Advances in Data Management
Semi-Structured Data Management

1 Semi-Structured Data

Semi-structured data is data that

- is self-documenting: it is not necessary to consult a schema in order to understand the data; indeed, there may be no schema associated with the data;
- is not rigidly structured: there may be aspects of the schema that are missing in parts of the data; and, conversely, parts of the data that do not correspond to objects in the schema.

Such data is not completely unstructured, and typically it can be modelled as a *tree* or a *graph*.

JSON, discussed earlier, is an example of a semi-structured data format.

2 Object-Oriented and Object-Relational Databases

Let us first take a step back and look briefly at the data types, storage and indexing capabilities typically supported by Object-Oriented (OO) and Object-Relational (OR) DBMSs.

Recall from Week 1 that the development of OO DBMSs arose from:

- the limited data types supported by the original relational model (just sets of tuples of atomic values) is insufficient for applications that need to handle more complex objects e.g. in areas such as computer aided design, computer aided software engineering, multimedia applications, GIS; and
- the emergence of OO programming languages such as C++, with their support for object identity, abstraction, encapsulation of data and code, and inheritance.

The advent of OO DBMS led the major relational DBMS vendors to extend their DBMSs with OO-like features such as object identifiers, user-defined types (UDTs), user-defined functions (written in a 4GL or external PL), structured types, collection types (sets, bags, lists, arrays), and type inheritance over UDTs.

These enhancements required extensions to support the storage, indexing and retrieval of objects:

- A class of objects can be stored within a table, also supporting indexes (on primary and secondary key attributes) for fast access to subsets of objects through different search attributes.
- Additional support is needed for Character Large Objects (CLOBS) - to represent large documents - and Binary Large Objects (BLOBS) - to represent large multimedia objects such as images and videos - that are not easily decomposed into tabular form.

CLOBS and BLOBS are stored in a separate file, or files, since they are typically too big to be stored "inline" with other shorter data fields within one physical record; a pointer to the LOB is stored "inline" within the record (c.f. the use of pointers in the leaves of unclustered B+ trees and R-trees).

- It is also possible store a LOB within a (clustered) B+ tree: the LOB is split into contiguous "chunks" that are stored in the leaves of the B+ tree; the search keys in the inner nodes are "byte offsets", allowing particular parts of the LOB to be found quickly by specifying a byte offset or range of bytes required.

  Secondary indexes over such LOBs can also be supported, e.g. by creating unclustered B+ trees that contain pointers to LOB "chunks" in their leaves.

Typically, LOBs are manipulated by specialist software external to the DBMS.

3 XML databases

3.1 Historical Background

The growth of the Web in the 1990s brought about the need for a common data format for different Web applications to exchange data.

Using proprietary data formats was sufficient for exchanging data within one business, or even for communicating data across a small number of partners, but was not scalable.

This gave rise to the need for a standard common format for data exchange between applications. The solution to this problem came with the advent of **XML** (Extensible Markup Language), which has since been standardised by the W3C. XML is a subset of the earlier SGML document markup language.

XML gave a significant impetus to web-based business-to-business (B2B) applications. Data was stored in relational databases and XML was used as a medium to transfer data between businesses.

XML quickly became the *de facto* standard for deploying applications that managed large volumes of data and that needed to be able to communicate with other businesses or to expose their data on the web.

This led to *XML-enabled databases* (XEDs), which initially adopted a simple strategy to store XML data: each XML document is decomposed and its data is stored within conventional database tables.

The popularity of XML also helped it gain ground in the application area of SGML itself, including the printing and publishing industries.

Along with this shift in use, a new challenge became apparent. The XML data produced in these application domains were no longer small or medium-sized documents. However, XML-enabled databases were not capable of supporting documents of this size, since decomposing and recomposing huge documents meant a high number of join operations, leading to unacceptable performance, both in retrieving documents and in querying them.

This technology gap was covered by the emergence of *native XML databases* (NXDs): database systems specifically developed for storing XML documents. They were able to preserve the nesting

and ordering of sub-documents and had better performance when handling large XML documents.

A third type of XML database system were OO and OR DBMSs that had existed before NXDs and that were subsequently extended to be able to handle XML documents natively. Their ability to handle both relational/OO data and XML data gave these systems the initial characterisation as "hybrid". However, nowadays, the category of *XML-enabled databases* is regarded as encompassing these systems.

Generating an XML document (or XML fragment) from structured data is known as XML *publishing* or *composition*.

The reverse process of generating structured data from an XML document or XML fragment is known as XML *shredding* or *decomposition*.

Being able to perform such data transformations between XML data and structured data within a single DBMS greatly simplifies this task.

3.2 XML Terminology

Data in XML documents is represented as strings of text. This data includes text markup that describes the data. A particular unit of data and markup is called an *element*.

In languages such as HTML, SGML and XML, the mark up of documents takes the form of *tags*, with `<tag>` and `</tag>` indicating the start and the end of an element.

Markup in an XML document looks similar to that in an HTML document. However, there are some key differences:

- XML is a meta-markup language: it doesn't have a fixed set of tags.
- To enhance interoperability, groups of people (communities, disciplines, industry sectors etc.) may agree to use only certain tags, e.g. as defined in an XML Schema — see below.
- Markup in XML only describes a document's structure; it doesn't say anything about how to display it (display instructions can be expressed in another W3C-recommended language called CSS - cascading style sheets).

An XML document can be viewed as a *tree*, whose nodes are the elements of the document. The sub-elements and the *attributes* of an element constitute its child nodes. Any non-marked-up *text* content of an element is also modelled as a child node of the element.

Data-centric XML documents are typically:

- used for machine (rather than human) consumption
- have fairly regular, fine-grained data

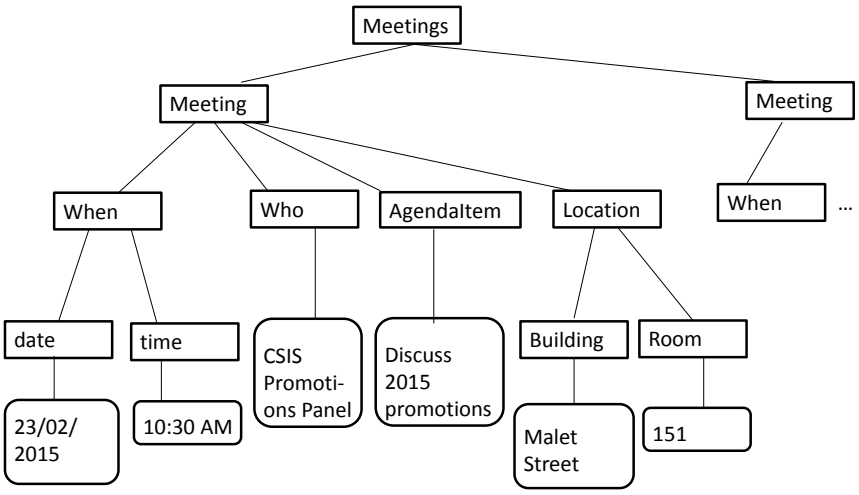

Figure 1: Tree representation of the Example Data-Centric document. **Exercise**: complete the figure to show the rest of the document

Example Data-Centric document:

```
<Meetings>
   <Meeting>
     <When date="23/02/2015" time="10:30AM" />
     <Who>CSIS Promotions Panel</Who>
     <AgendaItem>Discuss 2015 promotions</AgendaItem>
     <Location>
        <Building>Malet Street</Building>
        <Room>151</Room>
     </Location>
   </Meeting>
   <Meeting>
     <When date="27/02/2015" time="11:30AM" />
     <Who>Physics Staff Meeting</Who>
     <AgendaItem>Apologies for Absence</AgendaItem>
     <AgendaItem>Minutes of Previous meeting</AgendaItem>
     <AgendaItem>Head of Department report</AgendaItem>
     ...
     <Location>
        <Building>Malet Street</Building>
        <Room>250</Room>
     </Location>
   </Meeting>
...
```

```
</Meetings>
```

Document-centric XML documents are typically:

- designed for human consumption
- may not have regular structure
- have coarse-grained content (the smallest independent data unit may be a document itself)
- have "mixed" content (elements can have both other elements and text as child nodes)

Example Document-Centric document:

```
<Meetings>
    <Meeting>
        Please can
        <Who>CSIS Promotions Panel members</Who> come to the
        <Location> <Building>Malet Street</Building> building, room
                   <Room>151</Room> </Location> on
        <When date="23/02/2015" time="10:30AM" /> to
        <AgendaItem>discuss 2015 promotions</AgendaItem>
    </Meeting>
    ...
</Meetings>
```

Optional Reading: Chapter 23 of Silberschatz, Korth and Sudarshan — Sections 23.1 and 23.2.

3.3 XML Schemas

Unlike traditional relational or object-oriented data (so-called "structured data"), XML documents do not have to be associated with a schema. However having a specified schema can be useful when XML data needs to be stored and processed in large quantities.

The first XML schema definition language was the **Document Type Definition (DTD)**. Nowadays **XML Schema** is the W3C recommended language for specifying XML schemas.

Optional Reading: Chapter 23 of Silberschatz, Korth and Sudarshan — Section 23.3.2.

3.4 Querying XML data (optional)

The **XPath** and **XQuery** query languages are W3C standards for querying XML data.

3.4.1 XPath

XPath is used for specifying *path expressions* through XML data, viewed as a tree.

Each node in a path expression is delimitted by the symbol `/` (the symbol `.` is used for the same purpose in the object-oriented query language OQL — see Week 1).

An XPath expression is evaluated with respect to a *context node*. An *axis specifier* such as 'child' or 'ancestor' specifies the direction within the tree to navigate from the current context node.

The XPath syntax comes in two flavours: The abbreviated syntax is more compact and allows XPath expressions to be written and read more easily using familiar constructs. The full syntax is more verbose, but allows for more options to be specified. We will use here the abbreviated syntax.

There are two different kinds of paths:

- absolute location paths
- relative location paths

An absolute location path starts at the "top" of the document tree, with navigation along the 'child' axis, `/`, being the first step. For example:

```
/Meetings/Meeting/AgendaItem
```

selects the top-level Meetings node (the root of the tree), then the Meeting elements that are children of this Meetings node, and then the AgendaItem elements that are children of each of these Meeting elements.

This could be expressed more simply, using the 'descendant-or-self axis', `//`:

```
//AgendaItem
```

which returns all AgendaItem elements in the document.

A relative location path is a sequence of location steps each separated by `/` that is evaluated with respect to some other context node. For example:

```
Location/Building
```

would return a non-empty result if the current context node is one of the Meeting nodes in the document.

Attribute names are prefixed by `@`. For example:

```
/Meetings/Meeting/When/@date
```

returns the dates of all meetings.

Predicates are written as expressions within square brackets and can be used to restrict a node-set to select only those nodes for which some condition is true. For example

```
/Meetings/Meeting/Location[../When[@date="23/02/2015" and @time="10.30AM"]
```

returns the locations of meetings held on 23/02/2015 at time 10.30AM. Note here also the use of the 'parent' axis, `..`, to navigate one level up the tree to the parent Meeting node before coming down again to its When node.

The following example shows how to select one of a sequence of sibling elements:

```
/Meetings/Meeting/AgendaItem[2]/../When[@date="23/02/2015" and @time="10.30AM"]
```

This returns the the second agenda item of meetings held on on 23/02/2015 at time 10.30AM.

Finally, `text()` can be used match text nodes, removing the surrounding tags. So, for example,

```
/Meetings/Meeting/Location[./Building="Malet Street"]/Room/text()
```

returns the text value of all rooms in the Malet Street building, without the surrounding Room tags. Note here the use of the 'self' axis, `.` , to express a condition whose evaluation starts at the current context node.

3.4.2 XQuery

XQuery incorporates XPath as a sublanguage, but is more expressive as it can perform joins between different parts of an XML document, can create new XML trees, and can sort the output that it produces.

For example, the XQuery query

```
<AgendaItems>
    {
      //AgendaItem
    }
</AgendaItems>
```

returns a list of the AgendaItem elements in the document, under one AgendaItems parent element.

The use of curly brackets `{` and `}` indicates that the enclosing expression should be evaluated.

If we omitted the curly brackets, the result would simply be the string "//AgendaItem" withn the enclosing AgendaItems tags.

A slightly more complex example:

```
<AgendaItems count = '{ count(//AgendaItem) }'>
   {
      //AgendaItem
   }
</AgendaItems>
```

is similar, but returns also the number of AgendaItem elements as an attribute of the AgendaItems element.

More generally, XQuery supports FOR ... LET ... WHERE ... ORDER BY ... RETURN expressions (so-called FLWOR expressions):

- The FOR clause is like the FROM clause of SQL, and specifies variables that range over the results of XPath expressions. When more than one variable is specified, the Cartesian product of the possible values the variables can take is returned, similarly to the SQL FROM clause.
- The LET clause binds a variable to the collection of elements returned by a path expression, for convenience of query formulation.
- The WHERE clause, like the WHERE clause of SQL, performs additional tests on the values of variables in the FOR clause.
- The ORDER BY clause allows sorting of the output.
- Finally, the RETURN clause allows the construction of results in XML.

To illustrate:

```
for $i in (1, 2, 3)
return
   <tuple>  <i> { $i } </i> </tuple>
```

The variable `$i` is bound to each member of the collection (1, 2, 3) in turn, and the output returned is:

```
<tuple><i>1</i></tuple>
<tuple><i>2</i></tuple>
<tuple><i>3</i></tuple>
```

(note that the 'for' clause preserves the ordering of the collection that it is iterating through when it creates tuples).

A 'let' clause binds a variable to the entire result of an expression. If there are no 'for' clauses, then a single tuple is created. For example:

```
let $i := (1, 2, 3)
return
  <tuple>  <i> { $i } </i> </tuple>
```

results in:

```
<tuple><i>1 2 3</i></tuple>
```

When 'for' and 'let' clauses are combined, the variable bindings of 'let' clauses are added to the tuples generated by 'for' clauses. For example:

```
for $i in (1, 2, 3)
let $j := ('a', 'b', 'c')
return
   <tuple><i>{ $i }</i><j>{ $j }</j></tuple>
```

results in:

```
<tuple><i>1</i><j>abc</j></tuple>
<tuple><i>2</i><j>abc</j></tuple>
<tuple><i>3</i><j>abc</j></tuple>
```

A more realistic FLWOR query is the following:

```
for $m in /Meetings/Meeting/When
let $d := $m/@date
where $m/@time < "10.30AM"
order by $d
return <date> { $d } </date>
```

Applied to our earlier document, this returns the dates of meetings held at 10.30AM, in ascending order of the date.

This query lists who was present and the agenda items of each meeting:

```
for $m in //Meeting
let $a := $m/AgendaItem
return
  <Meeting> {
              <Who>$m/Who</Who>
              <AgendaItems> $a </AgendaItems>
            }
  </Meeting>
```

This query lists who was present at each meeting together with the number of agenda items:

```
for $m in //Meeting
let $a := $m/AgendaItem
return
  <Meeting> {
              <Who>$m/Who</Who>
              <count> { count($a) } </count>
            }
  </Meeting>
```

Its output is:

```
<Meeting>
   <Who>CSIS Promotions Panel</Who>
   <count>1</count>
</Meeting>
<Meeting>
   <Who>Physics Staff Meeting</Who>
   <count>2</count>
</Meeting>
```

3.4.3 SQL/XML

SQL/XML is a set of extensions to SQL for transforming between XML data and relational data, and for embedding XQuery into SQL. It is part of the SQL 2003 standard.

Key features of SQL/XML are the provision of a native XML data type for storing XML documents, and a set of functions for transforming between XML data and relational data.

Optional Reading: Chapter 23 of Silberschatz, Korth and Sudarshan — Sections 23.4 (on XPath and XQuery) and 23.6.3 (on SQL/XML).

3.5 Storing and indexing XML data

Today's XML-enabled databases store XML data in a variety of ways:

- shredding data-centric XML documents into a tabular reprensentation;
- storing document-centric XML documents as CLOBs;
- partial shredding of hybrid data- and document-centric XML;
- supporting a native **XML data type** for which tabular, CLOB-based or other storage methods provide the underlying implementation.

Special-purpose indexes are also typically supported for indexing XML data. There are several possible types of indexes:

- *Value indexes*, on a particular element or attribute in the data, e.g. to support fast retrieval of data values satisfying a particular condition in an XPath or XQuery query. These are implemented using conventional DB indexing approaches (hashing, B+ trees).
- *Full text indexes*, which index the lexical tokens appearing within text or within attribute values in the XML data. These are implemented using text indexing techniques adopted from Information Retrieval (inverted indexes).
- *Structural indexes*, for fast access to XML data based on path expressions. This class of indexes index *paths* (i.e. sequences of nodes) occurring in the XML document structure and they return the list of nodes reachable by a path.

  Such structural indexes can be implemented by treating paths as strings, and using these strings as search keys within a B+tree structure.

Optional Reading: Chapter 23 of Silberschatz, Korth and Sudarshan — Section 23.6.2.5.

3.6 JSON

The relative verbosity of XML has recently led to increasing adoption of JSON (JavaScript Object Notation) as a means of storing and exchanging semi-structured data.

See Homework Readings 2 and 3 from the Notes on "NoSQL, NewSQL and Big Data" for examples of JSON and for discussions of its use within document stores.

In this sense, Document Stores can be viewed as descendants of native XML databases.

Optional Reading

Putting it all together in a state-of-the-art RDBMS:

Read (for interest) Chapter 28 of A. Silberschatz et al. (2011), describing the main features of the Oracle DBMS.

4 Graph Data Management

Graph data storage, data models, query languages, and rule languages have been a topic of research for many decades. For example, see this presentation
`http://www.slideshare.net/infinitegraph/an-introduction-to-graph-databases`
for an overview of the history of graph databases, and a comparison of recent graph DBMS products.

In recent years there has been a resurgence of academic and industry interest in graph databases, due to the generation of large volumes of data from web-based and pervasive applicatioons centered on *relationships* between entities: e.g. the web graph itself, social and collaboration networks, communications networks, transport networks, biological networks, customer relationships, workflows, business processes etc.

Graph databases are characterised as those whose data storage, indexing and access methods are oriented towards effective support of graph algorithms such as graph traversal, subgraph matching, breadth-first/depth-first search, shortest path finding etc. (see presentation referenced above).

Some graph DBMSs (e.g. Neo4J) use so-called "native graph storage" and "index-free adjacency" to store their data: this basically means that nodes and edges of the graph are implemented as physical records, and the links between them are implemented as pointers. This avoids the need to use additional indexes in order to traverse paths in the graph, and can achieve faster graph traversal.

Other systems store the data graph as a set of relational tables or object classes. Others still use *triple stores* (see below) or document stores or key-value stores.

There are several different graph data models, e.g.: basic graph models comprising nodes and edges; Property graph models (as in Neo4J for example); and hypergraph data models[1].

[1]See Renzo Angles, Claudio Gutierrez, "Survey of graph database models", ACM Computing Surveys 40(1), 2008

Early research into graph databases proposed several declarative query languages[2], whose ideas have fed into the design of today's graph query languages.

There is as yet not a single standard graph data model or query language. The Linked Data Benchmark Council (LDBC : **http://www.ldbcouncil.org/**) is developing benchmarks and standards for graph data and RDF data.

Different graph DBMS products therefore support different graph query languages. For example, Cypher is a declarative graph query language originally developed as part of Neo4J which is now supported by several other products too (e.g. SAP HAHA Graph, RedisGraph, AgensGraph, Memgraph).

Commercial graph DBMSs generally support full acid transactions (e.g. Neo4J) as well as aiming for high performance, scalability and availability.

For example, Neo4J's Causal Clustering approach[3] provides a set of Read-Write servers (called Core Servers) and a set of Read-only servers (called Replica Servers).

A transaction is acknowledged to the end user application as having committed once it has been successfully committed locally by a *majority* of the Core Servers. If the set of Core Servers suffer enough failures so that transactions can no longer be committed, then they all become Read-only servers.

Replica Servers help in achieving high performance and scalability of graph queries and analytics. Transactions are replicated asynchronously from the Core Servers: from time to time a Replica Server will contact ('poll') a Core Server for any new transaction log records since the time of its last contact, and the Core server will send it these log records to apply to its local copy of the database.

Many Replica Servers can be supported by a relatively small number of Core Servers. The failure of a Replica Server does not affect the cluster's availability or fault tolerance properties, just its query throughput.

The cluster of servers is able to support *causal consistency*, meaning that updates that are written by an application will be visible to it on subsequent reads (i.e. 'read-your-own writes' semantics):

After one of its transactions T completes, a client application can ask the system for a 'bookmark', $b(T)$, that represents the state of the data subsequent to T's updates. The client can then pass $b(T)$ as a parameter of a subsequent transaction, T', that it executes, and the system will ensure that only servers that have processed T's updates will be used to execute T'.

5 RDF Stores

RDF can be regarded as a graph-based data model, but with some additional features such as properties being able to appear as the subject of a triple (i.e. as a graph node), not just as the predicate; the mixture of instance and schema objects within the same graph; and the ability to support inferencing through the RDF/S inference rules.

[2]See Peter T. Wood "Query languages for graph databases", SIGMOD Record 41(1), pp 50-60, 2012

[3]See https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/introduction

Similarly to other semi-structured data, RDF data can be stored and indexed in a variety of ways.

The survey by Luo et al. (see Reading) identifies three different perspectives on storing and indexing RDF data: a relational perspective; an entity perspective; a graph perspective.

We look briefly at the first one here. With this relational perspective, there are two major approaches to storing the RDF data:

1. As a single table, with schema (*subject, predicate, object*) — i.e. as a so-called *Triple Store*.
2. As a single table that has columns for the subject, the object, and each different predicate in the dataset.

With both approaches, resource URLs/URIs are typically mapped to unique, shorter, system-generated identifiers and are stored in a separate table. It is the system-generated identifiers that are then used inside the main data table(s).

With approach 1, clustered B+ tree indexes are constructed for a subset of the permutations of the subject (s), predicate (p) and object (o), in order to support effcient data retrieval.

For example, the HexaStore system creates 6 different clustered B+ tree indexes, on these permulations:

spo, sop, pso, pos, osp and ops

Naively, this would lead to a 6-fold replication of the data (but see Luo et al. pages 7-8 for optimisations).

There are also *quadruple stores* that can store quadruples of the form *(s,p,o, graphID)* for datasets comprising a set of distinct RDF graphs (rather than just one overall graph).

With approach 2, the single 'wide' table is actually divided into several smaller tables each containing a subset of related predicates as its columns; this is so as to avoid the many NULL values that would occur if storing a single 'wide' table.

At its most extreme, this approach employs one table, with two columns (subject and object), for each predicate. This is known as *column-oriented storage* and is also found in other types of DBMS (relational, NoSQL).

With all these RDF data storage approaches, it is possible to support additional indexes to further speed up query processing over the RDF data:

- Full-text indexes over the Literals appearing in the data.
- Join indexes to speed up table joins or self-joins.
- Value indexes on one or more columns.
- Structural indexes, for fast access to data based on path expressions.

Reading

Pages 152-157 in Chapter 6 of "Graph Databases", I.Robinson, J.Webber, E.Eifrem, 2015, on the topic of "Native Graph Storage" — gives a description of how data is stored in the Neo4J Graph DBMS.

Optional: "Storing and Indexing Massive RDF Data Sets", Yongming Luo et al. In: Semantic Search Over the Web, Roberto De Virgilio et al. (Eds), Springer 2012. Read to the end of Section 1.2 on page 13. Read the rest of the chapter for interest.