Databases

TDA357/DIT621– LP3 2023

Lecture 10

Ana Bove

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

February 13th 2023

Recall Last Lecture

- Databases in software application:
  - How to connect to a database from an application (Java/Python);
  - How to query and modify a database;
  - Problems with writing queries with strings;
  - Injection attacks;
  - Prepared statements (to prevent **SQL** injection);
  - Debugging;
  - DBMS vs. Java/Python: don't do what **SQL** do best in Java/Python!

Overview of Today's Lecture

- Semi-structured documents:
  - HTML;
  - XML and XPath;
  - **JSON**:
    - Syntax;
    - Paths;
    - **JSON** in PostgreSQL.

  (More on **JSON** next lecture!)

Semi-structured Documents (SSD)

So far we have seen relational databases.

But data does not always have to be in tables.

Data can be in graphs with nodes, in key-value files, in documents, ...

Today and next lecture we will look at *semi-structured documents*, that is, documents that have some kind of structure.

They are usually inefficient but more flexible (the structure is maybe not uniformed across the data) and portable.

Example: Some such (tree-structured) documents are: HTML, XML, **JSON**.

Note: The focus in the course has moved from XML to **JSON**.

Old exam questions about XML can be translated into the corresponding **JSON** questions.

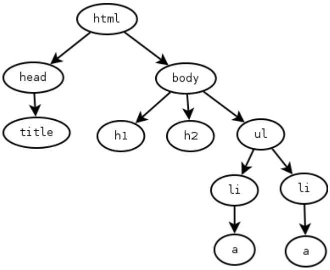
HTML: HyperText Markup Language

HTML is the language for web pages with ingredients:

- Text;
- Elements that indicate the structure of the page:
  - Delimited by start tags and end tags, e.g. <h1>. . . </h1>;
  - Elements can contain text and other elements;
  - Some elements have attributes, e.g. links: . . . .

```
<!DOCTYPE html>
<html>
  <head>
    <title>Universities in Gothenburg</title>
  </head>
<body>
  <h1>Universities</h1>
  <h2>Gothenburg</h2>
  <ul>
    <li><a href="http://www.chalmers.se/">Chalmers</a></li>
    <li><a href="http://www.gu.se/">GU</a></li>
  </ul>
</body>
</html>
```

Documents are Trees



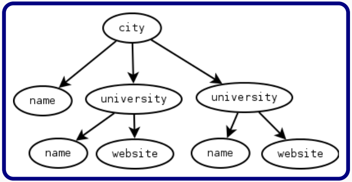

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Universities in Gothenburg</title>
  </head>
<body>
  <h1>Universities</h1>
  <h2>Gothenburg</h2>
  <ul>
    <li><a href="http://www.chalmers.se/">
        Chalmers</a></li>
    <li><a href="http://www.gu.se/">
        GU</a></li>
  </ul>
</body>
</html>
```

XML: eXtensible Markup Language

XML appeared in the '90s, after the rise of HTML.

Like HTML but for arbitrary tree-structured data.

Different document types use different sets of elements.




```
<city>
  <name>Gothenburg</name>
  <university>
    <name>Chalmers</name>
    <website>www.chalmers.se</website>
  </university>
  <university>
    <name>GU</name>
    <website>www.gu.se</website>
  </university>
</city>
```

Attributes vs Elements

- Text in XML can appear inside elements or in attributes.

```
<university>
  <name>Chalmers</name>
  <website>www.chalmers.se</website>
</university>
```

```
<university name = “Chalmers” >
  <website>www.chalmers.se</website>
</university>
```

```
<university name = “Chalmers” website = “www.chalmers.se” ></university>
```

- There is a short-hand for empty elements:

```
<university name = “Chalmers” website = “www.chalmers.se” />
```

Attributes vs Elements (Cont.)

Which one is best? Elements are easy to extend, attributes are limited.

From w3schools.com/xml/xml_dtd_el_vs_attr.asp: *"My experience is that attributes are handy in HTML but in XML you should try to avoid them"*

```
<university>
  <name>Chalmers</name>
  <channel>
    <website>www.chalmers.se</website>
    <instagram>...</instagram>
  </channel>
  <department>
    <name>Computer Science and Engineering</name>
    <code>CSE</code>
    <size>280</size>
  </department>
  <department>
    <name>Mathematics</name>
    <code>MV</code>
  </department>
</university>
```

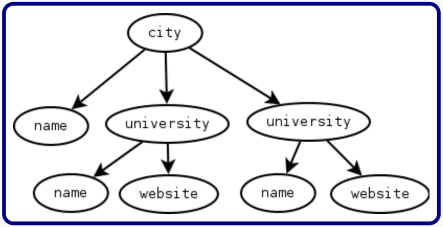
Well-formed XML Documents

- An XML document can contain (or refer to) a *DTD (Document Type Definition)* which defines:
  - The set of elements that are allowed;
  - How elements can be nested;
  - Which attributes elements have.
- An XML document is valid if it complies with the constraints expressed in its DTD;
- An XML document has a single root element;
- All elements have a start tag and an end tag (though recall short-hand for empty elements), and tags must be properly nested;
- HTML documents can be seen as XML documents of specific types:
  - XHTML (XML-compliant HTML) follows the rules of XML;
  - Plain HTML syntax is more relaxed, e.g. it allows some end tags to be omitted.

XPath Queries

Since documents are trees, we can access parts of a document by specifying the path from the root to the desired subtrees/parts.

The syntax is similar to Unix directory paths.



Example: Some XPath queries:

- /city/*[1]: first child of <city>
- /city/university: all <university> elements right under <city>
- /city//website: all <website> elements anywhere under <city>
- /city//@attrib: all values of "attrib" attributes anywhere under <city>

JSON: JavaScript Object Notation

JSON appeared in the 2000's, after the rise of JavaScript, and it is growing quickly into the standard data format of the web.

JSON documents have a very simple syntax:

object	::=	{ "key" : value , ... , "key" : value }
value	::=	null | true | false | "string" | number | array | object
array	::=	[value, ... , value]

Moreover:

- **JSON** data can be freely distributed into lines;
- The order of key-value pairs in an object does not matter;
- The order of items in an array does matter;
- The values in an array can have different structures/be heterogenous.

Example: **JSON** Values, Objects and Arrays

Some **JSON** numerical values: 10, -3.1416, 6.4e5.

A **JSON** object: { "city": "Gothenburg", "population": 550000 }.

A **JSON** array: [3, "a string here", [1, "text"], true, { "obj": null }, [ ]].

A bigger **JSON** object:

```
[ { "city": { "name": "Gothenburg",
              "universities":
                 [{ "name": "Chalmers",
                    "website": "www.chalmers.se"},
                  { "name": "GU",
                    "website": "www.gu.se"}
                 ]
            }
  },
  { "city": { "name": "Montevideo",
              "universities": "UdelaR",
              "population": 1750000
            }
  } ]
```

Simple Paths in **JSON**

Let d be the big document/object at the bottom of the previous slide.
We can use Java-like object syntax to retrieve the values in the document.

d[0]: The very first element in the array
{ "city": { "name": "Gothenburg", "universities": [...] } }.

d[1].city: The value of the key "city" in the second element in the array
{"name": "Montevideo", "universities": "UdelaR",
"population": 1750000 }.

d[0].city.name: "Gothenburg".

d[0].city.universities[1].name: "GU".

d[1].city.population: 1750000.

d[1].city.name.population: d[1].city.name is not an object so it has no "population" key!

XML vs JSON

```
<teacher>
  <name>Ana</name>
  <surname>Bove</surname>
  <course>Databases</course>
</teacher>
```

```
{ “teacher”:
     {“name”: “Ana”,
      “surname”: “Bove”,
      “course: “Databases”
     }
}
```

Tables vs **JSON**

name	**surname**	**course**
Ana	Bove	Databases
Nils Anders	Danielsson	Automata
Jonas	Duregård	Databases

[{“name”: “Ana”, “surname”: “Bove”, “course: “Databases”},
{“name”: “Nils Anders”, “surname”: “Danielsson”, “course: “Automata”},
{“name”: “Jonas, “surname”: “Duregård”, “course: “Databases”}
]

JSON in PostgreSQL

PostgreSQL has extensive support for **JSON**.

- Types:

  JSON: **JSON** values stored as text;

  JSONB: **JSON** values converted to an internal binary format; more efficient for queries;

- Functions for creating **JSON** values;
- Operators to extract values from **JSON** documents.

Example: Database for a Social Media Application

```
CREATE TABLE Users (
  uname TEXT PRIMARY KEY,
  email TEXT UNIQUE );

CREATE TABLE Posts (
  id SERIAL PRIMARY KEY,
  author TEXT NOT NULL REFERENCES Users,
  created TIMESTAMP NOT NULL,
  content JSONB NOT NULL );
```

The attribute of type **JSON** can contain data (given as string) in a more flexible format that can be extended without changing the DB design.

```
INSERT INTO Posts VALUES ( DEFAULT, 'AnaBove', CURRENT_TIMESTAMP,
  '{"link": "https://xkcd.com/327/", "preview": true}' :: JSONB );

INSERT INTO Posts VALUES ( DEFAULT, 'AnaBove', CURRENT_TIMESTAMP,
  '{"picture": "funnycat.gif", "prop": {"size":15434}}' :: JSONB );
```

Queries Involving **JSON** Documents

- Returns **JSON** strings, or **SQL** NULL values if no link key in content:

```
SELECT id, content->'link' AS url FROM Posts;
```

- Finds all pots with pictures:

```
SELECT id, content FROM Posts WHERE content->'picture' IS NOT NULL;
```

  Alternative:

```
SELECT id, content FROM Posts WHERE content ? 'picture';
```

- Finds all posts with enabled preview:
  Compares **JSON** values:

```
SELECT id, content FROM Posts WHERE (content->'preview') = 'true';
```

  Converts the **JSON** value to **SQL** Booleans:

```
SELECT id, content FROM Posts WHERE (content->'preview') :: BOOLEAN;
```

Queries Involving **JSON** Documents (Cont.)

- Gives all post sizes from 'prop', returns **JSON** type, gives NULL otherwise:

```
SELECT id, content, content->'prop'->'size' AS postsize FROM Posts;
```

- Can be combined with other **SQL** stuff to avoid NULL values; also, converts size to **SQL** number:

```
SELECT id, content,
        COALESCE(content->'prop'->'size','0') :: NUMERIC AS postsize
FROM Posts;
```

```
SELECT id, content,
        COALESCE(content->'prop'->'size','0') AS postsize
FROM Posts;
```

Building JSON Documents in Query Results

- Builds a **JSON** object from **SQL** values:

```sql
SELECT created, jsonb_build_object('postid', id, 'user', author) AS jsondata
FROM Posts;
```

- Builds **JSON** arrays instead:

```sql
SELECT created, jsonb_build_array (id, author) AS jsondata FROM Posts;
```

- Builds a **JSON** array with the users of all posts (might contain repetition!); returns only one row:

```sql
SELECT jsonb_agg (author) AS jsonarray FROM Posts;
```

- Collects some post information for each user into **JSON** arrays:

```sql
SELECT author,
        jsonb_agg (jsonb_build_object ('postid', id, 'time', created)) AS jsondata
FROM Posts
GROUP BY author;
```

Building **JSON** Documents in Query Results (Cont.)

- Gives an array per user with all his/her posts and their id; uses || to join two **JSON** objects:

```
SELECT author, jsonb_agg (jsonb_build_object ('postid', id) || content)
FROM Posts
GROUP BY author;
```

- All posts and their information, as one **JSON** array:

```
SELECT
  json_agg (jsonb_build_object ('id', id, 'author', author, 'created', created)
            || content)
FROM Posts;
```

A more Complex Example

Creates a **JSON** object for each user, containing their basic information and an array with the id and time for their posts.

COALESCE is used to replace **SQL** NULL values for users with no posts with **JSON** empty arrays.

```
SELECT jsonb_build_object ('uid', uname, 'email', email,
          'posts', (SELECT COALESCE (jsonb_agg (jsonb_build_object (
                                              'postid', id, 'time', created)),
                                         '[ ]')
                    FROM Posts WHERE author = Users.uname)
        ) AS user_data
FROM Users;
```

Note: Study this query carefully since it can come handy for the lab! :-)

A more Complex Example (Cont.)

A row in the previous query:

```
{"uid": "Nobody",
 "email": "nobody@example.com",
 "posts": [ {"time": "2023-02-09T19:11:18.026754",
             "postid": 5} ]
}
```

JSON Path Queries (more on paths next lecture!)

- Recall: gives **JSON** links in content; returns NULL if not such key in content

```
SELECT id, content->'link' AS url FROM Posts;
```

- Using a **JSON** path query for links: returns only the entries with such key in content

```
SELECT id, jsonb_path_query (content,'$.link') FROM Posts;
```

- This query fails when not all posts contain a link!

```
SELECT id, jsonb_path_query (content,'strict $.link') FROM Posts;
```

Querying JSON Objects with JSON Path (more next lecture!)

- Gives the values of all keys:

```
SELECT jsonb_path_query ('{"name": "foo", "size": 500, "type": "jpg"}',
'strict $.*');
```

- Gives just the first value:

```
SELECT jsonb_path_query_first ('{"name": "foo", "size": 500, "type":"jpg"}',
'strict $.*');
```

- Returns an array with the values of all keys:

```
SELECT jsonb_path_query_array ('{"name": "foo", "size": 500, "type": "jpg"}',
'strict $.*');
```

Overview of Next Lecture

- More on **JSON**:
  - **JSON** schema;
  - **JSON** path.

Reading:

Notes: chapters 9.1–9.3