

How to get started with Task 3

- Create a triggers.sql and add \i triggers.sql to end of runtests.sql
- Add three statements to triggers.sql :

CREATE FUNCTION ...

BEGIN

RETURN new;

END; ...

Your trigger code goes here!

Movet to tests.sql before submitting

CREATE TRIGGER ... INSERT ON Registrations ... ;

INSERT INTO Registrations('1111111111', 'CCC222', null); -- For testing

- Run it, should work and do nothing
- Make the trigger always raise an exception
- Make it only throw an exception if the user already passed the course
- Keep modifying until it does what it should ...

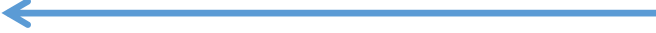
TDA357/DIT621 – Databases

Lecture 11 – Efficiency, Semi-structured Data Model, NoSQL and XML

Jonas Duregård

NoSQL databases

Stepping outside the box

- Data does not have to be in tables. How else can we do it?
- Graph databases
 - Our data is a graph with nodes
- Key/value stores
 - Store all data in a big map, lookup keys and get values
 - Simple, efficient, but kind of limited
- Document databases 
 - Store *documents*, that in turn contain structures
 - Inefficient, weak integrity, but lightweight and portable

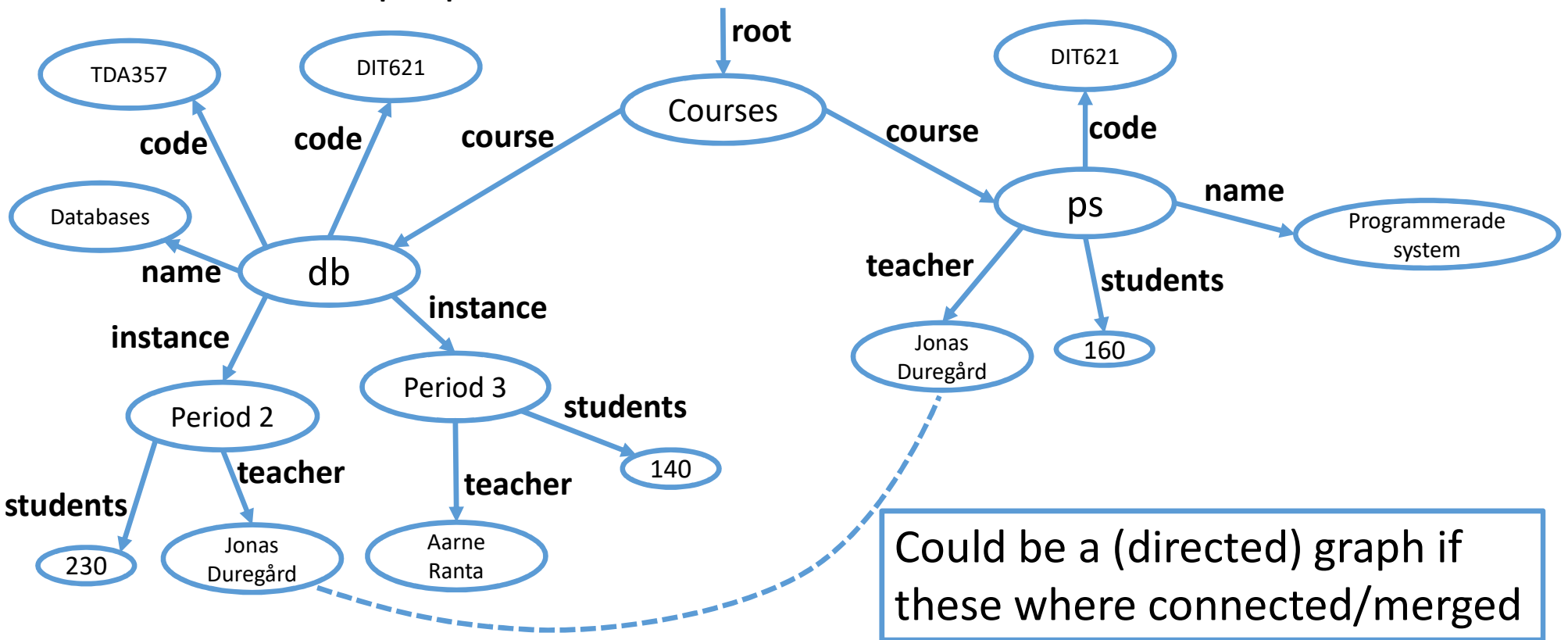
We will focus on this

Semi-structured data (SSD)

- The relational model has a very rich structure
 - Allows us to have strong constraints on data
- This structure also limits flexibility
 - Much of the design work is centered on deliberately preventing users from being flexible (by enforcing constraints)
- In semi-structured data models, the schema is flexible
 - Data is still structured
 - ... but the structure is not necessarily uniform across the data
 - E.g. data does not fit in tables where every row has the same columns

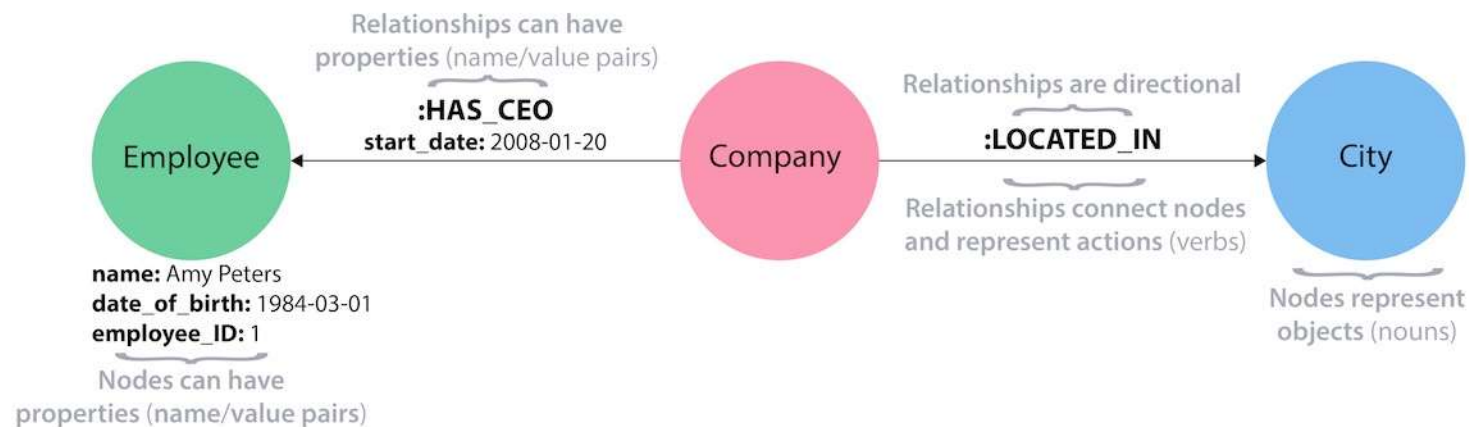
A different way of structuring data

- Here as a tree of objects, with labels on edges and data in nodes
- Note how the "properties" of courses differ



Side note: Graph databases

- All our data is stored in nodes and edges
- Somewhat similar to the entity-relationship model
- Example (from the neo4j graph database):

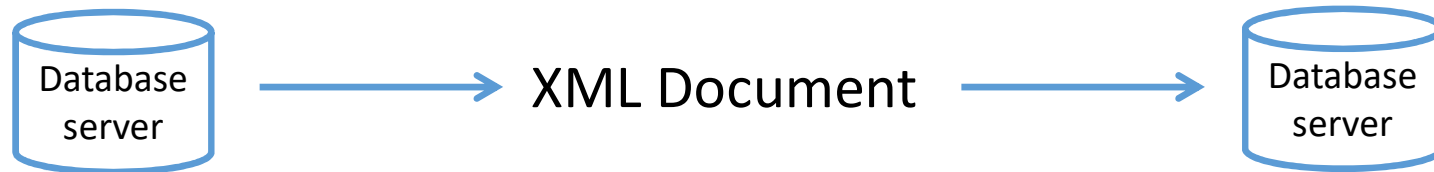


Examples of document-based SSD standards

- XML
 - Extensible Markup Language
 - Created in the 1990's
 - Syntax: `<tag attribute="value"><other_tag/>also some text</tag>`
- JSON
 - JavaScript Object Notation
 - Created in the 2000's
 - Collections of key/value pairs, very simple syntax
 - Used to various extents in lots of modern DBMS
- Both these are *document based*, a data set is most naturally described by a text document rather than a table
- Both are hierarchical, the documents have a tree-structure

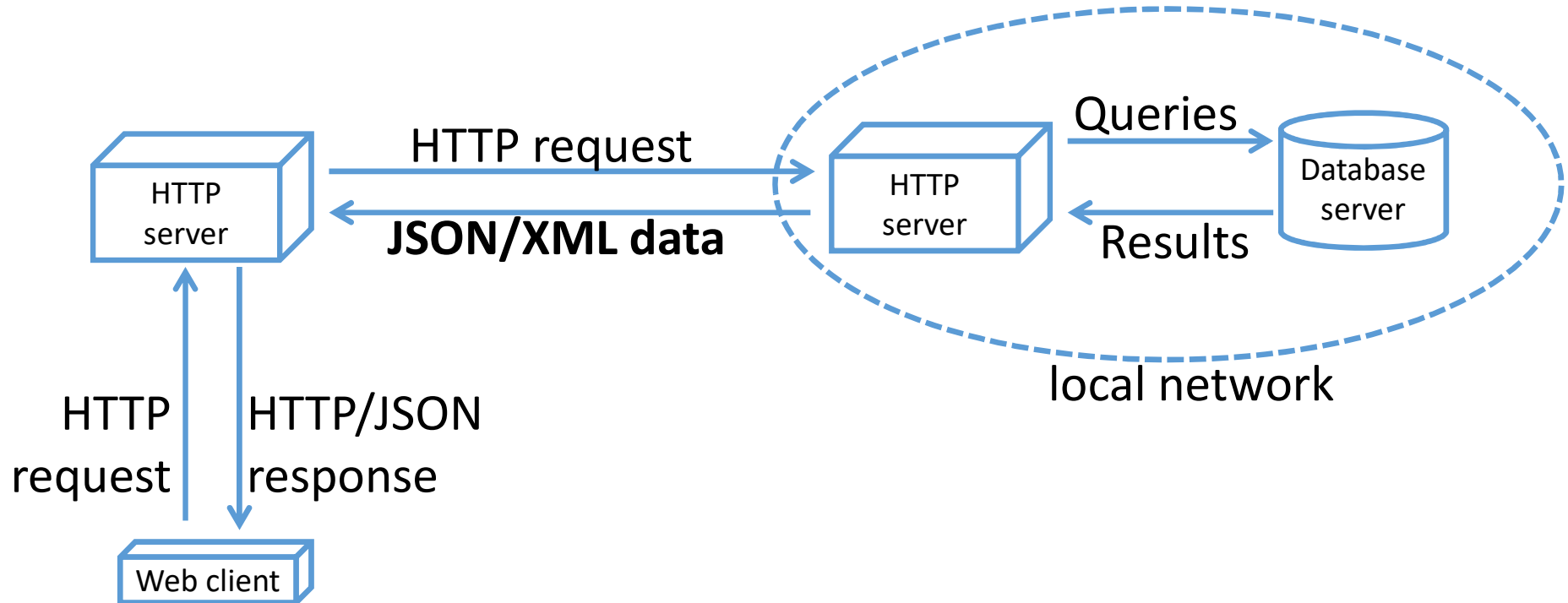
Data interchange formats

- Data interchange formats facilitate the transfer of data from one database to another
- Transforms data from one schema to another, via an intermediate format
- The interchange format must be flexible enough to conveniently represent data from both schemas



Cross domain communication

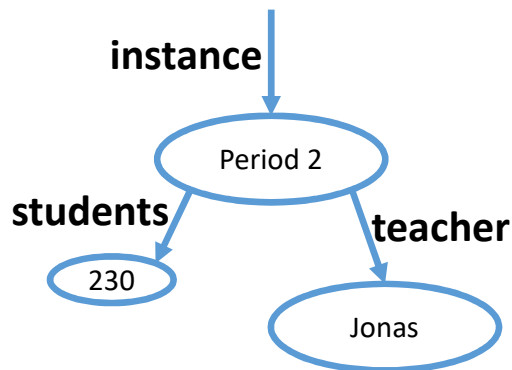
- When modern web services build web pages, it is not uncommon that they request information from other web servers
- Direct access to database servers over the Internet is not advisable



XML

- Derived from pre-existing document markup languages
 - Compare with HTML: HTML uses tags to format a web-page, XML uses tags to describe data
- Documents are built from elements, attributes and text

One way to express this tree in XML:



```
<instance name="Period 2">  
  <teacher>Jonas</teacher>  
  <students>230</students>  
</instance>
```

Closing tags

Contains 3 elements (instance, teacher, and students), 1 attribute (name), and 2 element texts (Jonas, 230)

Hierarchical structure of xml

- XML documents always have a single root element, that in turn may contain other elements with attributes/elements of their own etc.
- All tags must be closed
 - Allowed: `<grades><grade>G</grade><grade>VG</grade></grades>`
 - Not allowed: `<grades><grade>G<grade>VG</grades>`
 - Special case, self closing tag: `<tag/>` or `<tag att="val"/>`
- Tags must be properly nested
 - Allowed: `<a><c>text</c>`
 - Not allowed: `<a><c>text</c>`



Uses `` to close `<c>`

Mixing texts and elements

- It is valid to have text and subelements in the same element:

```
<tag>text<subtag></subtag></tag>
```

- This is considered bad practice, especially when you have things like

```
<tag>text<subtag></subtag>more text</tag>
```

- What is the semantic difference between the text before/after the subtag?
- In the hierarchical structure the two texts are on the same level

Attributes vs elements

- Two ways of representing a person in XML:

```
<Teacher>  
  <Firstname>Jonas</Firstname>  
  <Lastname>Duregård</Lastname>  
  <Course>Databases</Course>  
</Teacher>
```

Element-centric



Attribute-centric



```
<Teacher firstname="Jonas" lastname="Duregård" course="Databases"/>
```

- Attributes and elements can be mixed however we want. What should we use?
 - Having firstname as an attribute and lastname as an element seems odd
 - Maybe firstname/lastname should be attributes, and courses elements (the names feel "attributy" whereas a course feels more "entityish")

Not technical terms...



Attributes vs elements

- Suppose we want Jonas to have two courses, and each course to have both a name and a code?
- Elements are easy to extend, attributes are very limited

```
<Teacher>
  <Firstname>Jonas</Firstname>
  <Lastname>Duregård</Lastname>
  <Course>Databases</Course>
</Teacher>
```

Extra name element to avoid
mixing text and elements

```
<Teacher>
  <Firstname>Jonas</Firstname>
  <Lastname>Duregård</Lastname>
  <Course>
    <Name>Databases</Name>
    <Code>TDA357</Code>
  </Course>
  <Course>
    <Name>Programmerade system</Name>
    <Code>TDA143</Code>
  </Course>
</Teacher>
```

Summary: Attributes vs elements

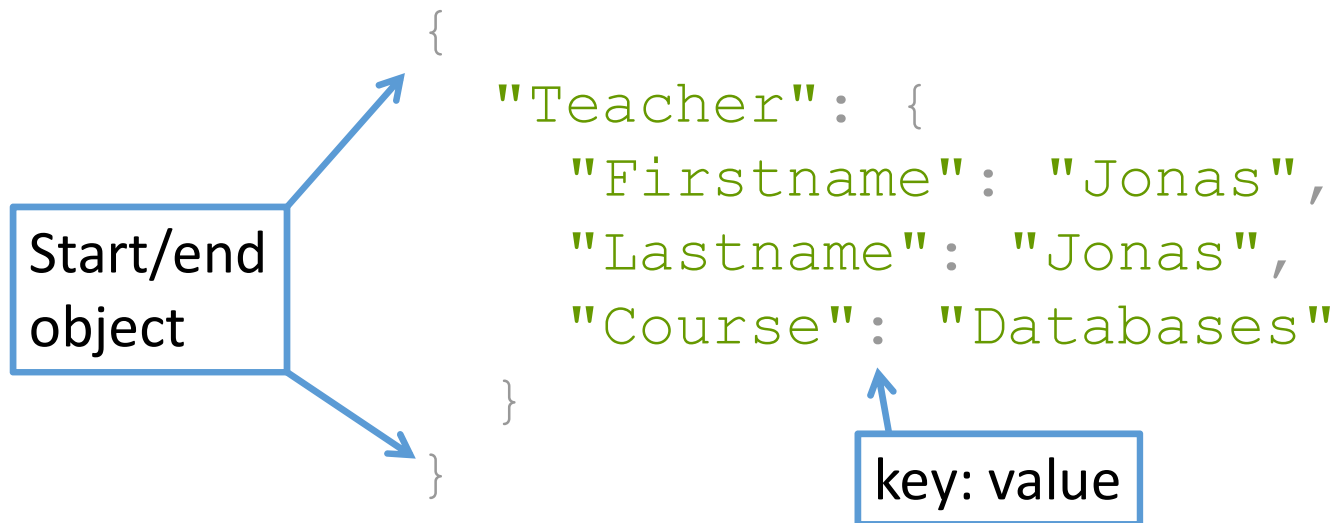
- Advantages of attributes:
 - Compact syntax
 - Correspond naturally to attributes in relational databases
- Advantages of elements:
 - Can represent complex objects (with attributes, subelements etc.)
 - Can have arbitrarily many elements with the same tag
 - Easily extensible (remember: we are using XML for flexibility!)
- Compare with ER-modelling: Anything that needs to have attributes of its own can never be an attribute
- Often, elements are used to represent the actual data, while attributes are used to describe "modifiers" of tags

Is an XML document a database?

- Yes, in a wide sense
- It contains data in a structured, (sort of) persistent manner
- It is very unlike a relational database:
 - There is no "XML-server" corresponding to PostgreSQL
 - There is no insert operation that adds data into a document
 - Documents are either generated by a program or written by hand
 - We typically do not write queries on our documents (but we can)
 - Documents are processed by programs, using library functions etc.
 - We do not have constraints on documents (but they can be *validated*)

JSON

- Think of JSON documents as Java(Script) objects without any methods
 - Objects that can have variables (that are objects or primitive types)
 - The "variable names" are called keys in JSON
- This document contains an object that has a single variable/key, "Teacher"
- The value of "Teacher" is an object containing three variables of type String



This (and all JSON)
is actual JavaScript
code!

XML or JSON

- Here is a tiny XML document, and a tiny JSON document
 - Notice how they are doing pretty much the same thing?

```
<Teacher>
  <Firstname>Jonas</Firstname>
  <Lastname>Duregård</Lastname>
  <Course>Databases</Course>
</Teacher>
```

Four elements and three strings

```
{
  "Teacher": {
    "Firstname": "Jonas",
    "Lastname": "Jonas",
    "Course": "Databases"
  }
}
```

Two objects, four keys, and three strings

XML or JSON

- Both XML and JSON can be used as semi-structured data formats
 - E.g. to receive data from a web server, for data exchange etc.
- Both are used in practice and there are good arguments for using either
- Traditionally this course teaches only XML
- Since last year we are switching focus towards JSON
 - It has simpler syntax
 - It is growing quickly into the standard data format of the web
 - Also this year JSON is used in the Assignment (Task 4)

So what will I need to know for the exam?

- *Read* and understand XML documents (already done)
- *Read, write, validate* and *query* JSON documents
(rest of this weeks lectures, and the exercise on Friday)
- Most old exam questions about XML can be translated into corresponding JSON questions (replacing DTD with JSONSchema and XPath with JSONPath)

Full syntax of JSON

Every JSON value is built from a combination of six types:

- Structures:
 - Objects: `{ "key1" : JSON, "key2" : JSON, ... }`
Can have 0 or more key:value pairs, values can be any JSON value
 - Arrays/lists: `[JSON, JSON, ...]`
Can have 0 or more items, each can be any JSON value
- Literals (usually the "leafs" in the document tree)
 - Java-esque strings: `"Hello world!\n"`
 - Numbers: `7`, `5.3`
 - Booleans: `true` or `false`
 - `null`

An example document

```
[{"city":"Boston", "population":700000},  
 {"city": "New York",  
  "boroughs": [  
    "The Bronx",  
    "Brooklyn",  
    "Manhattan",  
    "Queens",  
    "Staten Island"  
  ]  
}]
```

- An array containing two objects
- First object has two keys:
city (string) and population (number)
- Second object has two keys:
city (string) and boroughs (array)
- The array in boroughs contains five strings

Simple paths in JSON

- We can use Java-like object syntax to address sub-values in documents
- If we call this document d, what is the value of d[1].boroughs[2]?
 - Answer: "Manhattan"
- What about d[0]?
 - Answer: {"city":"Boston", "population":700000} (an object)
- What about d[0].city.population?
 - Answer: d[0].city is not an object, so applying .population is an error (or maybe null depending on your philosophical beliefs)

```
[{"city":"Boston", "population":700000}, {"city": "New York",  
  "boroughs": [  
    "The Bronx",  
    "Brooklyn",  
    "Manhattan",  
    "Queens",  
    "Staten Island"  
  ]  
}]
```


JSON and types

- Arrays in JSON are heterogenous, this is allowed:

`[1, "string", {"key":0}, true, [1,2]]`

- It's an array containing a number, a string, an object, a boolean and an array
 - In practice we may want to limit ourselves to homogenous arrays, where all values have the same type

JSON Documents do not need to be objects

- Often, the top level of a document is an object
- Sometimes it's an array
- It can in principle be just a number, a string, or a boolean
 - Examples, all of these are valid JSON documents:
 - 42
 - true
 - null
 - "hello world"

Which are well formed JSON documents?

`[[[1]]]`

- Yes, this is an array containing an array containing an array containing 1

`{"city" : "Gothenburg"}`

- No! The outer object is not on the form key:value

`"city" : "population" : 3000`

- No! This is on the form x:y:z which is never allowed in objects

`"city":{"boston"}`

- No! The object `{"boston"}` is not valid (contents should be key:value)

`[{}, [], ""]`

- Yes, it's an array containing the empty object, array and string

Building JSON documents using Postgres

Try it out

- All the code from this slide on is available on the course homepage in `jsonpostgres.sql`
- Run it, modify it to do something different, use part of it as a base for the information part of Task 4 ...

<https://www.postgresql.org/docs/current/functions-json.html>
<https://www.postgresql.org/docs/current/datatype-json.html>

JSON support in Postgres

Postgres (especially version 12) has extensive support for JSON

- Covered this lecture:
 - Two SQL types for: JSON (text format) and JSONB (faster binary format)
 - You can have JSON documents in table cells, views, query results, ...
 - Functions for building JSON documents
 - Operators for extracting values from JSON documents
(e.g. turn a JSON string to an SQL string to use it in a WHERE-clause)
- Covered next lecture:
 - Writing JSON Path queries to extract more advanced data from JSON

A very flexible table

- This could be a table in some social media application
- It has three regular old columns with regular old constraints, and a fourth column containing JSONB
- The idea is that the JSON can contain data in a more flexible format that can be extended without changing the DB design

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



The only new stuff on this slide!

Inserting data

Posts(id, author, created, content)

- One way of encoding a link post and a picture post

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

Sometimes, we need to use ::TYPE
to convert values to different SQL types

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

JSON values written as string :: JSONB

Posts(id, author, created, content)

Basic querying with JSON

SELECT * FROM Posts;

id	author	created	content
1	Jonas	2019-11-28 15:01:31.247316	{"link": "https://xkcd.com/327/", "preview": true}
2	Jonas	2019-11-28 15:01:31.252545	{"prop": {"size": 15434}, "picture": "funnycat.gif"}

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

id	url
1	"https://xkcd.com/327/"
2	

Gives content.link as a JSONB value

This is a JSON string (not an SQL string)

This is a (SQL) null value (not a JSON null value!)

```
Posts(id, author, created, content)
```

Using JSON in WHERE-clauses

- Find all picture posts:

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

id	content
2	{"prop": {"size": 15434}, "picture": "funnycat.gif"}

As a shorthand, Postgres allows: **WHERE** content ? 'picture';

Even more JSON querying

- Two ways of finding all posts with enabled previews

Compare to the JSON true value



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

Convert the JSON boolean to an SQL boolean



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

Always remember the difference between e.g. a JSON number and an SQL number

Nested access

- Select the size property of all posts

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

id	content	postsize
1	{"link": "https://xkcd.com/327/", "preview": true}	
2	{"prop": {"size": 15434}, "picture": "funnycat.gif"}	15434

A tiny JSON document (type JSONB), not an SQL number

Use what you already know

- All the new JSON features can be combined with the SQL feature we already know to express even more!
- Example: Select all post sizes, replace nulls with 0's and convert to SQL numbers

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

id	postsize
1	0
2	15434

SQL numbers

We can create a view from this, aggregate with SUM to compute total size for each user etc.

Building JSON in query results

- Even if you have no tables with JSON values, you can still use JSON in postgres
 - We can run queries that construct JSON documents from table data
 - This is what you will be doing in the lab (Task 4) so pay attention 😊

Building objects: jsonb_build_object

- A built in stored procedures (functions) for creating JSON objects:

jsonb_build_object(key1,value1,key2,value2,...)

- Example (three rows in Posts):

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

Selects id, and also a little JSON object for each row in Post

id	jsondata
1	{"user": "Jonas", "postid": 1}
2	{"user": "Jonas", "postid": 2}
3	{"user": "sanoJ", "postid": 3}

Table: Posts

<u>id</u>	author	...
1	Jonas	...
2	Jonas	...
3	sanoJ	...

Strings, numbers etc. are automatically converted from SQL to JSON

Aggregating into JSON arrays: jsonb_agg

- jsonb_agg is an aggregation function (like SUM, COUNT, ...)
 - Must either be the only thing selected, or have a GROUP BY
- Simple example, build an array with all post authors (3 rows in Posts):

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

jsonarray

["Jonas", "Jonas", "sanoJ"]
(1 row)

Groups together all rows
and takes the author value
from each row in the group

Always gives a single row (without GROUP BY)

Example using group by

- Task: For each user in Posts, create a JSON array containing JSON objects for every post the user has
- Desired output for our table:

Table: Posts

<u>id</u>	author	...
1	Jonas	...
2	Jonas	...
3	sanoJ	...

author	jsondata
Jonas	[{"user": "Jonas", "postid": 1}, {"user": "Jonas", "postid": 2}]
sanoJ	[{"user": "sanoJ", "postid": 3}]

Plan:

- Group rows by their author
- for each group use jsonb_agg to create a JSON array item for every post
- use jsonb_build_object to create objects in the array

Example continued

- Group rows by their author
- for each group use jsonb_agg to create a JSON array item for every post
- use jsonb_build_object to create objects in the array

Table: Posts

<u>id</u>	author	...
1	Jonas	...
2	Jonas	...
3	sanoJ	...

```
SELECT author, jsonb_agg(jsonb_build_object(  
    'postid',id,  
    'user',author)) AS jsondata  
FROM Posts  
GROUP BY Author;
```

author	jsondata
sanoJ	[{"user":"sanoJ","postid":3}]
Jonas	[{"user":"Jonas","postid":1}, {"user":"Jonas","postid": 2}]

Going nuts with correlated queries

- This beautiful query creates a JSON object for each user, containing their basic information and an array of posts

SELECT

```
json_build_object(  
  'uid', uid,  
  'email', email,  
  'posts', (SELECT coalesce(jsonb_agg(jsonb_build_object(  
    'postid', id,  
    'time', created)  
  ), '[]') FROM Posts WHERE author = U.uid)  
)
```

A single item in the SELECT, building a massive JSON object for each row in Users

Refers to outer query

FROM Users U;

jsonb_agg gives (SQL) null for empty sets, so coalesce to []

- One row from the result of the query on the last slide:

```
{ "uid" : "Jonas",  
  "email" : "jonas.duregard@chalmers.se",  
  "posts" :  
    [ { "time" : "2019-12-02T14:52:37.451796",  
        "postid" : 1 }  
      , { "time" : "2019-12-02T14:52:37.453225",  
        "postid" : 2 }  
    ]  
}
```

A number in an object in an array in an object 😊
This is at position x.posts[1].postid
Tells us that one of the posts for user Jonas has ID 2

Regarding the assignment

- In the assignment you are supposed to create a JSON object for a given student containing lots of information, including lists of passed courses and such
- Hey, that sounds a lot like the thing on the last slide!
 - It is possible to solve the whole information part of Task 4 using one glorious query
 - Experiment in a .sql file until you get it working, then move it into Java
- You don't have to use the JSON features of Postgres at all in the lab, but I highly recommend it

Tomorrow

- JSON Schema
 - Final piece of the puzzle for the assignment!
- JSON Path
- Tomorrows lecture will be given by Matthías Páll Gissurarson (whom most of you will have met in the lab / exercise sessions)