UNIVERSITEIT TWENTE.

Data Science [201400174]

Course year 2019/2020, Quarter 2A

DATE
February 4, 2020

EXCERPT

Semi-structured data [SEMI]

TEACHERS
Maurice van Keulen
Christin Seifert
Mannes Poel
Karin Groothuis-Oudshoorn
Elena Mocanu
Faiza Bukhsh
Nicola Strisciuglio

COURSE COORDINATOR
Christin Seifert
Maurice van Keulen

PROJECT OWNERS
Faiza Bukhsh
Karin Groothuis-Oudshoorn
Maurice van Keulen
Elena Mocanu
Mannes Poel
Michel van Putten
Mohsen Jafari Songhori
Luc Wismans

Topic 5

Semi-structured data [SEMI]

5.1 Introduction

There exist several data exchange and knowledge representation standards. This topic teaches the most important standards and skills to manipulate data in these standards: (a) XML and its associated standards SQL/XML, XPath and XQuery for publishing and manipulation with both relational as well as XML databases, (b) JSON storage and manipulation in relational databases, and (c) Semantic Web standard RDF with its associated standards

We will not be able to study all important XML and Semantic Web standards, some of the things we do not cover explicitly are XML namespaces, XSLT, XML Schema for XML, and OWL for Semantic Web. JSON is not well standardised through standards committees..

5.1.1 Global description of the practicum and project

The practical assignments let you practice with manipulating XML, JSON, and RDF data. We use a data set with voyages of the VOC for XML and JSON, and a data set about royal families for RDF.

5.1.2 Study material and tools

For XML and JSON, we use BaseX[1], which is a free and open source Java-based XML database engine which also incorporates JSON-support. It supports XQuery 3.1 and includes a wide variety of functionality such as indexing, full-text querying, ACID transactions, and many interfaces such as REST/RESTXQ, WebDav, etc. It also incorporates a rather nice GUI.

For RDF, we use the Apache Jena toolset[2] which is a free and open source Java framework for building semantic web and Linked Data applications. The framework is composed of different APIs interacting together to process RDF data. It includes a SPARQL query engine, a persistent triple store, a server, reasoning functionality, etc.

[1] http://basex.org
[2] https://jena.apache.org/

5.1.3 Deliverables and obligatory items

Topic teacher: Maurice van Keulen

Collect the answers to all questions of all assignments in one PDF-file and submit this file to Canvas.

5.2 Description of the practical assignments

5.2.1 Assignment 1: SQL/XML

Goal Transform relational data to XML with SQL/XML, and brush up prerequisite knowledge on SQL.

Preparation

1. Watch Stanford's prof. Jenifer Widom's lectures on SQL, if it's been a while since you studied it at: `https://www.youtube.com/playlist?list=PLroEs25KGvwzmvIxYHRhoGTz9w8LeXek0`.
2. To study the basics of XML, follow David Malan's (Harvard) XML lecture at: `http://youtu.be/n0MP0oLD_Xk`.
3. Study the paper by Andrew Eisenberg (IBM) and Jim Melton (Oracle) *SQL/XML is Making Good Progress*, ACM SIGMOD Record 31(2), 2002, available from: `http://www.sigmod.org/publications/sigmod-record/0206/standard.pdf`
4. Download the movie database, `movies.sql` from Canvas.

Installation PostgreSQL

There are two options for obtaining a PostgreSQL server: install PostgreSQL yourself, see: `http://postgresql.org`, or use the department's database server.

For the second option, you can obtain a database for your group as follows (only one person of the group needs to do this to obtain a database for the whole group).

- Go to DAB: `http://bronto.ewi.utwente.nl/dab`
- If you do not have an account in DAB, create one with "Register here". You need to use your student email address.
  NB: there is a known bug with the system that it sometimes produces an error if your click on "Register here". The problem is that there is a "dab" missing in the URL: please change it from `http://bronto.ewi.utwente.nl/register` to `http://bronto.ewi.utwente.nl/dab/register`
- Sign in and choose the course "ds19202a-semi" which stands for "Data Science 2019/2020 quartile 2A, topic SEMI".
- Fill in your group number and click on "Get credentials".
- A database will be created for you and the system provides you with the credentials: a username (which is the same as the database name) and a password.
- For your convenience, your database already has a *schema* `movies` with the data of `movies.sql` loaded in it.
- You can always return to DAB to look at your credentials again and to reset your database. **Warning**: you loose everything in the database when you reset it, i.e., for all schemas in the database.

The same server also runs a web-based database administration tool, called PhpPgAdmin. You can access it with this link: `http://bronto.ewi.utwente.nl/phppgadmin`. If you login with the credentials you obtained from DAB, you will see your database and the three schemas (there is also a fourth; just ignore it). You can easily create, inspect and drop the tables in your database with this tool[3]

Exercises

☞ **5.1** SQL/XML exercises

[3] You can also use any other *PostgreSQL client* such as PgAdmin (host = bronto.ewi.utwente.nl; port = 5432).

(a) Warming up: Give the old school SQL query that produces three columns: 'name', 'year', and 'rating'.
(b) Give the SQL/XML query that produces three columns, each with an element: `<name>`, `<year>`, `<rating>`.
(c) Give the SQL/XML query that produces a single column, with all three elements `<name>`, `<year>`, `<rating>`.
(d) Give the SQL/XML query that produces a single column, with all three elements `<name>`, `<year>`, `<rating>` nested inside an element `<movie>`. (Bonus: add an attribute 'id' with the movie identifier 'mid'.)
(e) Write an ordinary SQL query that produces four columns, 'name', 'year', 'rating', and 'nr_of_roles', where the last column contains the number of actor roles for each movie. Commit the result to your git repository. Then change the SELECT part of your query, such that it produces lines like the following, such that the result enumerates all roles instead of counting them (and commit this too):
`<movie><name>Jaws</name><year>1975</year><rating>8.2</rating><role>Chief Martin Brody</role><role>Quint</role> ... </movie>`
(f) Write an ordinary SQL query that produces four columns, 'name', 'year', 'rating', 'nr_of_roles', 'nr_or_directors' where the last column contains the number of directors for each movie. Tip: Solution 2e can be done with a GROUP BY clause, but also without. Commit the result to your git repository. Then change the SELECT part of your query, such that it produces lines like the following (and commit this too):
`<movie><name>Jaws</name><year>1975</year><rating>8.2</rating><role>Chief Martin Brody</role><role>Quint</role> ... <director>Steven Spielberg</director> </movie>`

□

5.2.2 Assignment 2: XPath and XQuery

Goal Formulate queries in XPath and XQuery based on natural language questions.

Preparation

1. Watch Jenifer Widom's lectures on XML, XPath and XQuery at `https://www.youtube.com/playlist?list=PLroEs25KGvwzmvIxYHRhoGTz9w8LeXek0`
2. Optionally, practice XQuery by doing an on-line tutorial, for instance at W3Schools: `https://www.w3schools.com/xml/xquery_intro.asp`
3. Download the database `voc.xml`. (The file contains information about the Dutch East India Company, often claimed to be the world's first multinational, that in 17th and 18th century made many in the Netherlands rich at the expensive of Asian countries, specifically today's Indonesia. For more information, see: `https://en.wikipedia.org/wiki/Dutch_East_India_Company`).

Installation BaseX

Install BaseX from: `http://basex.org/`.

Exercises

☞ **5.2** Path queries

(a) Return all all `boatname` elements of all voyages in the document.
(b) Some voyage elements contain an element `hired` which indicates that the boat was hired for that voyage. Return the `boatname` elements that have been hired.

□

☞ **5.3** Predicates

(a) Return the `harbour` elements that are a "destination" of the boat named "BATAVIA".
(b) Return the `voyage` elements for which the boat named "BATAVIA" sailed from or to the harbour "Batavia". (there are 7 such voyages)
(c) Return the `voyage` elements of which the "Willem IJsbrandsz. Bontekoe" was the "master".

(d) A voyage element sometimes contains an element `particulars` with a possible follow-up voyage (`next` element) and a description of some noteworthy facts. Give the voyages that mention the "Cape of Good Hope" in their `particulars` (there are two such voyages).

□

☞ **5.4** Positions

(a) Queries can also be used to check the correctness of the structure of a document. For example, it seems logical that, since each voyage was done by boat, each voyage element has a boatname descendant element. To check this and to inspect a possible violation, write a query that gives the first `voyage` element that has no boat name.

(b) XQuery defines 13 axis steps: 'ancestor', 'ancestor-or-self', 'attribute', 'child', 'descendant', 'descendant-or-self', 'following', 'following-sibling', 'namespace', 'parent', 'preceding', 'preceding-sibling', and 'self'. The notations '.', '/' and '//', are short-hands for axis-steps. Rewrite the following 3 queries, such that they do no longer contain short-hands:

- `doc("voc.xml")/voyage[number = "4144"]/*[4]`
- `doc("voc.xml")/voyage[number = "4144"]//*[4]`
- `doc("voc.xml")/voyage[number = "4144"]/descendant::*[4]`

(c) Master "Jakob de Vries" made 16 voyages. The document does not have them in chronological order. Give the oldest voyage of this skipper (i.e., the first of a sequence ordered by departure).

□

☞ **5.5** Aggregation

(a) In a single query, count the number of voyages in the document, count the number of boats (i.e., unique boat names), and count the number of masters (i.e., unique master names. The result should look as follows: `<totals><voyages> ... </voyages><boats> ... </boats><masters> ... </masters></totals>`

(b) Many boats carried soldiers. Count how many soldiers were carried in total. (No need to known what the elements `one`, `two`, ... mean, so just add them all up for the answer.)

□

☞ **5.6** Grouping

(a) For each master (i.e., unique master name), give his name and the number of voyages he did. (Grouping can also be done without the use of '`group by`'). The results should look as follows: `<master name=" ... " nrofvoyages= ... "/>`

(b) Return the `master` element(s) of the master(s) with the most voyages. (There are 3 masters that did 16 voyages)

□

5.2.3 Assignment 3: Handling JSON data

Goal Load JSON data in an XML database, query it with XQuery and transform it back to JSON.

Preparation

1. We use the same XML database: BaseX
2. Scan through the documentation of the BaseX JSON module: `http://docs.basex.org/wiki/JSON_Module`
3. Download the bata_2014 database, `bata_2014.json` from Canvas. It contains a JSON array containing all tweets from the Batavierenrace of 2014 as collected from the Twitter API. Load it in BaseX in the following manner: `Database` → `New...`, choose file with `Browse`, and press `OK`.

Exercises

☞ **5.7**

(a) Write an XQuery that produces the text of the first tweet.
(b) Write an XQuery that produces the first tweet as a JSON object.
Tips: construct a `json` element with a `type` attribute with value `"object"` and as contents the first tweet. Then use the function `json:serialize`.
(c) Write an XQuery that produces a JSON array of JSON objects that contain the text, username, and date of all tweets. In other words, reproduce the whole JSON database but without all details except these three attributes.
(d) Write an XQuery that produces a JSON output with per user how many tweets (s)he sent sorted by the amount of tweets descending.

□

5.2.4 Assignment 4: RDF data and SPARQL querying

Goal Formulate queries in SPARQL based on natural language questions

Preparation

1. Apache Jena's SPARQL tutorial: [4] `http://jena.apache.org/tutorials/sparql.html`
2. Apache Jena's Introduction to RDF and the Jena RDF API: `http://jena.apache.org/tutorials/rdf_api.html`
3. Apache Jena's documentation on how to execute SPARQL queries from Java: `https://jena.apache.org/documentation/query/app_api.html`
4. Documentation for Apache Jena's triple store TDB: `http://jena.apache.org/documentation/tdb/index.html`

Installation Apache Jena

Go to the "Download" page of Apache Jena: `http://jena.apache.org/download/index.cgi` and download the latest distribution. Version 2.12.1 was used for preparing this practicum. We have done some testing on version 3.3.0. There may be slight differences if you use a different version.

You need the Java 7 SE for 2.12.1 and Java 8 SE for 3.3.0. Newer versions of Jena may require even newer versions of Java SE installed. Different versions of Java SE (previously known as Java JDK) can be downloaded from `https://www.oracle.com/technetwork/java/javase/downloads`. You can check which version of Java you have on your computer by opening a terminal/command window and issuing the command '`java -version`'.

Then, unpack the downloaded file in a suitable folder. As instructed in the README, set the environment variable `JENA_HOME` to the directory where you unpacked Jena. Execute the command `bat\sparql.bat --version` (Windows) or `bin/sparql --version` (Linux, MacOS-X) to test whether or not your installation works.

The data: royal families

We use data on royal families downloaded from `http://www.daml.org/2001/01/gedcom/`. It is available from Canvas (`royal92.zip`). The ontology (i.e., the schema) is based on the GEDCOM standard `http://en.wikipedia.org/wiki/GEDCOM`:

- Entities: attributes
  - Individual: givenName, surname, sex
  - Family
  - Event: date, place
    - IndividualEvent
      - Birth
      - Death

[4] The tutorial should be sufficient. If you want to know more details, look into the reference documentation of ARQ, the SPARQL Processor for Jena: `http://jena.apache.org/documentation/query/index.html`

t	p	vt
a:Birth	a:date	
a:Birth	a:place	
a:Birth	rdf:type	
a:Death	a:date	
a:Death	a:place	
a:Death	rdf:type	
a:Divorce	rdf:type	
a:Family	a:divorce	a:Divorce
a:Family	a:marriage	a:Marriage
a:Family	rdf:type	
a:Individual	a:birth	a:Birth
a:Individual	a:childIn	a:Family
a:Individual	a:death	a:Death
a:Individual	a:name	
a:Individual	a:sex	
a:Individual	a:spouseIn	a:Family
a:Individual	a:title	
a:Individual	rdf:type	
a:Marriage	a:date	
a:Marriage	a:place	
a:Marriage	rdf:type	

Figure 5.1: Output of the query royalschema.rq

        - FamilyEvent
            - Marriage
            - Divorce
- Relationships (between): attributes
    - childIn (Individual, Family)
    - spouseIn (Individual, Family)
    - birth (Individual, Birth)
    - death (Individual, Death)
    - marriage (Family, Marriage)
    - divorce (Family, Divorce)

Instructions You can also get an idea of the ontology from the data itself. Execute the query `royalschema.rq` by issuing the following command:

```
bin/sparql --data=royal92.nt --query=royalschema.rq
```

or (Windows)

```
bat\sparql.bat --data=royal92.nt --query=royalschema.rq
```

The query determines all possible relationships of an instance of a certain type with a value or an instance of another type. Just have a look inside the file `royalschema.rq` and try to understand the query. You should have seen the output of Figure 5.1 □

Exercises

Just upfront. In all queries it is convenient to use the following two prefix declarations.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX a:  <http://www.daml.org/2001/01/gedcom/gedcom#>
```

We follow the structure of the SPARQL tutorial of Jena. If you are looking for examples and explanations, please look at the corresponding sections there.

☞ **5.8** Basic patterns

(a) Give a query that show all titles that a woman has ever had. With `SELECT DISTINCT` you can, just as in SQL, filter out any duplicates in the resulting list. There are 42 unique titles.
(b) Give a query that shows all places (a:place) where once a royal marriage (a:Marriage) has taken place. It is quite a list (I haven't counted them) starting with "Paris,France" and "Amsterdam,Netherlands".

□

☞ **5.9** Value constraints

(a) Expand the query of the previous assignment in such a way that it only lists places in The Netherlands (i.e., that contain the word "Netherlands"). There are three places in the Netherlands: Amsterdam, Haus Doorn, The Hague.
(b) Give a query that shows the name of the husband of "Beatrix of_Netherlands //". The answer should be "Claus /von_Amsberg/".

□

☞ **5.10** Optionals / UNION

(a) Give a query that lists all persons (a:Individual) who have "Beatrix" or "Beatrice" in their name, with their title when available in the data. There a 9 persons who are called "Beatrix" or "Beatrice", but one of them is present with and without title, so you could get her twice.
(b) Give a query that lists all persons in the family of "Beatrix of_Netherlands //", i.e., her husband as well as her children. The list should contain four persons: "Claus von Amsberg", "Constantine", "John Friso", and "William Alexander".

□

☞ **5.11** Executing queries from Java Jena is in essence a Java framework. The `sparql` command simply executes a Java program. Besides the tutorials and documentation of Jena, it may also be instructive to look into folder `src-examples` where you can find example Java programs.

(a) Implement a small Java program that executes one of the above queries and writes the result to the screen.

□

☞ **5.12** Storing and querying RDF data in a Triple store. TDB is Jena's triple store, i.e., its database for storing and querying RDF data sets. The triple store can be accessed and manipulated from Java, but there are also handy command-line utilities in the `bin` folder.

(a) Use the command-line utilities to create a database in some location (just create a new folder for the triple store's location), load the `royal92` data set in the database and execute one of the above queries on this database.

□

☞ **5.13** With SPARQL it is possible to query remote *SPARQL endpoints*. These are servers, probably running a triple store as in the previous question, that allow users to query their data over the internet.

As an example, suppose you have a file `document.xml` and the document contains `owl:sameAs` predicates to link the objects in the file to 'the same' objects in DBpedia. Then you can query all DBpedia-properties for your objects with the following SPAQRL-query:

```
SELECT ?local ?remote ?remote_property ?remote_value
WHERE {
  <document.xml#id> owl:sameAs ?remote .
  SERVICE <http://dbpedia.org/sparql> {
    ?remote ?remote_property ?remote_value
  }
}
```

(a) Add such a `owl:sameAS` triple to `royal92.owl` to link 'our' Beatrix-object to DBpedia's (see `http://live.dbpedia.org/page/Beatrix_of_the_Netherlands`). Then, query for all DBpedia properties of Beatrix using the above template.