Querying and Exchanging XML and RDF on the Web

WWW'12 Tutorial

Sherif Sakr (NICTA and UNSW)
Axel Polleres (Siemens AG)
Tutorial Overview

Session 1

XQuery Overview – Sherif

SPARQL Overview – Axel

XSPARQL: a combined language – Axel

Session 2

XQuery implementations – Sherif

SPARQL implementations – Sherif

XSPARQL implementations – Axel

(optional) Compression formats for XML+RDF: EXI+HDT – Sherif

Q/A - Discussion
XQuery

- XQuery is a **declarative** language in which a query is represented as an **expression**.
- XQuery expressions can be nested with full generality.

```xml
for $d in distinct-values(doc("ord.xml")//item/@dept)
let $items := doc("ord.xml")//item[@dept = $d]
order by $d
return <department name="{$d}"
   totalQuantity="{sum($items/@quantity)}"/>
```
XQuery, XSLT and XPath

- FLWOR Expressions
- XML Constructors
- Query Prolog
- User-Defined Functions

XQuery 1.0

XPath 1.0

XPath 2.0

XPath 2.0

Conditional Expressions
Arithmetic Expressions
Quantified Expressions
Built-In Functions & Operators
Data Model

Stylesheets
Templates etc.

Path Expressions
Comparison Expressions
Some Built-In Functions
The XQuery language is designed to operate over ordered, *finite sequences of items* as its principal data type.

The evaluation of any XQuery expression yields an ordered sequence of $n \geq 0$ items.

These *items* can be:

- Atomic values (integers, strings, ..., etc)
- Unranked XML tree nodes.
Items and Ordered Sequences

- A sequence of *n* items $X_i$ is written in parentheses and comma-separated form

  $$(X_1, X_2, ..., X_n)$$

- A single item $X$ and the singleton sequence $(X)$ are equivalent.

- Sequences *can not* contain other sequences (nested sequences are implicitly flattened)

  $$(0, (), (1, 2), (3)) = (0, 1, 2, 3)$$
  $$(0, 1) \neq (1, 0)$$

- Sequences *can* contain *duplicates*

  $$(0, 1, 1, 2)$$

- Sequences may be *heterogeneous*

  $$(42, "foo", 4.2, <a></a>)$$
XQuery = ½ Programming Language + ½ Query Language

- **Programming language** features:
  - Explicit iteration and variable bindings (for, let, …).
  - Recursive, user-defined functions.
  - Regular expressions, strong [static] typing.
  - Ordered sequences (much like lists or arrays).

- **Query language** features:
  - Filtering.
  - Grouping.
  - Joins.
XQuery

- A strongly-typed, Turing-complete XML manipulation language
  - Attempts to do static type checking against XML Schema
  - Based on an object model derived from Schema
- Unlike SQL, fully compositional, highly orthogonal:
  - Inputs & outputs collections (sequences or bags) of XML nodes
  - Anywhere a particular type of object may be used, may use the results of a query of the same type
  - Designed mostly by DB and functional language people
- Attempts to satisfy the needs of data management and document management
  - The database-style core is mostly complete (even has support for NULLs in XML!!)
  - The document keyword querying features are still in the works – shows in the order-preserving default model
Some Uses for XQuery

- Extracting information from a database for use in web service.
- Generating summary reports on data stored in XML database.
- Searching textual documents on the web for relevant information.
- Transforming XML data to XHTML format to be published on the web.
- Pulling data from different databases to be used for application integration.
- Splitting up an XML document into multiple XML documents.
XQuery Expressions

- Path expressions.
- FLWOR expressions.
- Expressions involving operators and functions.
- Conditional expressions.
- Quantified expressions.
- List constructors.
- Element constructors.
- Expressions that test or modify datatypes.
Path Expression

- In a sense, the *traversal* or *navigation* of trees of XML nodes lies at the core of every XML query language.
- XQuery embeds **XPath** as its tree navigation sub-language.
- Every XPath expression is a correct XQuery expression.
- Since navigation expressions extract (potentially huge volumes of) nodes from input XML documents, efficient XPath implementation is a prime concern to any implementation of an XQuery processors.
Path Expression

- Each path consists of one or more \textbf{steps}, syntactically separated by /

\[ s_0 / s_1 / \ldots / s_n \]

- Each step acts like an operator that, given a \textbf{sequence of nodes} (the context set), evaluates to a \textbf{sequence of nodes}.

- XPath defines the result of each path expression to be \textit{duplicate free} and \textit{sorted in document order}. 
FLWOR Expression

- A FLWOR expression binds some expressions, applies a predicate, and constructs a new ordered result.

\[
\text{for } \$\text{for\_var} \text{ in for\_expression} \\
\text{let } \$\text{let\_var} := \text{let\_expression}
\]

Variable bindings

where clause

Filtered variable bindings

order by clause

Ordered variable bindings

return clause

Result set
FLWOR Expression

- Has an analogous form to SQL’s
  SELECT..FROM..WHERE..GROUPBY..ORDER BY
- The model: bind nodes (or node sets) to variables; operate over each legal combination of bindings; produce a set of nodes
- “FLWOR” statement:
  for {iterators that bind variables}
  let {collections}
  where {conditions}
  order by {order-conditions}
  return {output constructor}
FLWOR Expression

- The **for** construct **successively binds** each item of an expression (**expr**) to a variable (**var**), generating a so-called **tuple stream**.
- This tuple stream is then **filtered** by the **where** clause, retaining some tuples and discarding others.
- The **return** clause is evaluated **once for every tuple** still in the stream.
- The result of the expression is an **ordered sequence** containing the **concatenated** results of these evaluations.
FLWOR Expression

**Iteration**

```
for $x$ in (3,2,1)  \Rightarrow  (3,"*",2,"*",1,"*")
return ($x","*")
```

```
for $x$ in (3,2,1)  \Rightarrow  (3,2,1,"*")
return $x","*")
```

```
for $x$ in (3,2,1)  \Rightarrow  (3,"a",3,"b")
return for $y$ in ("a","b")  \Rightarrow  2,"a",2,"b")
return ($x,$y)  \Rightarrow  1,"a",1,"b")
```
Inner Joins

return

<book>
    { $book/title }
    { $quote/price }
</book>
Outer Joins

for $book in document("bib.xml")//book
return
<book>
  { $book/title }
  {
    for $review in document("reviews.xml")//review
    where $book/isbn = $review/isbn
    return $review/rating
  }
</book>
Aggregation - Grouping

- **for** iterates on a sequence, binds a variable to each node.
- **let** binds a variable to a sequence as a whole.
- Together, they are used for representing aggregation and grouping expressions.

```xml
for $book in document("bib.xml")//book
let $a := $book/author
where contains($book/publisher, "Addison-Wesley")
return
  <book>
    {
      $book/title,
    <count> Number of authors: { count($a) } </count>
    }
  </book>
```
XQuery Operators and Functions

- Infix and prefix operators (+, -, *, ...).
- Parenthesised expressions.
- Arithmetic and logical operators.
- Collection operators **UNION**, **INTERSECT** and **EXCEPT**.
- Infix operators **BEFORE** and **AFTER** (<<, >>).
- User functions can be defined in XQuery.
Node Comparisons

**Node comparison**

**Node comparisons** based on *identity* and *document order*:

- \( e_1 \text{ is } e_2 \)  
  nodes \( e_{1,2} \) identical?
- \( e_1 << e_2 \)  
  node \( e_1 \) before \( e_2 \)?
- \( e_1 >> e_2 \)  
  node \( e_1 \) after \( e_2 \)?

---

**Node comparison examples**

\[
\begin{align*}
\text{eq} & :\quad \langle x \rangle 42 \langle /x \rangle \quad \Rightarrow \quad \text{true}() \\
\text{is} & :\quad \langle x \rangle 42 \langle /x \rangle \quad \Rightarrow \quad \text{false}() \\
\text{root} & :\quad \text{root}(e_1) \quad \Rightarrow \quad \text{nodes } e_{1,2} \text{ in same tree?} \\
\text{let} & :\quad \text{let } a := \langle x \rangle \langle y / \rangle \langle /x \rangle \\
\text{return} & :\quad \text{return } a << a/y \quad \Rightarrow \quad \text{true}()
\end{align*}
\]
# XQuery Comparisons

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
<td>comparing single values</td>
<td>eq, ne, lt, le, gt, ge</td>
</tr>
<tr>
<td></td>
<td>Untyped data =&gt; string</td>
<td></td>
</tr>
<tr>
<td><strong>General</strong></td>
<td>Existential quantification</td>
<td>=, !=, &lt;=, &lt;, &gt;, &gt;=</td>
</tr>
<tr>
<td></td>
<td>Untyped data =&gt; coerced to other operand’s type</td>
<td></td>
</tr>
<tr>
<td><strong>Node</strong></td>
<td>for testing identity of single nodes</td>
<td>is, isnot</td>
</tr>
<tr>
<td><strong>Order</strong></td>
<td>testing relative position of one node vs. another (in document order)</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
</tbody>
</table>
Logical Expression

- Logical Operators “and” and “or”.
- The concept of Effective Boolean Value (EBV) is key to evaluating logical expressions.
  - EBV of an empty sequence is false.
  - EBV of a non-empty sequence containing only nodes is true.
  - EBV is the value of the expression if the expression evaluates to a value of type xs:boolean.
  - EBV is an error in every other case.
- e.g: The expression “() and true()” evaluates to false (since () is false)
XQuery: Built-in Functions

- Over 100 functions built into XQuery.
- **String-related**
  - substring, contains, concat,…
- **Date-related**
  - current-date, month-from-date,…
- **Number-related**
  - round, avg, sum, …
- **Sequence-related**
  - index-of, distinct-values,…
- **Node-related**
  - data, empty,…
- **Document-related**
  - doc, collection, …
- **Error Handling**
  - error, exactly-one, …
- ….
XQuery expressions can contain user-defined functions which encapsulate query details. User-defined functions may be collected into modules and then ‘import’ed by a query.

### Declaration of n-ary function $f$ with body $e$

```
declare function $f$($p_1$ as $t_1$, ..., $p_n$ as $t_n$) as $t_0$ { e }
```

- If $t_i$ is omitted, it defaults to `item()`*.
- The pair ($f$, $n$) is required to be unique (overloading).
- Atomization is applied to the $i$-th parameter if $t_i$ is atomic.
User-Defined Functions Example

Reverse a sequence

Reversing a sequence does not inspect the sequence’s items in any way:

```plaintext
declare function reverse($seq)
{
    for $i at $p in $seq
        order by $p descending
        return $i
}
reverse((42,"a",<b/>,doc("foo.xml")))
```
Conditional Expression

- **Syntax:**
  
  ```
  if (expr1) then expr2 else expr3
  ```

  - if **EBV** of `expr1` is true, the conditional expression evaluates to the value of `expr2`, else it evaluates to the value of `expr3`.
  - Parentheses around **if** expression (`expr1`) are required.
  - **else** is always required but it can be just `else ()`.
  - Useful when structure of information returned depends on a condition.
  - Can be nested and used anywhere a value is expected.

  ```
  if($book/@year <1980 )
  then<old>{$x/title}</old>
  else<new>{$x/title}</new>
  ```
Conditional Expression

- Used as an alternative way of writing the FLWOR expressions.

**FLWOR:**
```
for $a in document("bib.xml")//article
  where $a/year < 1996
  return $a
```

**Conditional:**
```
for $a in document("bib.xml")//article
  return
  If ($a/year < 1996)
  then$a
  else()
```
Quantified Expressions

- Syntax:

  \[ \text{[some | every]} \var \text{ in } expr \text{ satisfies } test\_expr \]

- Quantified expressions evaluate to a boolean value.

- Evaluation:
  - \$var\$ is bound to each of the items in the sequence resulting from \$expr\$.
  - For each binding, the \$test\_expr\$ is evaluated.
  - In case of
    - Existential quantification ("some"), if at least one evaluation of \$test\_expr\$ evaluates \"true\", the entire expression evaluates \"true\".
    - Universal quantification ("every"), all evaluations of \$test\_exp\$ must result in an EBV of \"true\" for the expression to return \"true\".
Quantified Expressions

- Existential Quantification
  - Give me all books where “Sailing” appear at least once in the same paragraph.

```xml
for $b in document("bib.xml")//book
where some $p in $b//para satisfies(contains($p,"Sailing"))
return $b/title
```
Universal Quantification

Give me all books where “Sailing” appears in every paragraph.

```xml
for $b in document("bib.xml")//book
where every $p in $b//para satisfies(contains($p,"Sailing"))
return $b/title
```
XQuery List Constructors

- A list may be constructed by enclosing zero or more expressions in square brackets, separated by commas.
- For example, \([x, y, z]\) denotes a list containing three members represented by variables.
- \([\ ]\) denotes an empty list.
XML tree fragments may be constructed “on the fly” in queries:

```xquery
for $i in doc("auction.xml") /site/regions/australia/item
return
  <item>
    <name>{$i/name/text()}</name>
    <description>{$i/description}</description>
  </item>
```

XQuery Element Constructors
Library Modules

- Separate XQuery documents that contain function definitions.

Why?
- Reusing functions among many queries.
- Defining standard libraries that can be distributed to a variety of query users.
- Organizing and reducing the size of query modules.
**Library Modules**

**Main module**

```xml
import module
    namespace strings = "http://datypic.com/strings"
at "http://datypic.com/strings/lib.xq"

<results>strings:trim(doc("cat.xml")//name[1])</results>
```

**Library module**

```xml
module namespace strings = "http://datypic.com/strings";
declare function strings:trim($arg as xs:string?)
as xs:string? {
    (: ...function body here... :) 
}
```
Global Variables

- Declared and bound in the query prolog and used through the query.
- Can be
  - Referenced in a function that is declared in that module.
  - Referenced in other modules that import the module.

```lisp
declare variable $maxNumItems := 3;
declare variable $ordDoc := doc("ord.xml");

for $item in $ordDoc//item[position() <= $maxNumItems]
  return $item
```
Advantages of XQuery

- It is easy to learn if knowledge of SQL and XPath is present.
- When queries are written in XQuery, they require less code than queries written in XSLT do.
- XQuery can be used as a strongly typed language when the XML data is typed, which can improve the performance of the query by avoiding implicit type casts and provide type assurances that can be used when performing query optimization.
- XQuery can be used as a weakly typed language for untyped data to provide high usability.
- Because XQuery requires less code to perform a query than does XSLT, maintenance costs are lower.
- XQuery is supported by major database vendors.
Capabilities of XQuery

- Selecting information based on specific criteria
- Filtering out unwanted information
- Searching for information within a document or set of documents
- Joining data from multiple documents or collections of documents
- Sorting, grouping, and aggregating data
- Transforming and restructuring XML data into another XML vocabulary or structure
- Performing arithmetic calculations on numbers and dates
- Manipulating strings to reformat text
Uses of XQuery

- Extracting information from a relational database for use in a web service
- Generating reports on data stored in a database for presentation on the Web as XHTML
- Searching textual documents in a native XML database and presenting the results
- Pulling data from databases or packaged software and transforming it for application integration
- Combining content from traditionally non-XML sources to implement content management and delivery
- Ad hoc querying of standalone XML documents for the purposes of testing or research
Tutorial Overview

Session 1
- XQuery Overview – Sherif
- SPARQL Overview – Axel
- XSPARQL: a combined language – Axel

Session 2
- XQuery implementations – Sherif
- SPARQL implementations – Sherif
- XSPARQL implementations – Axel
  (optional) Compression formats for XML+RDF: EXI+HDT – Sherif

Q/A - Discussion
XQuery Processors

- **Native XML/XQuery Processors:** The implementations of this approach make use of storage models, indexing and querying mechanisms that have been designed specifically for XML data.

- **Streaming XQuery Processors:** The implementations of this approach receive the XML data in the form of continuous streams of tokens and apply on the fly the query processing functionalities over them.

- **Relational XQuery Processors:** The implementations of this approach makes use of the relational indexing and querying mechanisms for querying the source XML data.
Timber

• A native XML database which is able to store and query XML documents.

• It uses the tree-based query algebra (TAX) which considers collections of ordered labelled trees as the basic unit of manipulation

• Each operator takes one or more sets of trees as input and produces a set of trees as output.

• The query plan tree is evaluated by pipelining one operator at a time.

• The query execution heavily depends on structural joins.
Natix

- Another native XML database which clusters subtrees of XML documents into physical records of limited size.
- The XML data tree is partitioned into small subtrees and each subtree is stored into a data page.
- The query execution engine consists of an iterator-based implementation of algebraic operators which process ordered sequences of tuples.

DB2/System RX

- A native XML support and XQuery implementation in IBM DB2.
- It uses a new XML data type which can be used like any other SQL type.
- A column of type XML can hold one well-formed XML document for every row of the table.
- Relational and XML data are stored differently while the relational columns are stored in traditional row structures, the XML data is stored in hierarchical structures.
- It uses Structural Indexes, Value Indexes and Full-text Indexes.
- Supports interfaces for both SQL/XML and XQuery.

DB2/System RX
create table dept (deptID char(8),..., deptdoc xml);
Tag names and namespace URIs get replaced by unique StringIDs.
DB2/System RX

Document tree delivered by the XML parser gets broken into regions.

Regions index

page

page

page
FluXQuery

- A streaming XQuery processor that is based on an internal query language called FluX which extends the main structures of XQuery by introducing a construct for event-based query processing.
- The buffer size is then optimized by analyzing the schema constraints of the XML document as well as the query syntax.
- FluX queries are transformed into physical query plans which are translated into executable JAVA code or interpreted and executed using the Streamed Query Evaluator.

FluXQuery

Transformation into Normal Form

XQuery into Normal Form

Algebraic Optimization

Query Optimizer

Translation into FluX

FluX Query

Query Compiler

Physical Query Plan

Memory Buffers

Streamed Query Evaluator

Output Stream (XML)

Input Stream (XML)

XSAX

Runtime Engine
SQL Server

- Introduces native storage for XML data as a new, rich data type called XML. A table may contain one or more columns of type XML wherein both rooted XML trees and XML fragments can be stored.
- The hierarchical nature of the XML data is modelled as parent-child relationship using a node labelling scheme.
- XQuery expressions are compiled into a query trees that can be optimized and executed by the relational query processor.
- It uses a set of relational operators which is extended with additional operators for XML processing named as relational+ operators.
- The XML operator mapper recursively traverses the XML algebra tree and for each operator a relational operator sub-tree is generated and inserted into the overall relational operator tree for the XQuery expression.

Shankar Pal et al. XQuery Implementation in a Relational Database System. VLDB, 2005.
SQL Server

XQuery expression

XQuery Compiler

XML algebra tree (XmlOp operators)

XML Operator Mapper

Relational Operator Tree
(relational+ operators)

Relational Query Processor
Pathfinder

- A purely relational XQuery processor

http://pathfinder-xquery.org/
Pathfinder

Relational XML Encoding

Torsten Grust: Accelerating XPath location steps. SIGMOD, 2002
Pathfinder

Relational Algebra

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi$</td>
<td>column projection, renaming</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>row selection</td>
</tr>
<tr>
<td>$\cup$, $\setminus$</td>
<td>disjoint union, difference</td>
</tr>
<tr>
<td>$\delta$</td>
<td>duplicate elimination</td>
</tr>
<tr>
<td>$\bowtie$</td>
<td>equi-join</td>
</tr>
<tr>
<td>$\times$</td>
<td>Cartesian product</td>
</tr>
<tr>
<td>$\rho$</td>
<td>row numbering</td>
</tr>
<tr>
<td>$\sqcap$</td>
<td>staircase join</td>
</tr>
<tr>
<td>$\varepsilon$, $\tau$</td>
<td>element/text node construction</td>
</tr>
<tr>
<td>$\otimes$</td>
<td>arithmetic/comparison/Boolean operator *</td>
</tr>
</tbody>
</table>

- **No tree pattern matching** or similar operators involved here
- This algebra is efficiently implementable on (top of) SQL hosts
Pathfinder

Relational XQuery Compilation

let $auction := doc("auction.xml")
return for $b in $auction/site/regions
    return count($b/item)
Other Projects

- Galax
  - [http://www.galaxquery.org/](http://www.galaxquery.org/)
- MXQuery
  - [http://mxquery.org/](http://mxquery.org/)
- Zorba
- BaseX
  - [http://basex.org/](http://basex.org/)
- Xyleme
  - [http://www.xyleme.com/](http://www.xyleme.com/)
- eXist
- Saxon
- MarkLogic
XML Database Benchmarks

- XBench
- XMach
- XMark
- XOO7
- TPoX
- XPathMark
  - [http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/](http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/)
- MemBeR
  - [http://ilps.science.uva.nl/Resources/MemBeR/member-generator.html](http://ilps.science.uva.nl/Resources/MemBeR/member-generator.html)
- XSelMark
Tutorial Overview

Session 1

XQuery Overview – Sherif
SPARQL Overview – Axel
XSPARQL: a combined language – Axel
Compression formats for XML+RDF: EXI+HDT – Sherif

Session 2

XQuery implementations – Sherif
SPARQL implementations – Sherif
XSPARQL implementations – Axel
(optional) Compression formats for XML+RDF: EXI+HDT – Sherif

Q/A - Discussion
Two Alternatives to Store RDF Data

- **Relational RDF Stores**
  - Takes advantage of 30+ years of R&D
  - A schema design exercise?
  - Requires a SPARQL-to-SQL layer
  - Relative flexibility in choosing the actual back-end

- **Native RDF Stores**
  - Takes advantage of 30+ years of R&D
  - Highly tuned for RDF data
Relational RDF Stores

- **Vertical (triple) table stores**: where each RDF triple is stored directly in a *three-column* table (subject, predicate, object).

- **Property (n-ary) table stores**: where multiple RDF properties are modeled as *n-ary* table columns for the same subject.

- **Horizontal (binary) table stores**: where RDF triples are modeled as one horizontal table or into a set of vertically partitioned binary tables (one table for each RDF property).

Sherif Sakr, Ghazi Al-Naymat: *Relational processing of RDF queries: a survey*. SIGMOD Record 38(4), 2009
SELECT ?Z
WHERE { ?X hasTitle "Querying RDF Data" . ?X publicationType "Survey Paper" .
  ?X authoredBy ?Y .
  ?Y webPage ?Z .}
Vertical (triple) table stores

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id1</td>
<td>publicationType</td>
<td>Survey Paper</td>
</tr>
<tr>
<td>Id1</td>
<td>hasTitle</td>
<td>Querying RDF Data</td>
</tr>
<tr>
<td>Id1</td>
<td>authoredBy</td>
<td>Id2</td>
</tr>
<tr>
<td>Id2</td>
<td>hasName</td>
<td>John</td>
</tr>
<tr>
<td>Id2</td>
<td>affiliatedBy</td>
<td>UNSW</td>
</tr>
<tr>
<td>Id2</td>
<td>hasEmail</td>
<td><a href="mailto:John@cse.unsw.edu.au">John@cse.unsw.edu.au</a></td>
</tr>
<tr>
<td>Id2</td>
<td>webpage</td>
<td><a href="http://www.cse.unsw.edu.au/~john">www.cse.unsw.edu.au/~john</a></td>
</tr>
<tr>
<td>Id1</td>
<td>editedBy</td>
<td>Id3</td>
</tr>
<tr>
<td>Id3</td>
<td>hasName</td>
<td>Alice</td>
</tr>
<tr>
<td>Id3</td>
<td>affiliatedBy</td>
<td>NICTA</td>
</tr>
<tr>
<td>Id3</td>
<td>hasEmail</td>
<td><a href="mailto:Alice@nicta.com.au">Alice@nicta.com.au</a></td>
</tr>
<tr>
<td>Id3</td>
<td>roomNo</td>
<td>518</td>
</tr>
</tbody>
</table>

Select T3.Object
From Triples as T1, Triples as T2, Triples as T3, Triples as T4
Where
T1.Predicate="publicationType" and T1.Object="Survey Paper"
and T2.predicate="hasTitle"
and T2.Object="Querying RDF Data"
and T3.Predicate="webPage"
and T1.subject=T2.subject
and T4.subject=T1.subject
and T4.Predicate="authoredBy"
and T4.Object = T3.Subject

No schema changes as new triples are added to the store
SPARQL queries result in equivalent SQL queries involving multiple self-joins
Property Table Stores

Publication

<table>
<thead>
<tr>
<th>ID</th>
<th>publicationType</th>
<th>hasTitle</th>
<th>authoredBy</th>
<th>editedBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id1</td>
<td>Survey Paper</td>
<td>Querying RDF Data</td>
<td>Id2</td>
<td>id3</td>
</tr>
</tbody>
</table>

Person

<table>
<thead>
<tr>
<th>ID</th>
<th>hasName</th>
<th>affiliatedBy</th>
<th>hasEmail</th>
<th>webpage</th>
<th>roomNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id2</td>
<td>John</td>
<td>UNSW</td>
<td><a href="mailto:John@cse.unsw.edu.au">John@cse.unsw.edu.au</a></td>
<td><a href="http://www.cse.unsw.edu.au/~john">www.cse.unsw.edu.au/~john</a></td>
<td></td>
</tr>
<tr>
<td>Id3</td>
<td>Alice</td>
<td>NICTA</td>
<td><a href="mailto:Alice@nicta.com.au">Alice@nicta.com.au</a></td>
<td></td>
<td>518</td>
</tr>
</tbody>
</table>

Select Person.webPage
From Person, Publication
Where Publication.publicationType = "Survey Paper"
and Publication.hasTitle = "Querying RDF Data"
and Publication.authoredBy = Person.ID

Implicit or explicit knowledge is required to build the property tables
Horizontal (Binary) Table Stores

<table>
<thead>
<tr>
<th>publicationType</th>
<th>hasTitle</th>
<th>webPage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id1</td>
<td>Id1</td>
<td>Id2</td>
</tr>
<tr>
<td>Survey Paper</td>
<td>Querying RDF Data</td>
<td><a href="http://www.cse.unsw.edu.au/~john">www.cse.unsw.edu.au/~john</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasName</th>
<th>affiliatedBy</th>
<th>roomNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id2</td>
<td>Id2</td>
<td>Id3</td>
</tr>
<tr>
<td>John</td>
<td>UNSW</td>
<td>518</td>
</tr>
<tr>
<td>Id3</td>
<td>Id3</td>
<td></td>
</tr>
<tr>
<td>Alice</td>
<td>NICTA</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasEmail</th>
<th>authoredBy</th>
<th>editedBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id2</td>
<td>Id2</td>
<td>Id1</td>
</tr>
<tr>
<td><a href="mailto:John@cse.unsw.edu.au">John@cse.unsw.edu.au</a></td>
<td>Id3</td>
<td>Id1</td>
</tr>
<tr>
<td>Id3</td>
<td><a href="mailto:Alice@nicta.com.au">Alice@nicta.com.au</a></td>
<td></td>
</tr>
</tbody>
</table>

Select `webPage.value`  
From `publicationType`, `hasTitle`, `authoredBy`, `webPage`  
Where `publicationType.value = "Survey Paper"`  
and `hasTitle.value = "Querying RDF Data"`  
and `publicationType.ID = hasTitle.ID`  
and `publicationType.ID = authoredBy.ID`  
and `authoredBy.value = webPage.ID`

Does this work for Dbpedia which has approx. 65K properties?
RDF-3X

• An RDF query engine which tries to overcome the criticism that triples stores incurs too many expensive self-joins by creating the exhaustive set of indexes and relying on fast processing of merge joins.

• The physical design of RDF-3x is workload-independent and eliminates the need for physical-design tuning by building indexes over all 6 permutations of the three dimensions that constitute an RDF triple.

• Additionally, indexes over count-aggregated variants for all three two-dimensional and all three one-dimensional projections are created.

Thomas Neumann, Gerhard Weikum: The RDF-3X engine for scalable management of RDF data. VLDB J. 19 (1), 2010
RDF-3X

• The query processor uses the full set of indexes on the triple tables to rely mostly on *merge joins* over sorted index lists.

• The query optimizer relies upon its cost model in finding the lowest-cost execution plan and mostly focuses on join order and the generation of execution plans.

• Additionally, indexes over count-aggregated variants for all three two-dimensional and all three one-dimensional projections are created.

• It relies on two kinds of statistics:
  – *Specialized histograms* which are generic and can handle any kind of triple patterns and joins. The disadvantage of histograms is that it assumes independence between predicates.
  – *Frequent join paths* in the data which give more accurate estimation.
Hexastore

• RDF storage scheme with main focuses on scalability and generality in its data storage, processing and representation.

• Hexastore is based on the idea of indexing the RDF data in a multiple indexing scheme.

• Each RDF element type have its special index structures built around it.

• Every possible ordering of the importance or precedence of the three elements in an indexing scheme is materialized.

• In total, six distinct indices are used for indexing the RDF data. These indices materialize all possible orders of precedence of the three RDF elements.

• A clear disadvantage of this approach is that it features a worst-case five-fold storage increase in comparison to a conventional triples table.
Hexastore

An Example SPO Index of Hexastore
Oracle RDF

- Oracle introduced an Oracle-based SQL table function *RDFMATCH* to query RDF data.
- The results of *RDFMATCH* table function can be further processed by SQL’s rich querying capabilities and seamlessly combined with queries on traditional relational data.
- The core implementation of *RDFMATCH* query translates to a self-join query on triple-based RDF table store.
- The resulting query is executed efficiently by making use of B-tree indexes as well as creating *materialized join views* for specialized subject-property.

*Eugene Inseok Chong, Souripriya Das, George Eadon, and Jagannathan Srinivasan:* *An Efficient SQL-based RDF Querying Scheme.* VLDB, 2005
Oracle RDF

- Subject-Property Matrix materialized join views are used to minimize the query processing overheads that are inherent in the canonical triple-based representation of RDF.

- The materialized join views are *incrementally* maintained based on *user demand* and *query workloads*.

- A special module is provided to analyze the table of RDF triples and estimate the size of various materialized views, based on which a user can define a subset of materialized views.
SELECT t.r reviewer, t.c conf, t.a age
FROM TABLE(RDF_MATCH(
   `(?r rdf:type Student)
   (?r ReviewerOf ?c)
   (?r Age ?a)`,
RDFModels (`reviewers'),
NULL, NULL)) t
WHERE t.a < 25;
Workload Independent Property Tables

• The approach provides a tailored schema for each RDF data based on two main parameters:
  – The Support threshold which represents a value to measure the strength of correlation between properties in the RDF data.
  – The null threshold which represents the percentage of null storage tolerated for each table in the schema.

• The approach involves two phases
  – The clustering phase scans the RDF data to automatically discover groups of related properties (i.e., properties that always exist together for a large number of subjects).
  – The partitioning phase goes over the formed clusters and balances the trade-off between storing as many RDF properties in clusters as possible while keeping null storage to a minimum based on the null threshold.

Justin J. Levandoski and Mohamed F. Mokbel: RDF Data-Centric Storage. ICWS, 2009
Path-Based RDF Stores

- The main focus of this approach is to improve the performance for path queries by extracting all reachable path expressions for each resource and store them.
- There is no need to perform join operations unlike the flat triple stores or the property tables approach.
- The RDF graph is divided into subgraphs and then each subgraph is stored by applicable techniques into distinct relational tables. More precisely, all classes and properties are extracted from RDF schema data, and all resources are also extracted from RDF data.
- Each extracted item is assigned an identifier and a path expression and stored in corresponding relational table.
SW-Store

- The triples table is rewritten into $n$ two-column tables where $n$ is the number of unique properties in the data.
- In each binary table, the first column contains the subjects that define that property and the second column contains the object values for those subjects while the subjects that do not define a particular property are simply omitted from the table for that property.
- Each table is sorted by subject, so that particular subjects can be located quickly, and that fast merge joins can be used to reconstruct information about multiple properties for subsets of subjects.

Daniel J. Abadi, Adam Marcus, Samuel Madden, and Kate Hollenbach: SW-Store: a vertically partitioned DBMS for Semantic Web data management. VLDB J. 18(2), 2009
SW-Store

• For a *multi-valued* attribute, each distinct value is listed in a successive row in the table for that property.

• The implementation of SW-Store relies on a column-oriented DBMS, **C-store**, to store tables as collections of columns rather than as collections of rows.

• The advantage of this approach is that while property tables need to be carefully constructed so that they are wide enough but not too wide to independently answer queries, the algorithm for creating tables in the vertically partitioned approach is straightforward and need not change over time.

• The main disadvantages of this approach are: the increased cost of inserts new triples and the cost of tuple reconstruction
SW-Store

SQL/SPARQL Queries

Query Rewriter

Query Executor

RDF Inserts

RDF Batch Writer

Vertically partitioned storage system

Chunks of Relational Tuples
AdaptRDF

- A two-phase approach for designing efficient tailored but flexible storage solution for RDF data based on its query workload.
- A *workload-aware vertical partitioning* phase.
  - Reduces the number of join operations in the query evaluation process
- An *automated adjustment* phase
  - Maintains the efficiency of the performance of the query processing by adapting the underlying schema to cope with the dynamic nature of the query workloads
  - Rows pivoting and uninviting

Hooran MahmoudiNasab and Sherif Sakr: Efficient and Adaptable Query Workload-Aware Management for RDF Data. WISE, 2010
AdaptRDF

Initial query workload \hspace{1cm} RDF Dataset

\begin{align*}
\text{Vertical Partitioning Algorithm} \\
\text{(VP)}
\end{align*}

Initial Schema \\
(Triple Store + Property Tables)

\begin{align*}
\text{Monitoring Module} \\
\text{(Re-apply VP)}
\end{align*}

Query workload stream

Adjusted Schema

\begin{align*}
\text{Adjusting Module} \\
\text{(Pivoting/Unpivotting)}
\end{align*}
<table>
<thead>
<tr>
<th></th>
<th>Upper</th>
<th></th>
<th>Lower</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_6$</td>
<td>$a_2$</td>
<td>$a_3$</td>
</tr>
<tr>
<td>$a_6$</td>
<td>90</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>$a_2$</td>
<td>30</td>
<td>105</td>
<td>60</td>
</tr>
<tr>
<td>$a_3$</td>
<td>25</td>
<td>60</td>
<td>40</td>
</tr>
<tr>
<td>$a_7$</td>
<td>25</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>$a_{10}$</td>
<td>15</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>$a_8$</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>$a_1$</td>
<td>10</td>
<td>15</td>
<td>10</td>
</tr>
<tr>
<td>$a_9$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$a_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$a_5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
AdaptRDF

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id1</td>
<td>publicationType</td>
<td>Survey Paper</td>
</tr>
<tr>
<td>Id1</td>
<td>hasTitle</td>
<td>Querying RDF Data</td>
</tr>
<tr>
<td>Id1</td>
<td>authoredBy</td>
<td>Id2</td>
</tr>
<tr>
<td>Id2</td>
<td>hasEmail</td>
<td><a href="mailto:John@cse.unsw.edu.au">John@cse.unsw.edu.au</a></td>
</tr>
<tr>
<td>Id2</td>
<td>webpage</td>
<td><a href="http://www.cse.unsw.edu.au/~john">www.cse.unsw.edu.au/~john</a></td>
</tr>
<tr>
<td>Id1</td>
<td>editedBy</td>
<td>Id3</td>
</tr>
<tr>
<td>Id3</td>
<td>hasEmail</td>
<td><a href="mailto:Alice@nicta.com.au">Alice@nicta.com.au</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>affiliatedBy</th>
<th>roomNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id2</td>
<td>John</td>
<td>UNSW</td>
<td></td>
</tr>
<tr>
<td>Id3</td>
<td>Alice</td>
<td>NICTA</td>
<td>518</td>
</tr>
</tbody>
</table>

(a) Pivot (hasName, affiliatedBy, roomNo)

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasName</th>
<th>affiliatedBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id2</td>
<td>John</td>
<td>UNSW</td>
</tr>
<tr>
<td>Id3</td>
<td>Alice</td>
<td>NICTA</td>
</tr>
</tbody>
</table>

(b) Unpivot (roomNo)
RDF Benchmarks

• Lehigh University Benchmark (LUBM)
  – http://swat.cse.lehigh.edu/projects/lubm/
  – (University domain)

• Berlin SPARQL Benchmark (BSBM)
  – http://www4.wiwiss.fu-berlin.de/bizer/berlinsparqlbenchmark/
  – E-Commerce domain

• The SP²Bench SPARQL Performance Benchmark
  – http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B
  – DBLP Scenario

• The DBpedia SPARQL Benchmark
  – http://aksw.org/Projects/DBPSB
  – Wikipedia + YAGO
Benchmarkings (1)

• The benchmark implements two relational RDF storage solutions – *triple-store* and *vertically-partitioned* – in a *MonetDB/SQL*, a fully-functional open source column-store, and a well-known – for its performance – commercial row-store DBMS.

• The vertically-partitioned approach does not outperform triple-store when both are implemented in a row-store engine.

• The vertically-partitioned approach outperforms triple-store when both are implemented in a column-store. The processing efficiency of column-stores is particularly suited for RDF data management applications.

*Lefteris Sidirourgos, Romulo Goncalves, Martin L. Kersten, Niels Nes, and Stefan Manegold. Column-store support for RDF data management: not all swans are white. PVLDB 1(2), 2008*
Benchmarkings (1)

- Once the proper clustered indices are used in a row-store engine, the triple-store performs better than the vertically-partitioned approach.

- There is a potential of a scalability problems for the vertically-partitioned approach when the number of properties in an RDF dataset is high. With a larger number of properties, the triple-store solution manages to outperform the vertically-partitioned approach also on a column-store engine.
The benchmark evaluates the approaches of relational processing for RDF Queries: triple stores (TS), binary table stores (BS), property table stores (PS) and traditional relational stores (RS).

The experimental evaluation is done on top of the SP²Bench benchmark and compares the following metrics: loading time, storage costs and query performance.

The RS scheme is the fastest due to the less required number of insert tuple operations. Similarly, the TS requires less loading time than BS since the number of inserted tuples and updated tables are smaller for each triple.
Benchmarkings (2)

- The **RS** scheme represents the cheapest approach because of the normalized design and the absence of any data redundancy. The **BS** scheme represents the most expensive approach due to the redundancy of the ID attributes for each binary table.

- There is no clear winner between the triple store (TS) and the binary table (BS) encoding schemes. Triple store (TS) with its simple storage and the huge number of tuples in the encoding relation is still very competitive to the binary tables encoding scheme because of the full set of B-tree physical indexes over the permutations of the three encoding fields (subject; predicate; object).
The query performance of the (BS) encoding scheme is affected badly by the increase of the number of the predicates in the input query. It is also affected by the subject-object or object-object type of joins where no index information is available for utilization. Such problem could be solved by building materialized views over the columns of the most frequently referenced pairs of attributes.

Although their generality, there is still a clear gap between the query performance of the (TS) and (BS) encoding schemes in comparison with the tailored relational encoding scheme (RS) of the RDF data.
Matrix "Bit" loaded

- A compressed *bit-matrix* structure for storing huge RDF graphs.
- The RDF data is represented by a 3D bit-cube, where each dimension of the *bitcube* represents subjects, predicates, and objects.
- The volume of this *bitcube* is $V_s \times V_p \times V_o$.
- Each cell in the *bitcube* represents a unique RDF triple that can be formed by the combination of S, P, O positions which are the coordinates of that bit.
- For each matrix, it stores two *bitarrays* – row and column bitarray – which give a condensed representation of all the non-empty row and column values in the given BitMat.
- It employs an initial *pruning* technique, followed by a *variable-binding-matching* algorithm on *BitMats* to produce the final results.
- It does not build intermediate join tables and works directly on the compressed data.

Medha Atre, Vineet Chaoii, Mohammed J. Zaki, James A. Hendler: Matrix "Bit" loaded: a scalable lightweight join query processor for RDF data. WWW, 2010
Matrix "Bit" loaded

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>the matrix</td>
<td>releasedIn</td>
<td>&quot;1999&quot;</td>
</tr>
<tr>
<td>the_matrix</td>
<td>releasedIn</td>
<td>&quot;1999&quot;</td>
</tr>
<tr>
<td>the_matrix</td>
<td>similar to</td>
<td>the_matrix</td>
</tr>
<tr>
<td>the_matrix</td>
<td>similar to</td>
<td>the_matrix</td>
</tr>
<tr>
<td>the_matrix</td>
<td>rdf:type</td>
<td>:movie</td>
</tr>
<tr>
<td>the_matrix</td>
<td>rdf:type</td>
<td>:movie</td>
</tr>
</tbody>
</table>

S-O and O-S BitMats for Ps

Note: a = the_matrix, b = "1999", c = :movie, d = the_matrix

S-O Transpose

O-S
Jena

- Developed by the Hewlett Packard Semantic Web Lab.
- Jena is a Semantic Web application framework for Java.
- It provides a programming environment for semantic technologies such as RDF(S) and OWL that supports SPARQL queries and a rule-based inference engine.
- Its engine is a combination of in-memory and native.
- A native, on-disk implementation of Jena is the *Jena TDB triple store*.

[http://incubator.apache.org/jena/]
Jena

- Triples are stored in the form of a *dictionary*. Every node is given a unique ID and instead of storing the whole node, only their ID is put in the triple store.

- Jena supports different RDF syntaxes: RDF/XML, N3, N-Triples, Turtle.

- Jena SDB version can use PostgreSQL, MySQL, Oracle or MS SQL Server as its backend.
AllegroGraph

- AllegroGraph is a persistent graph database developed by Franz Inc. that uses in-memory utilization in combination with native engine in order to maintain billions of quads.
- AllegroGraph supports named graphs. It has an optimised indexing system consisting of different combinations of S, P, O, G and I.
- It has a mechanism of automatic index optimisation that optimises indices based on how much they are used.
- Querying can be done through SPARQL 1.1 queries, with support for full-text, geospatial and geotemporal search.
- It is also compatible with a lot of other Franz technologies such as RacerPro, Pepito, TopBraid Composer, etc. to improve performance and usability.

http://www.franz.com/agraph/allegrograph/
AllegroGraph

• In June 2011, AllegroGraph announced at Semtech conference a load and query of 310 Billion triples as part of a joint project with Intel.

• In August 2011, AllegroGraph announced that with the help of Stillwater SC and Intel, they achieved the industry's first load and query of 1 Trillion RDF Triples. Total load was 1,009,690,381,946 triples in just over 338 hours for an average rate of 829,556 triples per second.

• It is in late-stage development on a clustered version that will push storage into trillions of triples.

http://www.franz.com/agraph/allegrograph/
Virtuoso

- Virtuoso is an object-relational SQL database that supports SPARQL embedded into SQL for querying RDF data in the Virtuoso database.
- Querying can be done by using SPARQL queries. There is however no SPARQL endpoint, only an SQL command line interface through which you can send SPARQL 1.1 queries to the Virtuoso repository.
- There is also a possibility to send the queries using ODBC, JDBC, ADO.NET or OLE DB.
- Full-text search and geospatial search are also supported.
- The reasoner that is implemented in the Virtuoso store is a backward chaining OWL reasoner, but reasoning can be added through the Jena framework.

http://www.openlinksw.com/dataspace/dav/wiki/Main/VOSRDF
Mulgara

- Mulgara was formerly known as Kowari.
- An Open Source, massively scalable, transaction-safe, purpose-built database for the storage, retrieval and analysis of metadata.
- It supports RDF and OWL.
- It is optimised for large amounts of triples, for metadata in particular, but it keeps low memory requirements.
- Triples in the Mulgara store are stored natively (on-disk), both locally or distributed.
- Supports SPARQL in addition to a language called TQL where Full-text search functionality is also available.
- Mulgara supports: RDF/XML, N-Triples, Turtle.
- Mulgara has a reasoner that supports: DAML+OIL, RDF(S) and OWL.

http://www.mulgara.org/
Other Projects

- Sesame
  - [http://www.openrdf.org/](http://www.openrdf.org/)
- 3Store
- 4store
  - [http://4store.org/](http://4store.org/)
- BigOWLIM
- Bigdata
  - [http://www.systap.com/bigdata.htm](http://www.systap.com/bigdata.htm)
- BrightstarDB
  - [http://www.brightstardb.com/](http://www.brightstardb.com/)
- Apache Fuseki
Other Projects

• Parliament
  – http://parliament.semwebcentral.org/

• OpenAnzo
  – http://www.openanzo.org/

• StrixDB
  – http://www.strixdb.com/

• Redstore
  – http://www.aelius.com/njh/redstore/

• Meronymy
  – http://www.meronymy.com/

• Neo4j
  – http://neo4j.org/

• rdfstore-js
  – https://github.com/antoniogarrote/rdfstore-js
The benchmark compared between: Jena TDB, AllegroGraph, OWLIM-SE, Sesame and Mulgara.

The experiments have been conducting using the LUMB, BSBM and SP²Bench benchmarks.

The comparison metrics are: Loading time, Query time and ease of use.

The experiments reported that the Mulgara was the easiest to install, run and use.

OWLIM has the best performance for loading time and query time followed by Mulgara then Jena.

AllegroGraph is the most robust tested system.
Distributed Processing of RDF/SPARQL

• A horizontally scalable RDF database system.

• It installs a best-of-breed RDF-store on a cluster of machines (they use RDF-3X since they found this to be the fastest single-node RDF-store in their benchmarking).

• It partitions an RDF data set across the data stores. Instead of randomly assigning triples to partitions using hash partitioning, they take advantage of the fact that RDF uses a graph data model, they use an optimized graph partitioning algorithm.

• Triples that are close to each other in the RDF graph to be stored on the same machine and a smaller amount of network communication at query time is required.

Jiewen Huang, Daniel J. Abadi and Kun Ren: Scalable SPARQL Querying of Large RDF Graphs. PVLDB 4 (1), 2011
Distributed Processing of RDF/SPARQL

- In order to maximize the percentage of query processing that can be done in parallel, they allow some overlap of data across partitions.

- They use a method for automatic decomposition mechanism of queries into chunks that can be performed independently, with zero communication across partitions. These chunks are then reconstructed using the Hadoop MapReduce framework (*HadoopDB*).

- The performance is up to three orders of magnitude more efficient than popular multi-node RDF data management systems.
• The master node also serves as the interface for SPARQL queries. It accepts queries and analyzes them closely.
• If it determines that the SPARQL pattern can be searched for completely in parallel by each worker in the cluster, then it sends the pattern to each node in the cluster for processing.
• If, it determines that the pattern requires some coordination across workers during the search, it decomposes the query into subgraphs that can be searched for in isolation, and ships these subgraphs to the worker nodes.
PigSPARQL

- A system which gives us the opportunity to process complex SPARQL queries on a MapReduce cluster.
- SPARQL queries are translated into Pig Latin, a data analysis language developed by Yahoo! Research.
- Pig Latin programs are executed by a series of MapReduce jobs on a Hadoop cluster.
- PigSPARQL offers not only a declarative way of specifying the transformation part, but also a scalable implementation of the whole ETL-process on a MapReduce cluster.
PigSPARQL

Alexander Schätzl, Martin Przyjaciel-Zablock and Georg Lausen: *PigSPARQL: Mapping SPARQL to Pig Latin*. SWIM, 2011
HBase vs MySQL Cluster

- A comparison between HBase and MySQL Cluster for SPARQL query processing.
- An experimental comparison of the two proposed approaches on a cluster of commodity machines LUMB.
- Both approaches were up to the task of efficiently storing and querying large RDF datasets.
- The HBase solution was capable of dealing with larger RDF datasets and showed superior query performance and scalability.
- Cloud computing has a great potential for scalable Semantic Web data management.

Craig Franke, Samuel Morin, Artem Chebotko, John Abraham and Pearl Brazier: Distributed Semantic Web Data Management in HBase and MySQL Cluster. IEEE CLOUD, 2011
Tutorial Overview

Session 1
- XQuery Overview – Sherif
- SPARQL Overview – Axel
- XSPARQL: a combined language – Axel
- Compression formats for XML+RDF: EXI+HDT – Sherif

Session 2
- XQuery implementations – Sherif
- SPARQL implementations – Sherif
- XSPARQL implementations – Axel
  (optional) Compression formats for XML+RDF: EXI+HDT – Sherif

Q/A - Discussion
XML Compression

a) XML data transfer **Without** compression
XML Compressors

XML Compressors

- Queriable
  - Homomorphic: e.g. XGrind
  - Non-Homomorphic: e.g. XQueC

- Archival
  - e.g. XMill
XML Compressors

• **General Text Compressors**: Since XML data are stored as text files, these compressors use the traditional general purpose text compression tools. They are XML-Blind, i.e. they treat XML documents as usual plain text documents and thus apply the traditional text compression techniques.

• **XML Conscious Compressors**: This group of compressors are designed to take the advantage of the awareness of the XML document structure in order to achieve better compression ratios over the general text compressors.
  
  – **Schema dependent compressors**: where both of the encoder and decoder must have access to the document schema information to achieve the compression process.
  
  – **Schema independent compressors**: where the availability of the schema information is not required to achieve the encoding and decoding processes.
XML Compressors

- **Non-Queriable (Archival) XML Compressors**: This group of the XML compressors does not allow any queries to be processed over the compressed format. The main focus of this group is to achieve the highest compression ratio. By default, general purpose text compressors belong to the non-queriable group of compressors.

- **Queriable XML Compressors**: This group of the XML compressors allow queries to be processed over their compressed formats. The compression ratio of this group is usually worse than that of the archival XML compressors. However, the main focus of this group is to avoid full document decompression during query execution.
  
  - *Homomorphic compressors*: where the original structure of the XML document is retained and the compressed format can be accessed and parsed in the same way of the original format.
  
  - *Non-homomorphic compressors*: where the encoding process of the XML document serrates the structural part from the data part. Therefore, the structure of the compressed format is different from the structure of the original XML document.
XMill

• Separates the structure from data and the grouping of the data values into homogenous containers based on their relative paths in the tree and their data types.

• Both of the structural and data value parts of the source XML document are collected and compressed separately.

• In the structure part, XML tags and attributes are encoded in a dictionary-based fashion before passing it to a back-end general text compression scheme.

• In the data part, data values are grouped into homogenous and semantically related containers according to their path and data type. Each container is then compressed separately using specialized compressor that is ideal for the data type of this container.

Hartmut Liefke, Dan Suciu: XMILL: An Efficient Compressor for XML Data. SIGMOD, 2000
XMill

XML File

```xml
<customers>
  <customer id="c1">
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <invoice total="300">
      <items>
        <item>item1</item>
        <item>item2</item>
      </items>
    </invoice>
  </customer>
  <customer id="c2">
    <firstName>Bill</firstName>
    <lastName>Luis</lastName>
  </customer>
</customers>
```

Elements Table

<table>
<thead>
<tr>
<th></th>
<th>/customers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/customers/customer</td>
</tr>
<tr>
<td>2</td>
<td>/customers/customer/firstName</td>
</tr>
<tr>
<td>3</td>
<td>/customers/customer/lastName</td>
</tr>
<tr>
<td>4</td>
<td>/customers/customer/invoice</td>
</tr>
<tr>
<td>5</td>
<td>/customers/customer/invoice/items/item</td>
</tr>
<tr>
<td>6</td>
<td>/customers/customer/invoice/items/items/item</td>
</tr>
</tbody>
</table>

Attributes Table

<table>
<thead>
<tr>
<th></th>
<th>/customers/customer/@id</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>/customers/customer/invoice/@total</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Customer Name</th>
<th>First Name</th>
<th>Last Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>John</td>
<td>Smith</td>
</tr>
<tr>
<td>c2</td>
<td>Bill</td>
<td>Luis</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>item1</td>
<td></td>
</tr>
<tr>
<td>item2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invoice ID</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>300</td>
</tr>
<tr>
<td>c2</td>
<td></td>
</tr>
</tbody>
</table>
XGrind

- XML-conscious compression scheme to support querying without the need for a full decompression of the compressed XML document.
- \textit{XGrind} does not separate data from structure. It retains the original structure of the XML document.
- Element and attribute names are encoded using a dictionary-based encoding.
- Character data is compressed using semi-adaptive Huffman coding.
- The query processor of \textit{XGrind} can only handle exact-match and prefix-match queries on compressed values and partial-match and range queries on decompressed values.
- Several operations are not supported by \textit{XGrind}, for example, non-equality selections in the compressed domain. Therefore, \textit{XGrind} cannot perform any join, aggregation, nested queries, or construct operations.

XML Compressors

- XMLPPM
  - http://xmlppm.sourceforge.net/
- XWRT
  - http://sourceforge.net/projects/xwrt/
- Exalt
  - http://exalt.sourceforge.net/
- Rngzip
  - http://contrapunctus.net/league/haques/rngzip/
- Benchmark
  - http://xmlcompbench.sf.net/
XML Fast Infoset

- http://www.w3.org/XML/Binary/

- An international standard that specifies a binary encoding format for XML documents.

- It aims to provide more efficient serialization than the text-based XML format.

- The underlying file format is with tag/length/value blocks.

- Text values of attributes and elements are therefore stored with length prefixes rather than end delimiters.

- An index table is built for most strings, which includes element and attribute names, and their values. This means that the text of repeated tags and values only appears once per document.

http://java.sun.com/developer/technicalArticles/xml/fastinfoset/
XML EXI

- W3C recommendation for a compact, high performance XML representation that is designed to work well for a broad range of applications.
- It aims of improving performance and significantly reducing bandwidth requirements without compromising efficient use of other resources such as battery life, code size, processing power, and memory.
- EXI uses a grammar-driven approach and a straightforward encoding algorithm and a small set of datatype representations. Therefore, EXI processors should be relatively simple and can be implemented on devices with limited capacity.

http://www.w3.org/XML/EXI/
XML EXI

• EXI is schema informed, meaning that it can utilize available schema information to improve compactness and performance, but does not depend on accurate, complete or current schemas to work.

• A program module called an EXI processor is used by application programs to encode their structured data into EXI streams (EXI stream encoder) and/or to decode EXI streams to make the structured data accessible (EXI stream decoder).

• An EXI stream is an EXI Header followed by an EXI body. The EXI body carries the content of the document, while the EXI header communicates the options used for encoding the EXI body.
XML EXI

• The building block of an EXI body is an EXI event. An EXI body consists of a sequence of EXI events representing an EXI document or an EXI fragment.

• The EXI events permitted at any given position in an EXI stream are determined by the EXI grammar. As is the case with XML, the events occur with nesting pairs of matching start element and end element events where any pair does not intersect with another except when it is fully contained in the other.

• The EXI grammar incorporates knowledge of the XML grammar and may be augmented and refined using schema information and fidelity options.
XML EXI

- Each EXI stream begins with an EXI header.
- The **EXI header** can identify EXI streams, distinguish EXI streams from text XML documents, identify the version of the EXI format being used, and specify the options used to process the body of the EXI stream.
- The EXI Options field within an EXI header is optional. Its presence is indicated by the value of the presence bit that follows distinguishing bits.
- When the *alignment* option is *byte-alignment*, padding bits of minimum length required to make the whole length of the header byte-aligned are added at the end of the header. On the other hand, there are no padding bits when the alignment in use is *bit-packed*.
XML EXI

• An EXI header MAY start with an **EXI Cookie** which is a four byte field that serves to indicate that the stream of which it is a part.

• The **Distinguishing Bits** is a two bit field of which the first bit contains the value 1 and the second bit contains the value 0. It is used to distinguish EXI streams from text XML documents in the absence of an EXI cookie.

• **EXI Format Version** identifies the version of the EXI format being used. EXI format version numbers are integers.

• **EXI Options** provides a way to specify the options used to encode the body of the EXI stream (e.g. alignment, compression, schemaid, blocksize, valueMaxLength)
XML EXI

- The rules for encoding a series of Events as an EXI stream are very simple and are driven by a declarative set of grammars that describes the structure of an EXI stream.
- Every event in the stream is encoded using the same set of encoding rules, which are summarized as follows:
  1. Get the next event data to be encoded
  2. If fidelity options indicate this event type is not processed, go to step 1
  3. Use the grammars to determine the event code of the event
  4. Encode the event code followed by the event content
  5. Evaluate the grammar production matched by the event
  6. Repeat until the End Document (ED) event is encoded
RDF Compression
Motivation of RDF Compression

- Large RDF datasets
- Verbosity and Redundancy.
- Examples
  - Billion Triple 2010 (~3200M triples, 318 gzipped chunks, ~27GB)
  - Uniprot (~845M, 12 gzipped chunks, ~23GB)
Approaches of RDF Compression

Javier D. Fernández, Claudio Gutierrez, Miguel A. Martinez-Prieto. RDF compression: basic approaches. WWW, 2010
## Approaches of RDF Compression

<table>
<thead>
<tr>
<th></th>
<th>B. Triples</th>
<th>Uniprot</th>
<th>US Census</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Original Size</strong></td>
<td>541.5</td>
<td>239.4</td>
<td>148.2</td>
</tr>
<tr>
<td>a) Direct Compr.</td>
<td>37.9 (7.0%)</td>
<td>7.5 (3.1%)</td>
<td>6.4 (4.3%)</td>
</tr>
<tr>
<td>b) Adjacency List</td>
<td>35.5 (6.6%)</td>
<td>5.3 (2.2%)</td>
<td>4.8 (3.3%)</td>
</tr>
<tr>
<td>c) Delta+Triples</td>
<td>37.9 (7.0%)</td>
<td>9.1 (3.8%)</td>
<td>9.0 (6.1%)</td>
</tr>
<tr>
<td>d) Trie+Triples</td>
<td>39.4 (7.3%)</td>
<td>9.7 (4.1%)</td>
<td>9.0 (6.1%)</td>
</tr>
</tbody>
</table>
Approaches of RDF Compression

- RDF data at big scale is highly compressible.
- Dedicated data structures, e.g. adjacency lists, code triples efficiently and facilitate compression (both string with ppmdi and integer with Huffman).
- RDF URIs are prone to efficient compression with standard techniques, but compression of literals deserve finer approaches.
- The structure of RDF graphs differs from XML or Web data, hence, classical approaches such as are not directly applicable.
HDT Overview

- **RDF**
  - Logical and physical metadata describing the RDF data set. It serves as an entrance point to the information.

- **Dictionary**
  - Mapping between elements in the data set and unique IDs, thus contributing to compactness.

- **Triples**
  - Structure of the data after the ID replacement, in a compressed form.

[Diagram of RDF and related components]

http://www.rdfhdt.org/
HDT Overview

1. Extraction of RDF Features
2. Dictionary Building
3. Triples Encoding
4. Practical HDT

RDF

Publication
Management
Compression
Exchange
DBPedia Example
<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"

xmlns:htd="http://purl.org/HTD/htd#"
xmlns:do="http://purl.org/do/terms/"
xmlns:void="http://rdfs.org/ns/void#"
xmlns:foaf="http://xmlns.com/foaf/0.1/"
xmlns:scovo="http://purl.org/NET/scovo#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:swp="http://www.w3.org/2004/03/swp-2/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"

<hdt:Dataset rdf:about="http://example.org/ex/DBpediaEN">
  <hdt:publicationInformation rdf:parseType="Resource">
    <void:dataDump rdf:resource="http://example.org/ex/DBpedia.rdf"/>
    <do:title>DBpediaEN</do:title>
    <do:publisher>
      <foaf:Organization>
        <foaf:homepage rdf:resource="http://example.org/theCompany"/>
      </foaf:Organization>
    </do:publisher>
    <do:issued>2010-10-01</do:issued>
  </hdt:publicationInformation>

  <hdt:statisticalInformation rdf:parseType="Resource">
    <void:statItem rdf:parseType="Resource">
      <scovo:dimension rdf:resource="void:numberOfTriples"/>
      <rdf:value>7</rdf:value>
    </void:statItem>
    <void:statItem>
      <scovo:dimension rdf:resource="htd:subjectObjectRatio"/>
      <rdf:value>12.5</rdf:value>
    </void:statItem>
  </hdt:statisticalInformation>
</hdt:Dataset>
<hdt:formatInformation rdf:parseType="Resource">
<!-- ... -->
</hdt:formatInformation>
<hdt:additionalInformation rdf:parseType="Resource">
  <swp:signatureMethod rdf:resource="swp:JjcC14N-md5-xor-rsa"/>
  <swp:signature rdf:datatype="xsd:base64Binary">AZ8QWE...</swp:signature>
</hdt:additionalInformation>
</hdt:Dataset>
</rdf:RDF>
HDT Statistics

• **out-degree**, $\text{deg}^-(s)$
  – the number of triples of $G$ in which $s$ occurs as subject
  – $\text{deg}^-(G)$, $\text{deg}^-(G)$

• **partial out-degree**, $\text{deg}^- - (s, p)$
  – the number of triples of $G$ in which $s$ occurs as subject and $p$ as predicate
  – $\text{deg}^- - (G)$, $\text{deg}^- - (G)$

• **labeled out-degree**, $\text{deg}L^-(s)$
  – the number of different predicates (labels) of $G$ with which $s$ is related as a subject in a triple of $G$
  – $\text{deg}L^-(G)$, $\text{deg}L^-(G)$

• **subject-object ratio**, $\alpha_s - o$
  – the proportion of common subjects and objects in the graph $G$
  – $\alpha_s - o = |SG \cap OG| / |SG \cup OG|$

• Symmetrically, **in-degrees**: $\text{deg}^+(o)$, $\text{deg}^+(G)$, etc.
DBPedia Example

Out-degree(page1) = 4
Partial out-degree(page1,#label) = 2
Labeled out-degree(page1) = 3

Out-degree(page2) = 2
Labeled out-degree(page2) = 2

Out-degree(page3) = 2
Partial in-degree(page3,#broader) = 2
Labeled in-degree(page3) = 1
Dictionary in Practice

• Subset distinction:
  – (1) Common subject-objects
  – (2) The non common subjects
  – (3) The non common objects
  – (4) Predicates

• List of strings matching the mapping of the four subsets, in order from (1) to (4).
  – A reserved character is appended to the end of each string and each vocabulary to delimit their size.
Triples

- Contains the structure of the data after the ID replacement.
Compact Triples

Subject Grouping

1 2 6 .
1 3 2 .
2 1 3 .
2 2 4 .
2 2 5 .
2 4 1 .
3 3 2 .

1 2 6 ; 3 2 .
2 1 3 ; 2 4 , 5 ; 4 1 .
3 3 2 .

Adjacency Lists Splitting

Predicates:

 subject 1  subject 2  subject 3

2 3 0 1 2 4 0 3 0

Objects:

6 0 2 0 3 0 4 5 0 1 0 2 0

Compact Triples
Binary Triples

Predicates:

Objects:

Compact Triples

Bitsequence-based reorganization

Bitmap Triples
HDT Bitmap Triples Compression

From imagination to impact

HDT-Plain / HDT-Compress

Text compression:
gzip, bz2, PPM

Specific compression:
Huffman (S), RRR (B)

Javier D. Fernández, Claudio Gutierrez, Miguel A. Martínez-Prieto. : Compact Representation of Large RDF Data Sets for Publishing and Exchange. ISWC, 2010
Tutorial Overview

Session 1
- XQuery Overview – Sherif
- SPARQL Overview – Axel
- XSPARQL: a combined language – Axel

Session 2
- XQuery implementations – Sherif
- SPARQL implementations – Sherif
- XSPARQL implementations – Axel
  (optional) Compression formats for XML+RDF: EXI+HDT – Sherif

Q/A - Discussion