Advanced Database and Information Systems

Riccardo Salvalaggio

19th of April, 2021

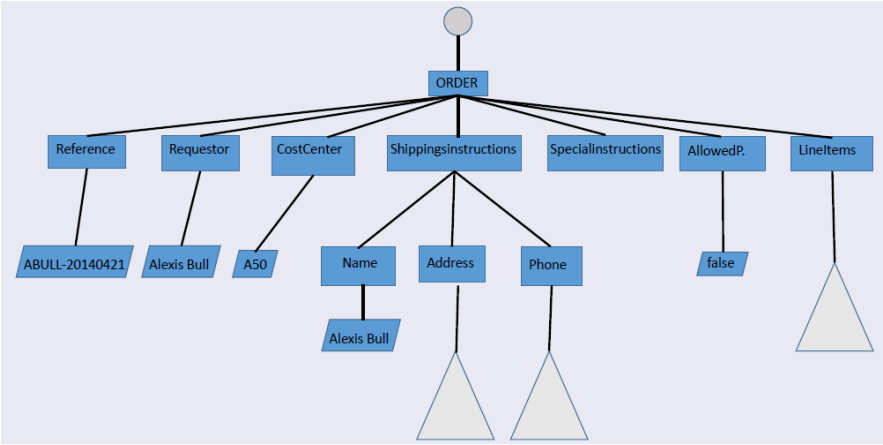
Contents

1 Semi-structured Data models: XML, XPath, XRel

Structured Data models are obsolete nowadays: inadequate representation, semantic overloading, difficulty on recursion, rigid schema, limited throughput. Now, semi-unstructured data format: data is no longer dependent on a schema (such as tables), is self-descriptive (good labels), more scalability. This needed is born because data is interlinked between each other (e.g. IoT, smart systems) and generated from DBs. Anyway, a standardized representation is needed: a new language to structure in a flexible way data and semantically annotate it.

1.1 XML

Extensible Markup Language, derived from HTML structure (use of tags), born in late 90s use a tree-structure to describe and organize data. Start- and end-tag with the inner part of the document are called *element*, the name of it is the name of the tag and the content is the enclosed part. If an element doesn't contain any other tag, the content will be called element text. A tag without content is called empty tag <tag/>.

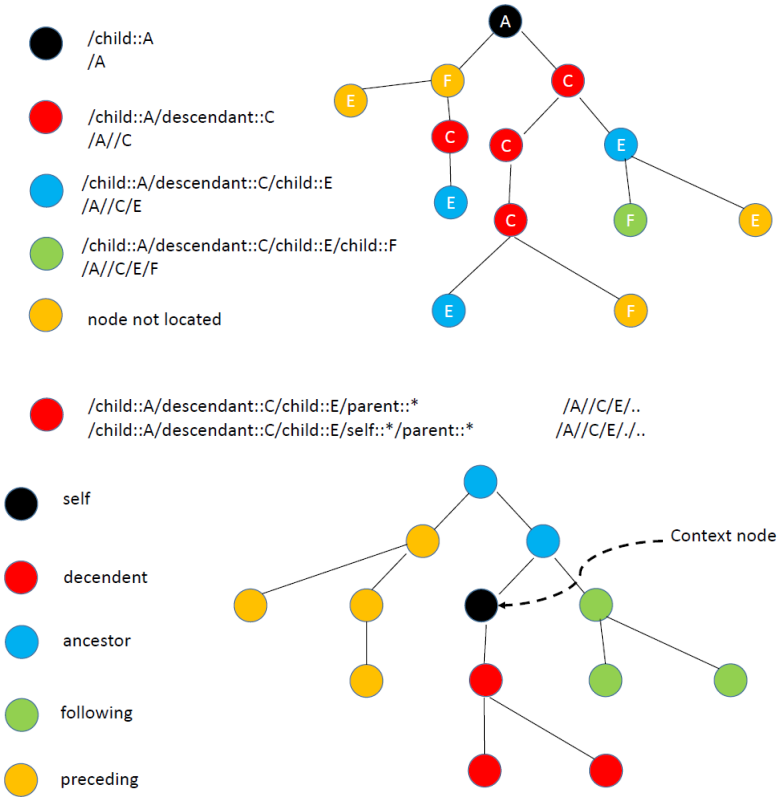


Tags are ordered, attributes not. An XML-tree is ordered (if depth-first search will reproduce the document order). The textual corresponding representation is called serialization. It is possible document-recursion (e.g. a message in XML could contain an XML-document) but name conflicts may appear. Definition of dbis-namespace:
<xmlns:dbis="http://.... ">
DTD (Document Type Definition): define attributes rules of a document (element and attributes types). A document conforms to XML syntax is *well-formed*, if it is also conformed to DTD is *valid.*
Definition of element types: **<!ELEMENT EName Content>**
Definition of attribute types: **<!ATTLIST EName Attr1 AttrType1 ....**
Elements types re global wrt DTD, attributes types are local.
Element content: EMPTY, ANY, (#PCDATA) - ,|*+?
Attribute definition:

Types: CDATA, (Val1|...|Valn), ID, IDREF
Val: #REQUIRED, #IMPLIED, val, #FIXED val

1.2 XPath

XPath is a language to locate subtrees inside an XML-tree by means of location paths (sequence of steps separated by '/'). Each location is formed of an axis, a nodetest and some (opt.) predicates:
Axis::node-test[predicate]



The path is evaluated relative to a respective context and locates a set of nodes. A context is given by position, size and the node. An absolute location path is the root node. $L_0 = \{r\}$ – root $L_i(k)$ set of nodes determined wrt context node k. $L_i = U_{k\ e\ Li-1} L_i(k)$. **Node-test:** element name, *, text(), node().
Predicates: Boolean expression.
e.g.: /descendant::*[child::Address]
/child::Orders/child::Order[child::CostCenter= ”A50”]/child::LineItems.
Context is a fact relative to a node set, is composed by: context size (cardinality), context position (its position relative to document order).

1.3 XRel: How to store XML data physically

Two approaches: Storing in column (CLOB - Character Large OBject, type + indexing), Mapping to relational tables (Interval-based, Path-based).
XRel: is a Path-Based approach to storage and retrieval of XML documents using relational databases. **Region of an element (node test):** pair of start and end position in the XML doc.
Region of an attribute: equal to the parent node plus one.

Output all the books published by “Addison-Wesley”.

```
//book[./publisher = "Addison-Wesley"]

SELECT E1.Start, E1.End
FROM Element E1, Element E2, Text T, Path P1, Path P2
WHERE P1.PathExpr like '#%/book'
   AND E1.pId = P1.PId
   AND P2.PathExpr like '#%/book#/publisher'
   AND E2.pId = P2.PId
   AND T.Value = "Addison-Wesley"
   AND T.Start > E2.Start
   AND T.End < E2.End
```

Use of queries with indexing.

1.4 XQuery

The mission of the XML Query project is to provide flexible query facilities to extract data from real and virtual documents. Collections of XML files will be accessed like DBs.
XQuery is a functional language, have a declarative semantics and is strongly typed. An item is an atomic value or a node.
Two ways to create element node:

```
<Order PONumber = "1800">
    Dummy
</Order>
```

```
element Order {attribute PONumber {"79100"}, "Dummy"}
```

```
<Order>
{ doc("Orders.xml")//LineItems }
</Order>
```

A **FLWOR-expression** is built out of ***for-, let-, where-, order- and return***-clauses. It evaluates to a stream of tuples, where the tuples are (variable-name, value) pairs.
Two main commands:
XMLTable: maps results of an XQuery into relational rows and columns.
XMLQuery: return XML-document
With XMLTable we refer directly to the path and use passing XMLDoc instruction. With XMLQuery we refer to doc and use RETURNING CONTENT FROM DUAL instruction.

Injection of elements

```
SELECT x.* FROM OrderTable,
     XMLTABLE(
        '<mySpecial>{
            let $a := //Orders//*[name="Alexis Bull"]/Address/*
            return <A>{$a}</A>
        }</mySpecial>'
        passing XMLDoc) x;
```

Reversal of structure[3]

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('let $a := //Order
      return
        <ShippingInstructions>{$a/ShippingInstructions/*}
        <Order>
           {for $b in $a/*
            where fn:not($b is $a/ShippingInstructions)
            return $b}
        </Order>
        </ShippingInstructions>'
passing XMLDoc) x;
```

Join

```
SELECT x.* FROM OrderTable,
     XMLTABLE('for $a in /Orders//LineItems/Item,
                   $b in /Orders//LineItems/Item
                   where $a/text() <  $b/text()
     return
       <result Left = "{ $a/ancestor/Order/attribute::PONumber }"
               Right = "{ $b/ancestor/Order/attribute::PONumber }"/>'
     passing XMLDoc) x;
```

Quantifiers `some`, `every` and conditional expressions

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a in //state["CA"]
    where some $b in $a/../zipCode/text()
    satisfies ($b > 1000)
    return $a/ancestor::Order'
    passing XMLDoc) x;
```

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a in //LineItems
    return
    if (count($a/Item) > 2)
    then <Result status="good"/> else <Result status="failure"/>'
    passing XMLDoc) x;
```

Position in a sequence

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a at $i in //LineItems/Item
    return
    <Result> {$a, $i} </Result>'
    passing XMLDoc) x;
```

XQuery built-in functions: E.g. aggregation

```
SELECT x.* FROM OrderTableA,
    XMLTABLE('for $a in //LineItems
    return <Result> {sum($a/Item) * count($a/Item)} </Result>'
    passing XMLDoc) x;
```

Comparison operators

- Atomic values: `eq,ne,lt,le,gt,ge`
- Sequences of atomic values: `=,!=,<,<=,>,>=`

  Evaluates to `true` if the comparison holds for at least one pair of values of the given sequences. Relation steht.

  Thus `(1,2)=(2,3)` and `(2,3)=(3,4)`, but `(1,2)!=(3,4)`.
- Nodes: `is`
  where it is referred to identity.
- Document order: `<<,>>`.
- Nested structure: `fn:deep-equal()`

Negation: `fn:not()`

2 JSON and MONGODB

2.1 JSON (JavaScript Object Notation)

It is a very lightweight data exchange format based on JavaScript used for data exchange over Web.
It is based on two basic constructs:
1. Array: comma-separated list of things enclosed by brackets.
2. Object: comma-separated set of pairs enclosed by braces.

```
[
  { "ISBN": "ISBN-10",
    "price": 80.00,
    "title": "Foundations of Database
    "authors": [ "Abiteboul",  "Hull"
    "publisher": "Addison Wesley",
    "year": 1995,
    "sections": [
      { "title": "Section 1",
        "sections": [
          { "title": "Section 1.1" },
          { "title": "Section 1.2" }
        ]
      },
      { "title": "Section 2" }
    ]
  }, … …
]
```

2.2 MongoDB

Document database designed for ease of development and scaling.
NoSQL database types:
1. Key-value stores
2. Document databases: data structure.
3. Wide-column stores: store columns of data.
4. Graph stores: store info about networks.
Uses JSON (BSON). MongoDB stores data records as BSON documents. BSON is a binary representation of JSON documents.

```
var mydoc = {
              _id: ObjectId("5099803df3f4948bd2f98391"),
              name: { first: "Alan", last: "Turing" },
              birth: new Date('Jun 23, 1912'),
              death: new Date('Jun 07, 1954'),
              contribs: [ "Turing machine", "Turing test", "Turingery" ],
              views : NumberLong(1250000)
            }
```

- Database = a number of collections.
- Collection = a list of documents.
- Each document stored in a collection requires a unique _id field that acts as a primary key.
CRUD operations: Create, Read, Update, Delete.

Insert commands: .insertOne(), .insertMany().

```
db.inventory.insertMany([
  { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
  { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
  { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
  { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
  { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);
```

```
db.inventory.find( {} )
SELECT * FROM inventory
```

```
db.inventory.find( { status: "D" } )
SELECT * FROM inventory WHERE status = "D"
```

```
db.inventory.find( { status: "A", qty: { $lt: 30 } } )
SELECT * FROM inventory WHERE status = "A" AND qty < 30
```

```
db.inventory.find( {
    status: "A",
    $or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]
} )
SELECT * FROM inventory WHERE status = "A" AND ( qty < 30 OR
item LIKE "p%")
```

```
{
 _id : "jane",
 joined : ISODate("2011-03-02"),
 likes : ["golf", "racquetball"]
}
{
 _id : "joe",
 joined : ISODate("2012-07-02"),
 likes : ["tennis", "golf", "swimming"]
}
```

```
db.users.aggregate(
  [
    { $project : { name:{$toUpper:"$_id"} , _id:0 } },
    { $sort : { name : 1 } }
  ]
```

```
{
 "_id": "10280",
 "city": "NEW YORK",
 "state": "NY",
 "pop": 5574,
 "loc": [
  -74.016323,
  40.710537
 ]
}
```

```
db.zipcodes.aggregate( [
  { $group:
    {
      _id: { state: "$state", city: "$city" },
      pop: { $sum: "$pop" }
    }
  },
  { $sort: { pop: 1 } },
  { $group:
    {
      _id : "$_id.state",
      biggestCity:  { $last: "$_id.city" },
      biggestPop:   { $last: "$pop" },
      smallestCity: { $first: "$_id.city" },
      smallestPop:  { $first: "$pop" }
    }
  },
] )
```

Figure 1: Largest and smallest cities by state.

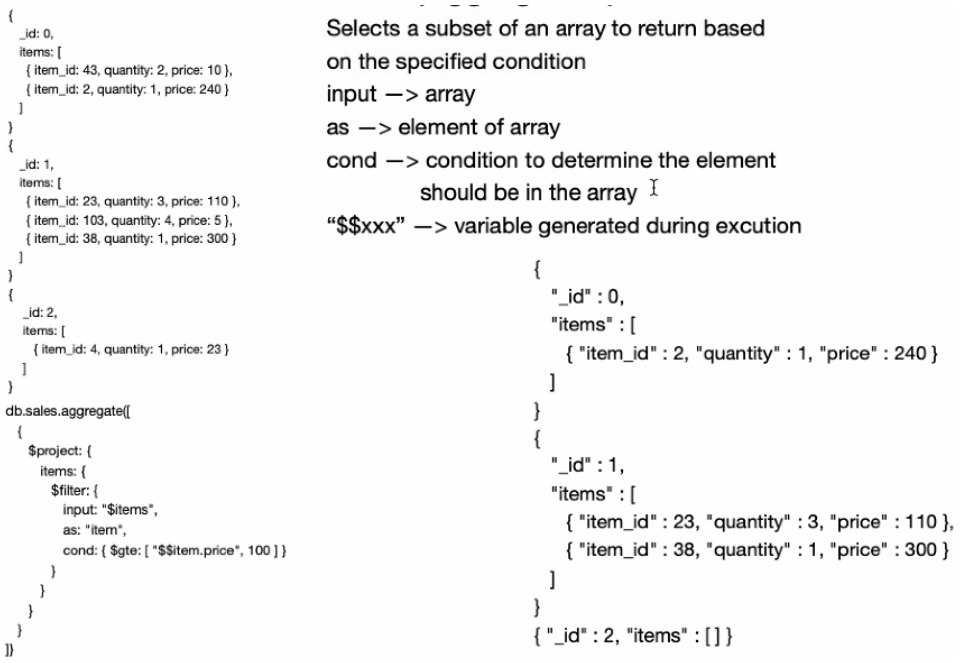


Figure 2: filter.

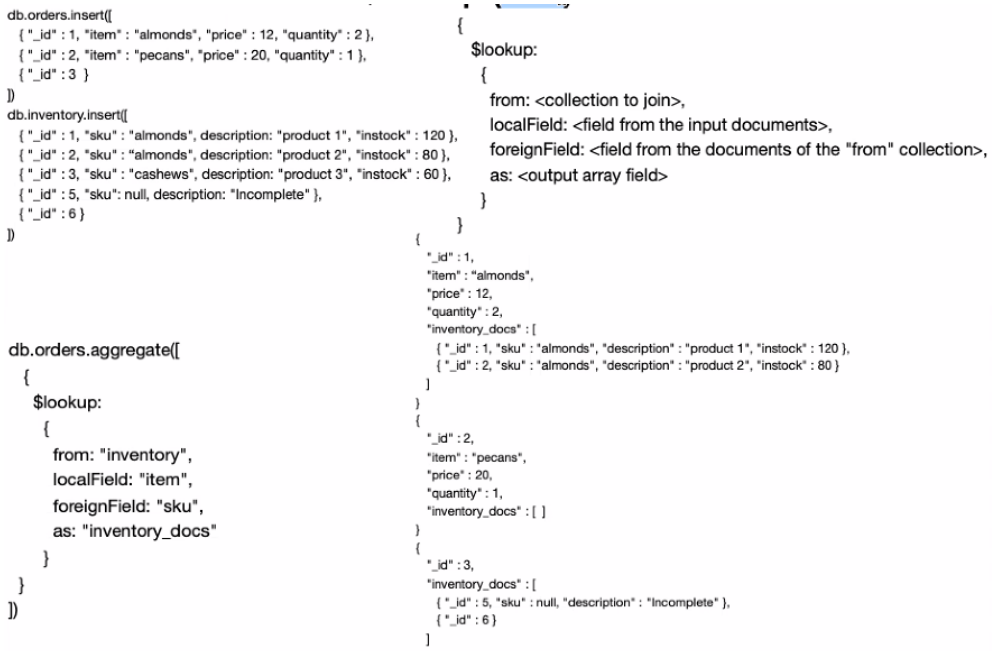


Figure 3: lookup(join).

Figure 4: push.

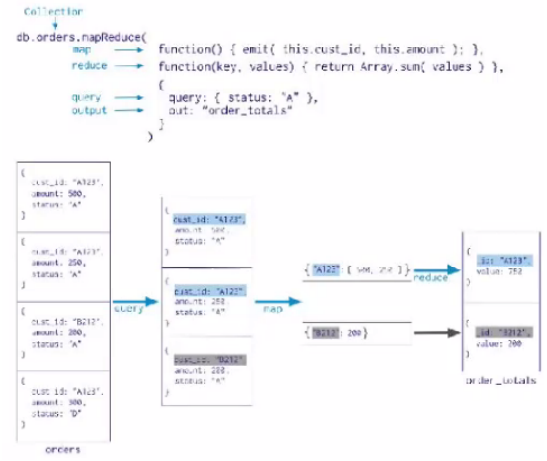


Figure 5: Map-reduce:Used to condense large volumes of data and emits key-value pairs. For keys with multiple values, MONGODB applies the reduce phase and then stores results in a collection.