TDA357/DIT621 – Databases

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

Jonas Duregård

Efficiency of databases

In this course we do not talk a lot about efficiency, primarily for two reasons:

1. Predicting database performance is difficult due to automatic optimizations
   - Writing a more complicated query for efficiency may make no difference
   - Worse: It may degrade performance because the DBMS fails to optimize it
2. Premature optimization is a problem
   - A lot of people worry about performance when they should be worrying about correctness and ease of use (and productivity)

- A good approach to efficiency in most cases: Write a simple and elegant solution. If it is too slow, write a messy but hopefully efficient solution
    - Use the simple solution as a reference to test the messy one

A famous quote (my emphasis)

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%."

- Donald Knuth (1974)

About complexity of database operations

- Analyzing database operations using asymptotic complexity (O-notation) can be misleading
- The reason is that most work of a typical persistant database is spent on:
  - Disk access (reading/writing data from disk)
  - Network communication (receiving queries and sending results)
- The former means that rather than the number of rows, we should discuss how many different disk locations we fetch rows from
- The latter means writing fewer queries is better

General techniques for efficient databases

- Today I will show you two techniques that can be used to make databases more efficient:
  - Indexes
  - Materialized views
- Neither of them are "silver bullets"
  - All obvious optimizations are done automatically by the DBMS
  - Both techniques can inadvertantly degrade performance
  - Require careful consideration

Indexes

- I have a large printed collection of karaoke tracks, organized by producing label
- Users typically want to look up tracks by artist or title
- Problem: Searching through the whole collection linearly is time consuming
- Solution: Print two indexes, one where all tracks are ordered by artist and one where they are ordered by title
  - Users can binary search (most of them without knowing it)
  - Whenever we add new tracks to the collection, we also need to update both the indexes

Indexes in SQL

- An index is separate from, but connected to a table
- It is created on a set of columns on that table
- It allows us to quickly find all rows with given values for those columns
  - An index on (artist,album) would allow us to quickly find all karaoke tracks with a certain artist and album name
- How lookups work is DBMS-specific, but hash-tables are typically used
- Created automatically for primary keys and UNIQUE-constraints
- Automatically used when selecting/updating/deleting using WHERE-clauses
- Updated automatically when a row is deleted/updated/inserted
  - May make inserts a lot slower
- Uses disk space, often more than the table itself

Why not create all indexes automatically?

- For a table with 10 columns, there are 2^{10}-1=1023 possible indexes
- Updating a thousand indexes with each insert/update/delete would be extremely detrimental to performance
- Disk space usage could suddenly become an issue if you use about 1000 times more space
  - Megabytes become gigabytes, gigabytes become terrabytes...

When should indexes be used?

- These conditions give a hint that using indexes may be a good idea:
  - You have many rows (if you have a few hundred rows, the extra disk access for reading the index is more time consuming than a linear lookup)
    - Why are you even worrying about performance for hundreds of rows?!
  - You are frequently doing lookups/joins etc. on a non-key
  - You are not worried about inserts being slower or disk space issues

Creating indexes

- Most DBMS support the statement:
  CREATE INDEX index_name ON table(attributes);
  - Not part of the SQL standard
- Can be done on existing tables with data (may take some time to create)
- Existing SQL queries do not need to be changed in any way ☺

Materialized views

- The views you have been writing are *virtual*
  - They are just a name for a query
  - Using a view in a query FROM-clause is the same as using a subquery
- Obvious opportunity for performance gain:
  Caching the result of the view could save a lot of time
- Obvious new performance problem:
  When the table data is changed, the query result needs to be updated

Materialized views in SQL

- Replace CREATE VIEW by CREATE MATERIALIZED VIEW
- Instead of just writing a query, you create a special kind of table that is automatically updated to reflect the result of the query
  - May have to recompute the whole query when a table is updated
  - If updates are more frequent than selections, the materialized view will be less efficient than a virtual view

Materialized views in postgres

In postgres, things work a bit differently:

- Materialized views are NOT automatically updated
- They reflect only the data that is in the tables when it is created
- User needs to run REFRESH MATERIALIZED VIEW; to update it
  - Can be done via a trigger on underlying tables to emulate the standard behavior (not a FOR EACH ROW trigger though!)

When should materialized views be used?

- You have a very costly query in a view
- One of two situations:
  - The underlying tables are rarely modified and view is often selected from
  - You are OK with the view showing slightly outdated data (postgres specific)
- The latter could be implemented by scheduling a REFRESH to be done e.g. once per hour/day/week
  - Example: We have a view that shows the number of members in all Facebook groups, but it doesn't have to be up to this minute

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

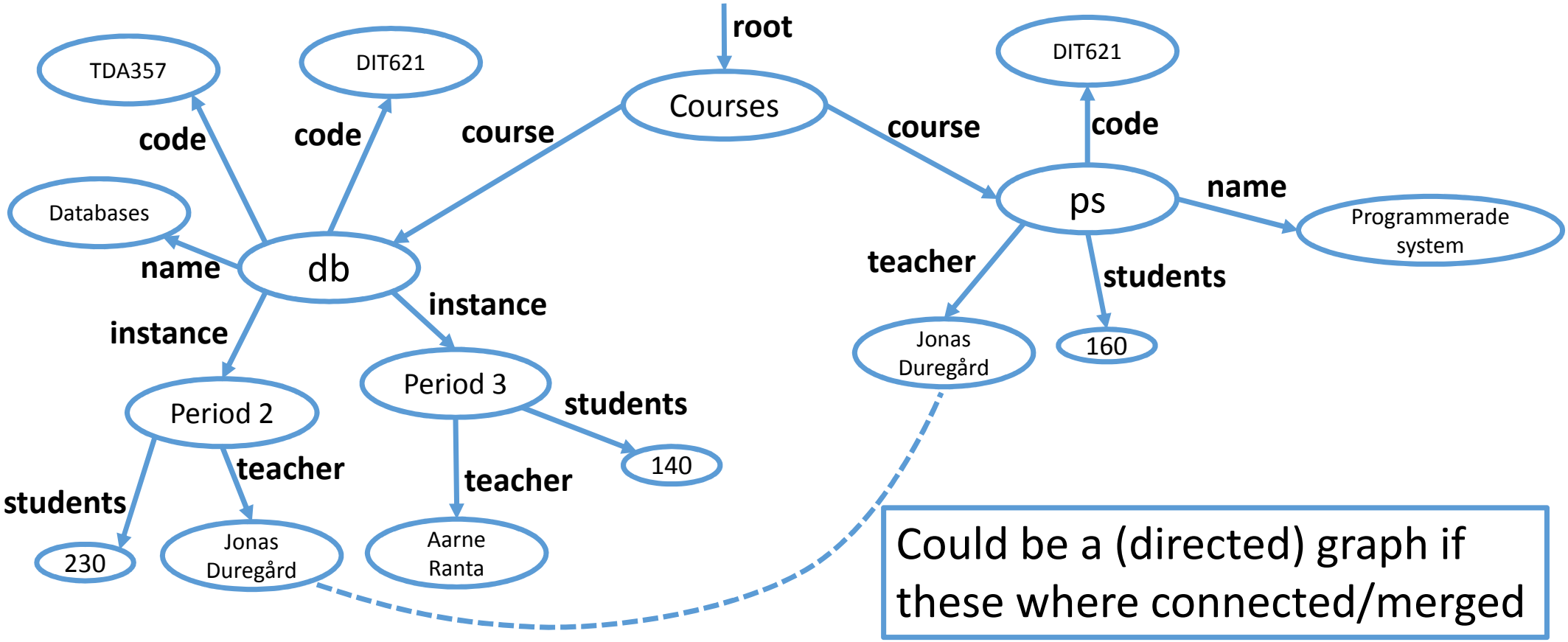
Semi-structured data (SSD)

- 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

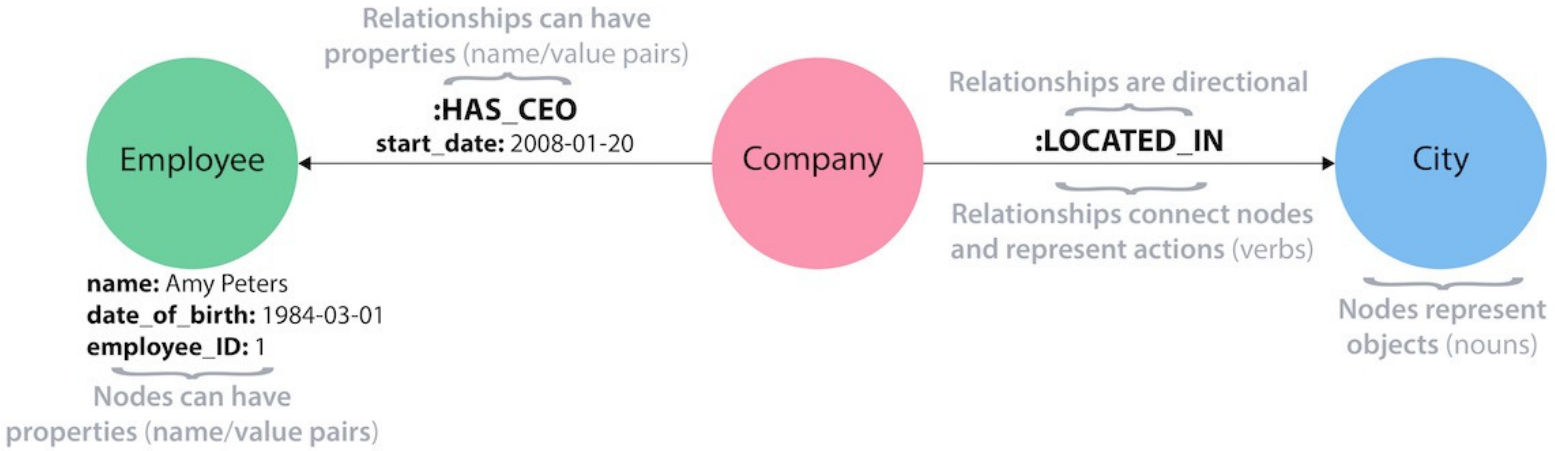
A different way of structuring data

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



Graph databases

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



Schemas for SSD

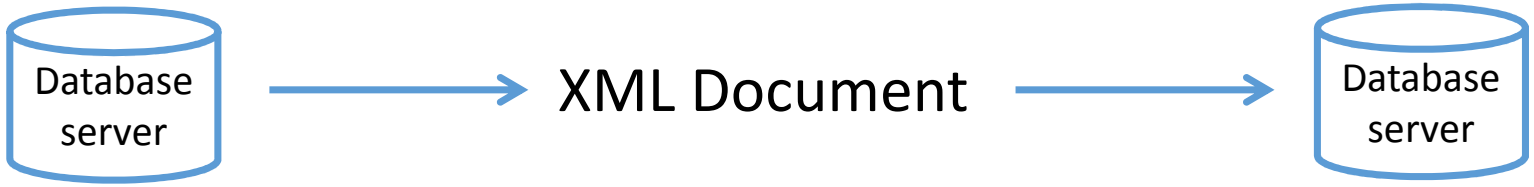
- Inherently, semi-structured data does not have schemas
  - The type of an object is its own business
  - The schema is given by the data
- We can of course decide to restrict graphs in any way we like. Examples:
  - Decide that all course nodes must have a code attribute
  - Each course node should have either a teacher or at least two instances
    - ... and each instance node must have a teacher
- Enforcing these restrictions automatically is a separate issue...

Examples of document-based SSD standards

- XML
  - Extensible Markup Language
  - Created in the 1990's
  - Syntax: <tag attribute="value"><other_tag/>also 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

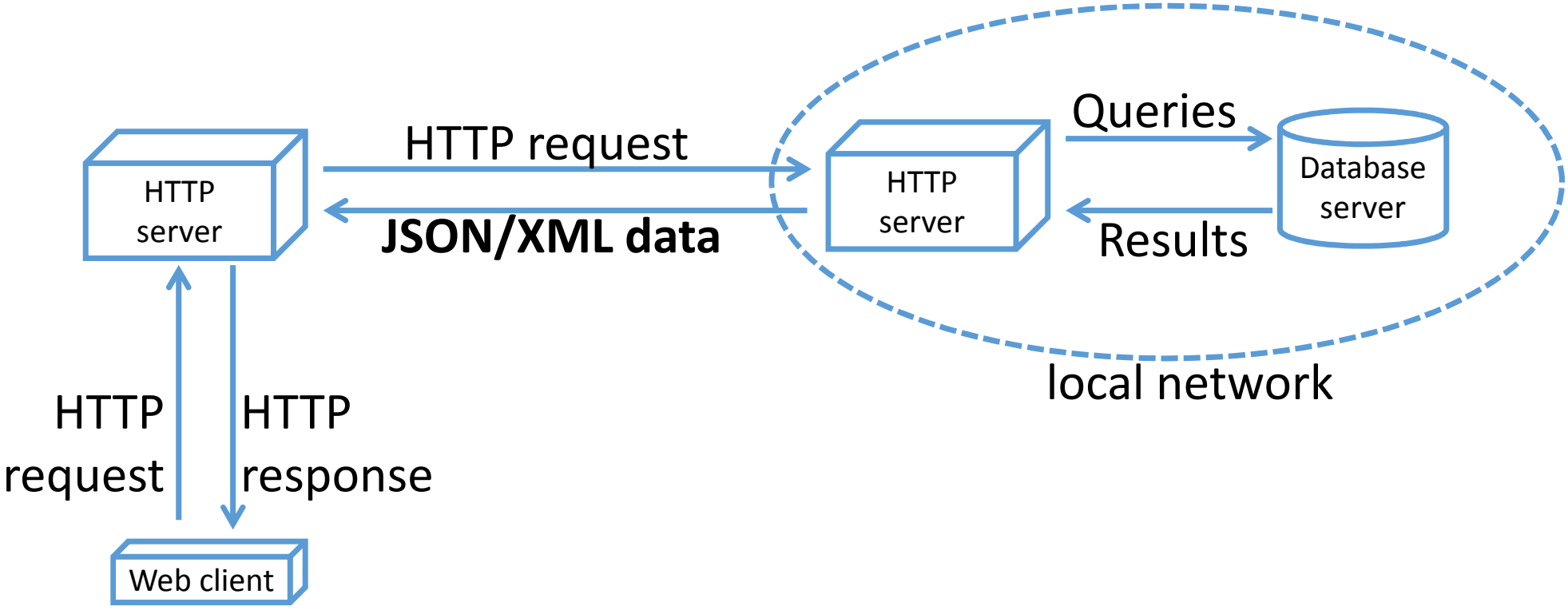
Data interchange formats

- 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



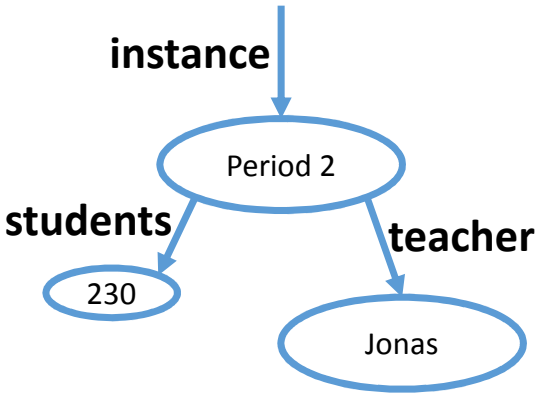
Cross domain communication

- 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:

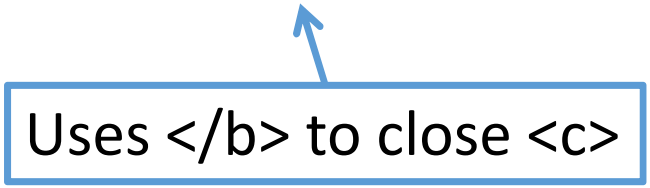
```
<instance name="Period 2">
  <teacher>Jonas</teacher>
  <students>230</students>
</instance>
```

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><b><c>text</c></b></a>`
  - Not allowed: `<a><b><c>text</b></c></a>`



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 en element seems odd
  - Maybe firstname/lastname should be attributes, and courses elements (the names feel "attributy" whereas a course feels more "entityish")

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>
```

```
<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>
```

Extra name elementto avoid mixing text and elements

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) persistant 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*)

Validation of XML documents

- Step 1: Check well-formedness
  - Not every text documnet is an XML document, before processing it in any way it needs to be parsed, and that validates that it is well-formed (every tag is closed, tags are correctly nested etc.)
- Step 2: Validating the data against a schema
  - The schema is provided separately from the data, in a special schema format
  - For XML, there are two popular standards: DTD and XML Schema
- Validation is different from constraints, they are not checked when we add data (since documents are generated, not constructed by insertions)
  - Validation is typically used when you receive a document from a third party (user input or data received from a remote server etc.)

XML Schema and JSON Schema

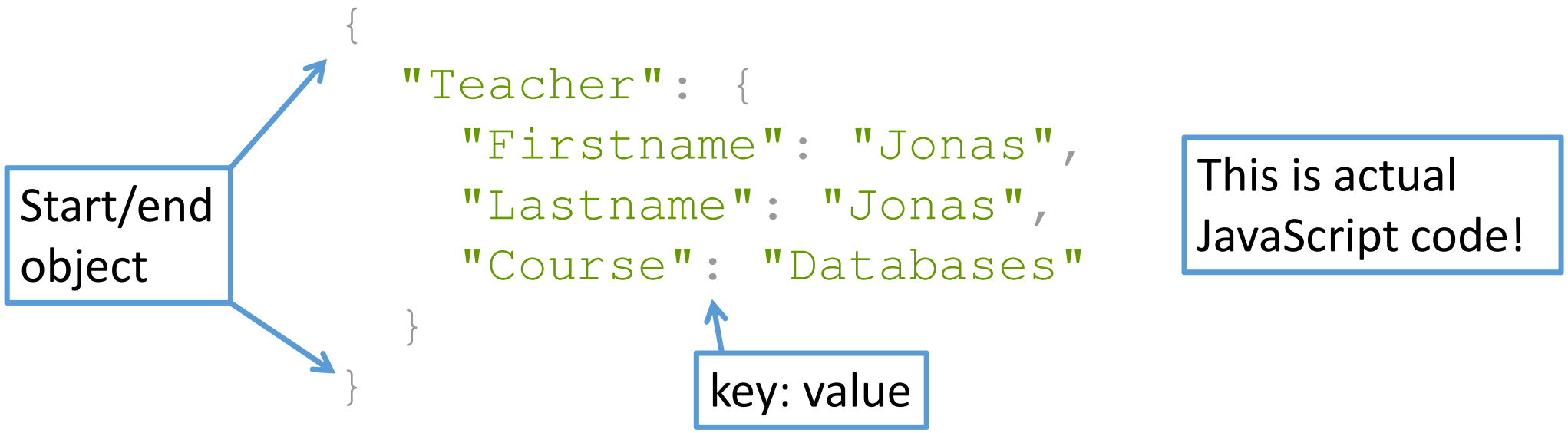
- Basic idea: Instead of using a separate language to describe schemas, we use XML to describe XML schemas and JSON to describe JSON schemas
- The schema is a document that describes the structure of other documents!
- Tomorrow we will have a much closer look at JSON schemas

An XML document describing other documents
(saying all teacher elements have first- and lastname attributes)

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="Teacher">
        <complexType>
        <attribute name="FirstName" use="required" type="string">
        <attribute name="LastName" use="required" type="string">
        </complexType>
    </element>
</schema>
```

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, "Teacher"
- The value of "Teacher" is an object containing three variables of type String



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 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
- From this year we are switching focus towards JSON
  - It has simpler syntax
  - It is growing quickly into the standard data format of the web

So what will I need to know for the exam?

- *Read* and understand XML documents
- *Read*, *write*, *validate* and *query* JSON documents (tomorrows lecture, and the exercise on Friday)
- For this years exam, any questions about JSON can also be answered using XML instead (but that requires learning about DTDs, XML Schema and XPath, which I do not cover in these lectures)
  - Most old questions about XML can be translated into corresponding JSON questions (replacing DTD with JSONSchema and XPath with JSONPath)