of tables
  - The WHERE ... = ... effectively ask for a JOIN
  - This JOIN effectively asks for a **list intersection**
  - If we CREATE an index for the respective tables on the respective join attributes, this list intersection gets fast



Performance 3/4

- Join ordering
  - Typical SQL-from-SPARQL queries require multiple joins
  - Order of joins can make a **huge** performance difference
  - For our example query, the film table (actors – movies) is more than ten times larger than the spouse table
  - Join order 1: look at all married couples and for each get their movies and check whether they overlap

    materializes list of movies of all married people (small)
  - Join order 2: look at all pairs of actors who played in the same movie, and for each check whether they are married

    materialized all pairs of actors from same movie (large)

Performance 4/4

- Join ordering, continued
  - Without further ado, SQLite seems to take the order of the tables in the FROM clause as its join order

Times with INDICES build for all tables and columns.

```
SELECT  spouse.person1, spouse.person2
FROM    film as film1, film as film2, spouse
WHERE   spouse.person1 = film1.actor
AND     spouse.person2 = film2.actor
AND     film1.movie = film2.movie;
```

15 secs.

$3! = 3 \cdot 2 \cdot 1 = 6$

Alternatives: (note that there are 6 possible orderings)

FROM spouse, film as film1, film as film2 — 8 secs.

FROM spouse, film as film2, film as film1 — 2.5 secs.

References

- Textbook
  - Nothing about this topic in the text book by Manning, Raghavan, and Schütze
- Wikipedia
  - http://en.wikipedia.org/wiki/Ontology_(information_science)
  - http://en.wikipedia.org/wiki/SPARQL
  - http://en.wikipedia.org/wiki/SQL
  - http://en.wikipedia.org/wiki/SQLite
  - http://en.wikipedia.org/wiki/Freebase