7 JSON and XML: Giving Messages a Structure

(some) Limitations of Relational Databases

- *Documents*: how to store a book or a system manual using a relational table?

  As one huge string? By introducing an attribute for each book chapter? For each paragraph? . . .
- *Data Interchange*: for example, for eBusiness, in which format shall messages be interchanged between partners?

  By writing emails? By exporting/importing relations?

Wanted

- A language to structure data in a flexible way would be helpful.
- A language to semantically annotate data would be helpful.
- This langauge should be easy to read for humans and machines.

Two Candidates

JSON (JavaScript Object Notation)

- Based on JavaScript Syntax
- Widely used
- `https://www.json.org/`; the standard (16 pages)

XML (Extensible Markup Language)

- Based on document markup languages (SGML, HTML, ...)
- Widely used, in particular the HTML variant
- the standard `https://www.w3.org/XML/`

Supported by all serious databases

Example Document in JSON

```
{
"PONumber"                : 1600,
"Reference"               : "ABULL-20140421",
"Requestor"               : "Alexis Bull",
"CostCenter"              : "A50",
"ShippingInstructions" : {
    "name"   : "Alexis Bull",
    "Address": { "street"  : "200 Sporting Green",
                 "city"    : "South San Francisco",
                 "state"   : "CA",
                 "zipCode" : 99236,
                 "country" : "United States of America" },
                 "Phone"   : [ { "type" : "Office", "number" : "909-555-7307" },
                               { "type" : "Mobile", "number" : "415-555-1234" } ] },
"Special Instructions" : null,
"AllowPartialShipment" : false,
"LineItems": [ { "ItemNumber" : 1,
                 "Part"       : { "Description" : "Hammer",
                                  "UnitPrice"   : 10.95,
                                  "UPCCode"     : 13131092899 },
                 "Quantity"   : 9.0 },
               { "ItemNumber" : 2,
                 "Part"       : { "Description" : "Screw Driver",
                                  "UnitPrice"   : 9.95,
                                  "UPCCode"     : 85391628927 },
                 "Quantity"   : 5.0 } ]
}
```

Examples

- key-value pair

  "name" : "Alexis Bull",

  "zipCode" : 99236
- array

  [1, 2]
- object

  { "type" : "Office", "number" : "909-555-7307" },

```
{"name"   : "Alexis Bull",
 "Address": { "street"  : "200 Sporting Green",
              "city"    : "South San Francisco",
              "state"   : "CA",
              "zipCode" : 99236,
              "country" : "United States of America" },
 "Phone" : [ { "type" : "Office", "number" : "909-555-7307" },
             { "type" : "Mobile", "number" : "415-555-1234" } ]
 }
```

Definition: JSON Value

JSON values are defined as follows.

- Any *number* is a JSON value.
- A *string* is a JSON value of the form "s" where s is a string in U^* and U is the set of all printable unicode characters except `"` and `\`. To account for those characters U additionally contains `\"` and `\\` as well as several special codes.
- If $v_1, \ldots, v_n$ are JSON values and $s_1, \ldots, s_n$ are pairwise distinct string values, then $\{s_1 : v_1, \ldots, s_n : v_n\}$ is a JSON value, called an *object*. In this case, each $s_i : v_i$ is called a *key-value pair* of this object.

  There is no order defined on the elements of an object.
- If $v_1, \ldots, v_n$ are JSON values then $[v_1, \ldots, v_n]$ is a JSON value called an *array*. In this case $v_1, \ldots, v_n$ are called the *elements* of the array.

  The elements of an array are ordered.
- `true, false, null` are JSON values.

In the case of arrays and objects the values v_i can again be objects or arrays, thus allowing an arbitrary level of nesting. Moreover, if $n = 0$, then the object/array is empty.

Access to JSON Values Using PostgreSQL

- path expressions built out of steps separated by -> or ->>,
- functions JSON_EACH, JSON_OBJECT_KEYS, JSON_TYPEOF,
- a postgresql instance

```
DROP TABLE PurchaseOrder;
CREATE TABLE PurchaseOrder (
Id INT NOT NULL,
Document JSON NOT NULL );
INSERT INTO PurchaseOrder VALUES
(123, '{"client":"GL", "nr_of_items":1, "items":  {"good":"luck"} }' );
```

Simple path expressions

```
SELECT * FROM PurchaseOrder;
SELECT Document->'client' as Client FROM PurchaseOrder;
SELECT po.Id, po.Document->'client' as client,
       po.Document->'items'->'good'
FROM PurchaseOrder po;
```

PostgreSQL JSON Path Expressions

- JSON data is internally represented by an object.
- JSON objects *o* have the following operations

o->*i*	get JSON array element *i* as JSON object
o->`'key'`	get JSON field 'key' as JSON object
o->>*i*	get JSON array element *i* as text
o->>`'key'`	get JSON field 'key' as text
o#>{*path*}	get JSON field according to path as JSON object
o#>>{*path*}	get JSON field according to path as text

```
SELECT Document #> '{items,good1}' FROM PurchaseOrder;
```

Non existing paths yield NULL values

Using JSON Addressing

JSON in WHERE

Need to obtain the underlying text to compare

```
SELECT po.Document->'items' AS Items
FROM PurchaseOrder po
WHERE po.Document->>'client' = 'GL';
```

JSON as a number in WHERE

Need to transform (*cast*) the text to a number to compare

```
SELECT po.Document->'client' AS Clients
FROM PurchaseOrder po
WHERE CAST(po.Document->>'nr_of_items' AS INTEGER) < 2;
```

Using JSON Addressing

Aggregate functions apply

```
SELECT SUM (CAST (po.Document->>'nr_of_items' AS INTEGER))
FROM PurchaseOrder po
WHERE po.Document->'client' = 'GL';
```

json_each

```
SELECT json_each (Document) FROM PurchaseOrder;
```

returns a table of keys and values (as JSON objects)

```
(client,"""GL""")
(nr_of_items,1)
(items,"{""good"":""luck""}")
```

XML Technologies: XML documents

Example Document in XML

```
<ORDER PONumber="1600">
  <Reference>ABULL-20140421</Reference>
  <Requestor>Alexis Bull</Requestor>
  <CostCenter>A50</CostCenter>
  <ShippingInstructions>
     <name>Alexis Bull</name>
     <Address>
       <street>200 Sporting Green</street>
       <city>South San Francisco</city>
       <state>CA</state>
       <zipCode>99236</zipcode>
       <country>United States of America</country>
     </Address>
     <Phone> ... </Phone>
  </ShippingInstructions>
  <SpecialInstructions/>
  <AllowPartialShipment>false</AllowedPartialShipment>
  <LineItems>
    ...
  </LineItems>
</ORDER>
```

XML Basics

- XML is a *Markup Language*; XML is a subset of SGML, which is a metalanguage standardized 1986 used to define the structure and content of documents. (Note: HTML is a specific application of SGML.)

  XML Markup is indicated by a *start tag* `<aTagname>` followed by an *end tag* `</aTagname>`, both enclosing the markup'ed part of the document.
- Start and corresponding end tag, including the enclosed part of the document, is called *element*; the name of the element is the name of the tag and the *content* of the element is the enclosed part.
- The content of an element may contain simply text without any further tags, only (other) elements, or a mixture of both.

  If the content of an element does not contain any further tags, the content is called *element text*.

  If the content of an element is built out of other elements only, it is called *element content*.

  In the remaining case it is called *mixed content*.
- A tag without content is called *empty tag* with shorthand notation `<aTagname/>`
- Start tags may be further described using attributes.

Attributes

- Elements with attributes:

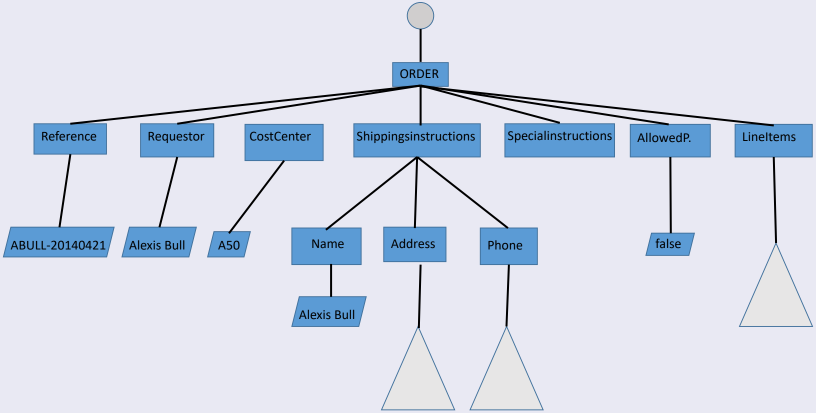
  $<$`aTagname attr`$_1$ `= "val`$_1$`"...attr`$_k$ `= "val`$_k$`"`$>$, $k \geq 1$.
- *empty* element with attributes:

  $<$`aTagname attr`$_1$ `= "val`$_1$`"...attr`$_k$ `= "val`$_k$`"/`$>$, $k \geq 1$.

  Example:

  `<Mitglied Organisation = "EU" Art = "member"/>`.
- An element may contain each attribute at most once.

XML Representation Structure: XML Tree



Document Order

- Tags of a document are ordered; the *document order* is defined by the sequence in which the start tags appear in the document.
- Attributes are not ordered.
- A XML tree is *ordered*, if depth-first search of the tree produces the document order.
- To each ordered XML tree there exists a textual representation called *serialization*, which reflects the document-order.

  Note: *A node in the tree represents the corresponding elements start and end tag; the element content and the tags and contents of the respective subtree are included.*

XML Technologies: XPath

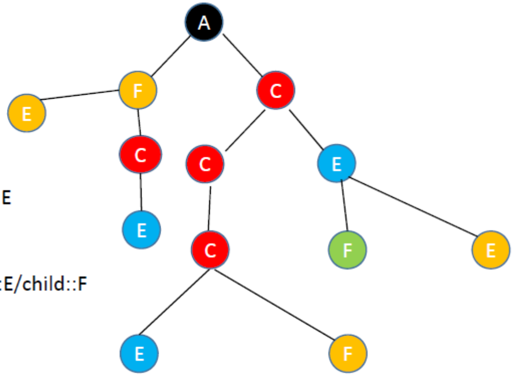
Querying XML documents.

- XPath is a language to locate subtrees of an XML tree by means of *location paths*.
- A location path is a sequence of '/'-separated *location steps*.
- Each location step is formed out of an *axis*, a *node test* and zero or more *predicates*:
  axis::node-test[predicate].

For a detailed presentation see:
http://www.w3.org/TR/xpath/ which gives a concise specification of XPath 1.0,
http://www.w3.org/TR/xpath-31/ which specifies the most recent version.

Some (very) simple location paths; short notation.

⚫	/child::A /A
🔴	/child::A/descendant::C /A//C
🔵	/child::A/descendant::C/child::E /A//C/E
🟢	/child::A/descendant::C/child::E/child::F /A//C/E/F
🟡	node not located



- A location path P is evaluated relative to a respective context and identifies a set of nodes, i.e., a set of root nodes of subtrees.
- A context is given by a *context node*, a *context position*. and a *context size*.
- A location path starting with '/' is *absolute* and otherwise *relative*.

  The context node of an absolute location path is the root node of the document. The root node of the document links to the root element of the document, which is the root node of the respective XML tree.

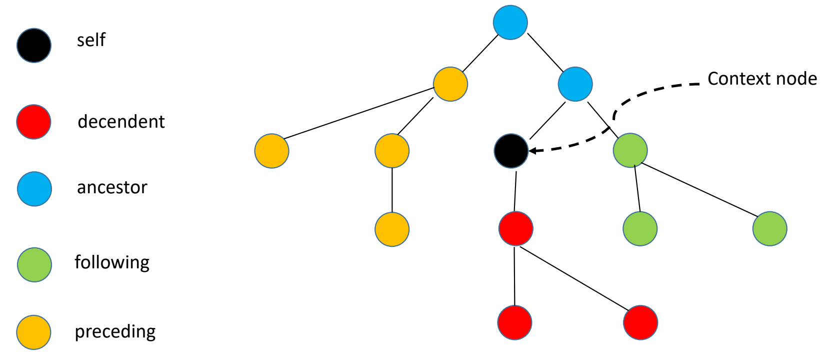
location step: *axis::node-test[predicate].*

Axes:[1]

- the *child axis* contains the children of the context node,
- the *descendant axis* contains the descendants of the context node; a descendant is a child or a child of a child and so on,
- the *parent axis* contains the parent of the context node, if there is one,
- the *ancestor axis* contains the ancestors of the context node; the ancestors of the context node consist of the parent of context node and the parent's parent and so on,
- the *following-sibling axis* contains all the following siblings of the context node,
- the *preceding-sibling axis* contains all the preceding siblings of the context node,
- the *following axis* contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants,
- the *preceding axis* contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors,
- the *attribute axis* contains the attributes of the context node,
- the *self axis* contains just the context node itself,
- the *descendant-or-self axis* contains the context node its descendants,
- the *ancestor-or-self axis* contains the context node and its ancestors.

[1] Cf. W3C XPath Recommendation http://www.w3.org/TR/xpath.

Locating all nodes inside the document.



location step: *axis::node-test[predicate].*

Node test

- Element name, resp. attribute name
- '$*$': any attribute or element
- `text()` only text nodes,
- `node()` no restriction

Examples

- /child::ORDER/child::Requestor
- /descendant::ShippingInstructions
- /descendant::ShippingInstructions/name[Alexis Bull]/parent::*/parent::Requestor
- /descendants::LineItems[position()=2]

SQL/XML

Outline

- How to *publish* (export) data in a relational databse as XML documents?
- How can we *extract* data out of a XML document and store (import) it in a relational database?
- How can we *map* SQL queries such that they can be executed on a XML document?

SQL/XML is part of the SQL:2003-Standard

- SQL/XML provides an additional type XML.
- but each vendor supports specific functionality

Datatype XML

- `XMLPARSE()` transforms an XML document given as a string into a value of type `XML`.
- `XMLSERIALIZE()` transforms a value of type XML to a string.
- SQL/XML defines a range of function that allow the generation of XML from a database query.

Example

XML Type

```
create table ABC (xtest xml);
```

- Insert from string

```
insert into ABC (xml '<Mondial><Land LCode = "D">...</Land></Mondial>')
```

- Insert from computed selection

```
insert into ABC XMLTYPE(
  SELECT XMLELEMENT ( NAME Land, XMLATTRIBUTES (L.LCode AS LCode),
            XMLELEMENT (NAME LName, L.LName),
              ( SELECT XMLAGG( XMLELEMENT ( NAME Lage,
                  XMLELEMENT (NAME Kontinent, La.Kontinent),
                  XMLELEMENT (NAME Prozent, La.Prozent) ) )
                FROM Lage La WHERE La.LCode = L.LCode),
            ( SELECT XMLAGG( XMLELEMENT ( NAME Mitglied,
                          XMLATTRIBUTES( M.Organisation AS Organisation,
                                         M.Art AS Art) ) )
              FROM Mitglied M WHERE M.LCode = L.LCode) ) AS Mondial
  FROM Land L WHERE LCode = 'D');
```

Example

```
CREATE TABLE ABC (XTest xml);
INSERT INTO ABC XMLTYPE(
  SELECT XMLELEMENT ( NAME Land, XMLATTRIBUTES (L.LCode AS LCode),
            XMLELEMENT (NAME LName, L.LName),
            ( SELECT XMLAGG( XMLELEMENT ( NAME Lage,
                    XMLELEMENT (NAME Kontinent, La.Kontinent),
                    XMLELEMENT (NAME Prozent, La.Prozent) ) )
                FROM Lage La WHERE La.LCode = L.LCode),
            ( SELECT XMLAGG( XMLELEMENT ( NAME Mitglied,
                            XMLATTRIBUTES( M.Organisation AS Organisation,
                                           M.Art AS Art) ) )
             FROM Mitglied M WHERE M.LCode = L.LCode) ) AS Mondial
  FROM Land L);

Tabelle ABC(XTEST):
==========================================================================
<LAND LCODE="A   ">
   <LNAME>Austria</LNAME><MITGLIED ORGANISATION="EU" ART="member"/>
</LAND>
---------------------------------------------------------------------------
<LAND LCODE="CH  "><LNAME>Switzerland</LNAME></LAND>
---------------------------------------------------------------------------
<LAND LCODE="D   ">
   <LNAME>Germany</LNAME>
   <LAGE><KONTINENT>Europe</KONTINENT><PROZENT>100</PROZENT></LAGE>
   <MITGLIED ORGANISATION="EU" ART="member"/><MITGLIED ORGANISATION="WEU" ART="member"/>
</LAND>
----------------------------------------------------------------------------
...
```

Example: Extraction via XPath

```
INSERT INTO ABC XMLTYPE(
  SELECT XMLELEMENT ( NAME Country,
    XMLELEMENT (NAME CoCode, L.CoCode),
    XMLELEMENT (NAME CoName, L.CoName),
    ( SELECT XMLAGG( XMLELEMENT ( NAME Location,
      XMLELEMENT (NAME Continent, La.Continent),
      XMLELEMENT (NAME Percentage, La.Percentage) ) )
      FROM Location La WHERE La.CoCode = L.CoCode) )
FROM Country L WHERE CoCode = 'TR');
```

```
CREATE TABLE ABCDB
(Continent VARCHAR(1024), Percentage VARCHAR(1024));
```

```
INSERT INTO ABCDB
SELECT * FROM
(SELECT
XPATH(XTest, '/descendant::LOCATION[1]/CONTINENT') Erg1,
XPATH(XTest, '/descendant::LOCATION[1]/PERCENTAGE') Erg2
FROM ABC);
```