CS 632: Course Seminar Presentation

On the paper

RDF-3X: a RISC-style Engine for RDF,

Thomas Neumann and Gerhard Weikum, PVLDB 2008

Presented by: Jiji Angel
Guided by: Prof S Sudarshan
Seminar Outline

- Introduction
  - RDF
  - SPARQL

- Implementation details of RDF-3X
  - Storage & Indexing
  - Query Processing & Optimization
  - Selectivity Estimates

- Experimental Setup & Evaluation Results

- Conclusion
Introduction

• RDF – Resource Description Framework
• Originally was used to model data for semantic web
• Primarily used for knowledge representation and data interchange

• Usages:
  – Ontology representation for semantic web
  – Knowledge base representation; Examples: Freebase, DBpedia, YAGO
  – Import/export data format
  – Non-proprietary data exchange format
RDF Triples

- In RDF every data item is represented using a triple
  \((\text{subject}, \text{predicate}, \text{object})\) aka \((\text{subject}, \text{property}, \text{value})\)

For example, information about the movie “Sweeney Todd” may be 'triplified' as:

- \((\text{id1}, \text{hasTitle}, 'Sweeney Todd')\),
- \((\text{id1}, \text{producedYear}, '2007')\),
- \((\text{id1}, \text{directedBy}, 'Tim Burton')\),
- \((\text{id1}, \text{hasCasting}, \text{id2})\),
- \((\text{id1}, \text{hasCasting}, \text{id3})\),
- \((\text{id2}, \text{roleName}, 'Sweeney')\),
- \((\text{id3}, \text{roleName}, 'Lovett')\),
- \((\text{id2}, \text{actor}, \text{id11})\),
- \((\text{id3}, \text{actor}, \text{id12})\),
- \((\text{id11}, \text{hasName}, 'Johny Depp')\),
- \((\text{id12}, \text{hasName}, 'Helena Carter')\)
Each set of triples is called an RDF graph.

Each triple is represented as a node-arc-node link; nodes denote subject or object; links denote the predicate.
RDF

- Extends the linking structure of the Web by using URIs (Uniform Resource Identifiers) for relationship
- Subjects and predicates are identified by URI values
- Schema language is RDFS (RDF Schema)

Triples are:
(http://www.w3.org/People/EM/contact#me, http://www.w3.org/2000/10/swap/pim/contact#fullName, “Eric Miller”),
(http://www.w3.org/People/EM/contact#me, http://www.w3.org/1999/02/22-rdf-syntax-ns#type, http://www.w3.org/2000/10/swap/pim/contact#Person)
Freebase – a knowledge base

- Open knowledge base; Collaboratively edited
- Creative Commons Attribution License
- Repository size: 47+ million topics, 2+ billion facts, (as of 15/03/2015)
- Initially seeded by pulling data from sources such as Wikipedia, MusicBrainz etc.
- Uses RDF graph model for data storage
- Freebase triplestore is named as graphd
- Developed by Metaweb and later acquired by Google
Introduction - SPARQL

- SPARQL – SPARQL Protocol and RDF Query Language
  - Official standard query language for querying RDF repositories
- SPARQL queries are basically pattern matching queries on triples from the RDF data graph
SPARQL Query Examples

- Select all the movie titles (assume that predicate <hasTitle> implies movie titles)

```
SELECT ?title
WHERE { ?x <hasTitle> ?title }
```

- Select the director of the movie 'Sweeney Todd'

```
SELECT ?directorName
WHERE { ?movieId <hasTitle> "Sweeney Todd".
       ?movieId <directedBy> ?directorName }
```

- Select all the roles and the actors who have acted those roles

```
SELECT (?role AS ?RoleName) (<?actor AS ?ActorName>)
WHERE { ?roleId <roleName> ?role.
       ?roleId <actor> ?actorId.
       ?actorId <hasName> ?name }
SPARQL Query Examples

- Select all the movie titles along with the year in which it was produced. 
  Make sure to get the movie titles even if the production year details are not available.

  ```sparql
  SELECT (?title AS ?MovieTitle) (?pYear AS ?ProductionYear)
  WHERE { ?movieId <hasTitle> ?title
     OPTIONAL {?movieId <producedYear> ?pYear}}
  ```

- Select all the movies released after all movies titled 'Sweeney Todd'.

  ```sparql
  SELECT (?title AS ?MoviesAfterSweeney)
  WHERE { ?movieId <hasTitle> ?title.
    ?movieId <producedYear> ?pYear.
    FILTER ( ?pYear <= ALL { SELECT ?pYearSweeney
      WHERE {?movieId <hasTitle> 'Sweeney Todd'.
        ?movieId <producedYear> ?pYearSweeney}}) }
  ```
SPARQL Query Examples

- To retrieve the titles of all the movies with Johny Depp by the SPARQL query:

\[
\text{SELECT} \ ?\text{title} \\
\text{WHERE} \ \{ \\
\ ?m \ <\text{hasTitle}> \ ?\text{title}. \ ?m \ <\text{hasCasting}> \ ?c. \ ?c \ <\text{actor}> \ ?a \ ?a \\
<\text{hasName}> \ "\text{Johny Depp}\" \ \} \\
\]

- To retrieve movie titles and the list them if the number of actors is more than 10, SPARQL query can be written as:

\[
\text{SELECT} \ (?\text{title} \ AS \ ?\text{movieTitle}) \ (\text{COUNT}(\?\text{actors}) \ AS \ ?\text{numberOfActors})) \\
\text{WHERE} \ \{ \\
\ ?x \ <\text{hasTitle}> \ ?\text{title}. \ ?x \ <\text{hasCasting}> \ ?y. \ ?y \ <\text{actor}> \ ?\text{actors} \\
\text{GROUP BY} \ ?x \\
\text{HAVING} \ (\text{COUNT}(\?\text{actors}) \ > \ 10) \\
\text{ORDER BY} \ ?\text{numberOfActors} \\
\]

Introduction – SPARQL Syntax

- Each pattern consists of S, P, O and each of these may be either a variable or a literal
- A dot(.) corresponds to join/conjunction; UNION keyword is used for disjunctions
- ORDER BY keyword : orders the result
- DISTINCT keyword : removes duplicates from the result
- REDUCED keyword : may but need not remove duplicates

```sparql
PREFIX foo: <...>
PREFIX rdf: <...>
SELECT [DISTINCT | REDUCED] ?variable1 ?variable2 ...
WHERE {
  pattern1. pattern2. ...
ORDER BY
LIMIT
OFFSET
```
Motivation and Problem

- Managing large-scale RDF data involves technical challenges:
  - Physical database design is difficult because of the absence of a global schema
  - RDF data is fine-grained and meant for on-the-fly applications; this calls for appropriate choice of query processing and optimization algorithms
  - Statistics gathering for join-order and execution-plan optimization is not very obvious
  - RDF stores data as graph rather than tree structure used by XML
Contribution & Outline

- **RDF-3X (RDF Triple eXpress)**
  - RDF-3X engine is an implementation of SPARQL that achieves excellent performance through RISC-style architecture, puristic data structures and operations
  - **Key Features:**
    - Physical design is workload independent. With exhaustive compressed indexes, it eliminates need for physical-design tuning
    - Query processor rely mostly on merge joins over sorted index lists
    - Query optimizer focuses on join order in generating the execution plan; dynamic programming for plan enumeration
      - Cost model is based on RDF-specific statistics synopsis
Storage and Indexing
RDF Data Storage

- There are three approaches followed by various implementations:
  - Giant Triple Table method
  - Property Table method
  - Cluster Property Table method
All triples are stored in a single, giant triple table with generic attributes subject, predicate, object.

RDF-3X follows this approach.

### Triple Table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>hasTitle</td>
<td>“Sweeney Todd”</td>
</tr>
<tr>
<td>id1</td>
<td>producedYear</td>
<td>2007</td>
</tr>
<tr>
<td>id1</td>
<td>directedBy</td>
<td>“Tim Burton”</td>
</tr>
<tr>
<td>id1</td>
<td>hasCasting</td>
<td>id2</td>
</tr>
<tr>
<td>id1</td>
<td>hasCasting</td>
<td>id3</td>
</tr>
<tr>
<td>id2</td>
<td>roleName</td>
<td>“Sweeney Todd”</td>
</tr>
<tr>
<td>id3</td>
<td>roleName</td>
<td>“Lovet”</td>
</tr>
<tr>
<td>id2</td>
<td>actor</td>
<td>id11</td>
</tr>
<tr>
<td>id3</td>
<td>actor</td>
<td>id12</td>
</tr>
<tr>
<td>id11</td>
<td>hasName</td>
<td>“Johny Depp”</td>
</tr>
<tr>
<td>id12</td>
<td>hasName</td>
<td>“Helena Carter”</td>
</tr>
</tbody>
</table>
Property Table method

- Separate tables for each predicate

### Triple Table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>hasTitle</td>
<td>“Sweeney Todd”</td>
</tr>
<tr>
<td>id1</td>
<td>producedYear</td>
<td>2007</td>
</tr>
<tr>
<td>id1</td>
<td>directedBy</td>
<td>“Tim Burton”</td>
</tr>
<tr>
<td>id1</td>
<td>hasCasting</td>
<td>id2</td>
</tr>
<tr>
<td>id1</td>
<td>hasCasting</td>
<td>id3</td>
</tr>
<tr>
<td>id2</td>
<td>roleName</td>
<td>“Sweeney Todd”</td>
</tr>
<tr>
<td>id3</td>
<td>roleName</td>
<td>“Lovet”</td>
</tr>
<tr>
<td>id2</td>
<td>actor</td>
<td>id11</td>
</tr>
<tr>
<td>id3</td>
<td>actor</td>
<td>id12</td>
</tr>
<tr>
<td>id11</td>
<td>hasName</td>
<td>“Johny Depp”</td>
</tr>
<tr>
<td>id12</td>
<td>hasName</td>
<td>“Helena Carter”</td>
</tr>
</tbody>
</table>

### Property Table

<table>
<thead>
<tr>
<th>hasTitle</th>
<th>S</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>“Sweeney Todd”</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>producedYear</th>
<th>S</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>2007</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>directedBy</th>
<th>S</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>“Tim Burton”</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>roleName</th>
<th>S</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>id2</td>
<td>“Sweeney Todd”</td>
<td></td>
</tr>
<tr>
<td>id3</td>
<td>“Lovet”</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>actor</th>
<th>S</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>id2</td>
<td>id11</td>
<td></td>
</tr>
<tr>
<td>id3</td>
<td>id12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasName</th>
<th>S</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>id11</td>
<td>“Johny Depp”</td>
<td></td>
</tr>
<tr>
<td>id12</td>
<td>“Helena Carter”</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hasCasting</th>
<th>S</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>id2</td>
<td></td>
</tr>
<tr>
<td>id1</td>
<td>id3</td>
<td></td>
</tr>
</tbody>
</table>
Cluster-property Table method

- Correlated predicates are kept together in a single table

### Triple Table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>hasTitle</td>
<td>“Sweeney Todd”</td>
</tr>
<tr>
<td>id1</td>
<td>producedYear</td>
<td>2007</td>
</tr>
<tr>
<td>id1</td>
<td>directedBy</td>
<td>“Tim Burton”</td>
</tr>
<tr>
<td>id1</td>
<td>hasCasting</td>
<td>id2</td>
</tr>
<tr>
<td>id1</td>
<td>hasCasting</td>
<td>id3</td>
</tr>
<tr>
<td>id2</td>
<td>roleName</td>
<td>“Sweeney Todd”</td>
</tr>
<tr>
<td>id3</td>
<td>roleName</td>
<td>“Lovet”</td>
</tr>
<tr>
<td>id2</td>
<td>actor</td>
<td>id11</td>
</tr>
<tr>
<td>id3</td>
<td>actor</td>
<td>id12</td>
</tr>
<tr>
<td>id11</td>
<td>hasName</td>
<td>“Johny Depp”</td>
</tr>
<tr>
<td>id12</td>
<td>hasName</td>
<td>“Helena Carter”</td>
</tr>
</tbody>
</table>

### Property Table

<table>
<thead>
<tr>
<th>Subject</th>
<th>hasCasting</th>
<th>roleName</th>
<th>actor</th>
<th>hasName</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>id2</td>
<td>“Sweeney Todd”</td>
<td>id11</td>
<td>“Johny Depp”</td>
</tr>
<tr>
<td>id1</td>
<td>id3</td>
<td>“Lovet”</td>
<td>id12</td>
<td>“Helena Carter”</td>
</tr>
</tbody>
</table>

### Left Over Triple Table

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>id1</td>
<td>hasTitle</td>
<td>“Sweeney Todd”</td>
</tr>
<tr>
<td>id1</td>
<td>producedYear</td>
<td>2007</td>
</tr>
<tr>
<td>id1</td>
<td>directedBy</td>
<td>“Tim Burton”</td>
</tr>
<tr>
<td>id11</td>
<td>hasName</td>
<td>“Johny Depp”</td>
</tr>
<tr>
<td>id12</td>
<td>hasName</td>
<td>“Helena Carter”</td>
</tr>
</tbody>
</table>
Triple Store and Mapping Dictionary

- RDF-3X uses giant triple table approach
  - Drawback – literals can be very large and may contain lot of redundancy

- Solution used by RDF-3X:
  - Use dictionary compression: Mapping Dictionary
    - Compresses the triple store
    - Fast query processing
  - Store all the triples in a clustered B\(^+\)-tree
    - Triples are sorted lexicographically
    - Eases SPARQL range queries
Mapping Dictionary

- Used to map literals to a corresponding id
  - This compresses the triple store
  - Simplifies query processing
- Incurs a minor cost of additional dictionary indices
Compressed Indexes

- When literals are prefixes and variables are suffixes in the pattern, the query acts like a range query; suffices to have single index-range-scan
  - For example: (literal1, literal2, ?x)
- To guarantee that queries with all possible patterns are answered in a single index scan, RDF-3X maintain all six possible permutations of subject(S), predicate(P) and object(O), in six separate indices
  - SPO, SOP, OSP, OPS, PSO, POS
  - Triples in the index are sorted lexicographically
  - Are directly stored in the leaf pages of the clustered B+-tree
  - This ordering causes neighboring triples to be very similar
  - Hence compression of triples is possible: instead of storing full triples RDF-3X stores only the changes between the triples
Sort Orders

- Which sort order to choose?
  - 6 possible orderings, store all of them (SPO, SOP, OSP, OPS, PSO, POS)
  - Will make merge joins very convenient

- Each SPARQL triple pattern can be answered by a single range scan

- Eg: If we need to know all actors of a film, the subject ("Film object") and predicate (<hasActor>) remain the same. So, we use the index on sort order “SPO”

- On the other hand, if we need to find all movies in which an actor has acted, the object ("Actor") and predicate (<hasActor>) remain the same. So, the index on sort order “OPS” would be more suitable
Compressed Triple Structure

- Comparison of triples is the difference in their id values
  - Triples are sorted lexicographically which allows SPARQL pattern matching into range scans
  - Can be compressed well (delta encoding)
  - Efficient scan, fast lookup if prefix is known
  - Structure of byte-level compressed triple is

<table>
<thead>
<tr>
<th>Gap</th>
<th>Payload</th>
<th>Delta</th>
<th>Delta</th>
<th>Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Bit</td>
<td>7 Bits</td>
<td>0-4 Bytes</td>
<td>0-4 Bytes</td>
<td>0-4 Bytes</td>
</tr>
</tbody>
</table>

- Header byte denotes number of bytes used by the three values (5*5*5=125 size combinations)
- Gap bit is used when only value3 changes and delta is less than 128 (that fits in header)
Triple Compression Algorithm

\[
\text{compress}\left((v_1, v_2, v_3), (\text{prev}_1, \text{prev}_2, \text{prev}_3)\right)
\]

// Writes (v1, v2, v3) relative to (prev1, prev2, prev3)

if \(v_1 = \text{prev}_1 \land \text{prev}_2 = \text{prev}_2\)

    if \(v_3 - \text{prev}_3 < 128\)
        write \(v_3 - \text{prev}_3\)
    else \(\text{encode}(0, 0, v_3 - \text{prev}_3 - 128)\)

else if \(v_1 = \text{prev}_1\)

    \(\text{encode}(0, v_2 - \text{prev}_2, v_3)\)

else

    \(\text{encode}(v_1 - \text{prev}_1, v_2, v_3)\)

\[
\text{encode}(\delta_1, \delta_2, \delta_3)
\]

// Writes the compressed tuple corresponding to the deltas

write \(128 + \text{bytes} (\delta_1) * 25 + \text{bytes} (\delta_2) * 5 + \text{bytes} (\delta_3)\)

write the non-zero tail bytes of \(\delta_1\)

write the non-zero tail bytes of \(\delta_2\)

write the non-zero tail bytes of \(\delta_3\)
Compressing Triple Example

- Example 1: Suppose the first triple is (10,20,1123) and the next triple is (10,20,1173).
  
  \[ v_1 = \text{prev}_1 \text{ and } v_2 = \text{prev}_2 \]
  
  Also, \( v_3 - \text{prev}_3 < 128 \)
  
  Hence, the delta entry would be \( 1173 - 1123 = 50 \) in the header record.
  
  Hence, the size of this tuple is only 1 byte; gap bit set to 0.

- Example 2: Suppose the first triple is (10,20,1000), second triple is (10,20,1500).
  
  \[ v_1 = \text{prev}_1 \text{ and } v_2 = \text{prev}_2; \text{ but } (1500 - 1000) = 500 !< 128 \]
  
  Function call: \textbf{encode (0,0,372)}
  
  Header will contain \( 128 + 0 + 0 + 2 = 130 \)
  
  \( \delta_1 \) has 0 non-zero bytes, \( \delta_2 \) has 0 non-zero byte, \( \delta_3 \) has 2 non-zero bytes.
  
  Hence, the overall size of the tuple will be 3 bytes.
Aggregated Indices

• For many SPARQL queries indexing partial triples rather than full triples would be sufficient

  SELECT ?a ?c
  WHERE { ?a ?b ?c}

• Aggregated Indices:
  - Two-value indices: Each of the possible pairs out of a triple (SP, PS, SO, OS, PO, OP) and the number of occurrences of each pair in the full set of triples
  - One-value indices: Three one valued indices, (S/P/O, count) are stored
RDF-3X Indexing – Three Types

Indices

- Six triple indexes: SPO, PSO, SOP, OSP, POS, OPS
- Six two valued aggregated indices and their count: SP, PS, PO, OP, SO, OS
- Three one valued aggregate indices and the respective counts
- Experimentally total size of all indexes is less than original data
Query Processing and Optimization
Translating SPARQL Queries

- **Step 1**: Convert the SPARQL query into a query graph representation, interpreted as relational tuple calculus expression
- **Step 2**: Conjunctions are parsed and expanded into a set of triple patterns
- **Step 3**: Literals are mapped to ids through dictionary lookup
- **Step 4**: Multiple query patterns are computed by joining individual triple patterns
- **Step 5**: If distinct results are to be obtained, duplicates are removed from the result
- **Step 6**: The result contains ids now; dictionary lookup is performed to get back the actual string equivalents
SPARQL Query Graph

- Each triple pattern corresponds to one node in the query graph
- An edge between two nodes is a common query variable

**SELECT ?title WHERE {**

- ?m <hasTitle> ?title.
- ?m <hasCasting> ?c.
- ?a <actor> ?c.
- ?a <hasName> “Johny Depp”}

**SPARQL query**

```
P1 = ?m <hasTitle> ?title
P2 = ?m <hasCasting> ?c
P3 = ?a <actor> ?c
P4 = ?a <hasName> “Johny Depp”
```

**Triple Form**

- $P_2.c = P_3.c$
- $P_1.m = P_2.m$
- $P_3.a = P_4.a$

**Possible Join Tree**

**Query Graph**
Optimizing Join Ordering

• SPARQL query execution demands join queries which can be really complex:
  – SPARQL queries contain star-shaped subqueries and hence strategies to handle bushy join trees are required
  – Since large number of joins are common in SPARQL queries, fast plan enumeration and cost optimization are required

• RDF-3X uses decision cost based dynamic programming approach for optimizing join orderings
DP Based Join Optimization

- RDF-3X uses bottom-up dynamic programming approach
  - Takes a connected query graph as input and outputs an optimal bushy join tree
  - Enumerates DP table with the initial set of triples efficiently and correctly
  - Unused (unbound) variables are projected away by using aggregated index
  - The plans that are costlier and equivalent to other plans are pruned
    - Sometimes plans are retained even if they are costlier based on order optimization
  - The larger optimal plan is generated by joining optimal solutions to smaller problems that are adjacent in the query graph
Selectivity Estimates

- Identification of lowest-cost execution plan hugely relies on the estimated cardinalities and selectivities
- A bit different from standard join ordering:
  - One big "relation", no schema
  - Selectivity estimates are hard
  - Standard single attribute synopses are not very useful:
    - Only three attributes and one big relation;
    - But (?a, ?b, "Mumbai") and (?a, ?b, "1974-05-30") produces vastly different values for ?a and ?b
- Two kinds of statistics are maintained by RDF-3X
  - Selectivity Histograms
  - Frequent Join Paths
Selectivity Histograms

- Query optimizer uses aggregated indexes for calculations based on triple cardinalities.

- For estimating join selectivity, histogram buckets with additional information are maintained, as follows.

---

<table>
<thead>
<tr>
<th>Start (s, p, o)</th>
<th>End (s, p, o)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of triples</td>
<td>3000</td>
</tr>
<tr>
<td>Number of distinct 2-prefixes</td>
<td>3</td>
</tr>
<tr>
<td>Number of distinct 1-prefixes</td>
<td>1</td>
</tr>
<tr>
<td>Join partners on subject</td>
<td>4000 0 200</td>
</tr>
<tr>
<td>Join partners on predicate</td>
<td>50 400000 200</td>
</tr>
<tr>
<td>Join partners on object</td>
<td>6000 0 9000</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Range Start</th>
<th>Range End</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10, 2, 30)</td>
<td>(10, 5, 12000)</td>
</tr>
<tr>
<td>Number of triples = 3000</td>
<td></td>
</tr>
<tr>
<td>Number of distinct 2-prefixes = 3</td>
<td></td>
</tr>
<tr>
<td>Number of distinct 1-prefixes = 1</td>
<td></td>
</tr>
<tr>
<td>Join partners on subject</td>
<td>4000 0 200</td>
</tr>
<tr>
<td>Join partners on predicate</td>
<td>50 400000 200</td>
</tr>
<tr>
<td>Join partners on object</td>
<td>6000 0 9000</td>
</tr>
</tbody>
</table>

---

**Bucket structure**

**Example Bucket implementation**
Selectivity Histograms

- Generic but assumes predicates are independent
- Aggregates indexes until they fit into one page
- Merge smallest buckets (equi-depth)
- For each bucket compute statistics
- 6 indexes, pick the best for each triple pattern
- Assumes uniformity and independence, but works quite well
Frequent Paths

- Correlated predicates appear in SPARQL queries in two ways:
  - Stars of triple patterns: a number of triple patterns with different predicates sharing the same subject
    
    ```sparql
    SELECT r_1, r_n
    WHERE { (r_1 p_1 r_2). (r_1 p_2 r_3). ... (r_1 p_n r_n)}
    ```
  - Chains of triple patterns: a number of triple patterns where object of the first pattern is subject of the second pattern
    
    ```sparql
    SELECT r_1, r_{n+1}
    WHERE { (r_1 p_1 r_2). (r_2 p_2 r_3). ... (r_n p_n r_{n+1})}
    ```

- Most frequent paths (i.e., the paths with the largest cardinalities) are computed, the result cardinalities are materialised along with the path description p_1, p_2, ... p_n.
Frequent Path Mining Algorithm

`FrequentPath(k)`

// Computes the k most frequent paths

\[ C_1 = \{P_p | p \text{ is a predicate in the database}\} \]

sort \( C_1 \), keep the \( k \) most frequent

\( C = C_1, i = 1 \)

**do**

\( C_{i+1} = \phi \)

**for each** \( p' \in C, p \text{ predicate in the database} \)

**if** top \( k \) of \( C \cup C_{i+1} \cup \{P_{pp'}\} \) include all subpaths of \( pp' \)

\[ C_{i+1} = C_{i+1} \cup \{P_{pp'}\} \]

**if** top \( k \) of \( C \cup C_{i+1} \cup \{P_{pp'}\} \) include all subpaths of \( pp' \)

\[ C_{i+1} = C_{i+1} \cup \{P_{pp'}\} \]

\( C = C \cup C_{i+1}, \) sort \( C \), keep \( k \) the most frequent

\[ C_{i+1} = C_i \cap C, i = i + 1 \]

**while** \( C_i \neq \phi \)

**return** \( C \)
Estimates for Composite Queries

- Combining histogram with frequent path statistics
- Long join chain decomposed to subchains of maximal length
  - For example consider a query like:
    \[
    \text{?x}_1 \text{a}_1 \text{v}_1 . \text{?x}_1 \text{p}_1 \text{?x}_2 . \text{?x}_2 \text{p}_2 \text{?x}_3 . \text{?x}_3 \text{p}_3 \text{?x}_4 . \\
    \text{?x}_4 \text{a}_4 \text{v}_4 . \text{?x}_4 \text{p}_4 \text{?x}_5 . \text{?x}_5 \text{p}_5 \text{?x}_6 . \text{?x}_6 \text{a}_6 \text{v}_6
    \]
- For subchains \( \text{p}_1 - \text{p}_2 - \text{p}_3 \) and \( \text{p}_4 - \text{p}_5 \), selectivity estimation is done using frequency path and for selections histograms are used
- In absence of any other statistics, assume the above two estimators as probabilistically independent - use product formula with per-chain and per-selection statistics as factors
Evaluation

• RDF-3X is compared with:
  – MonetDB (column store approach)
  – PostgreSQL (triple store approach)

• Three different data sets:
  – Yago, Wikipedia-based ontology: 1.8GB
  – LibraryThing : 3.1
  – Barton library data : 4.1GB
Evaluation
sample query (Q5) select ?a ?c where
{ ?a <origin> <marcorg/DLC>. ?a <records> ?b. ?b <type>?c. filter (?c != <Text>) }
Conclusion

- RDF-3X is a fast and flexible RDF/SPARQL engine
  - Exhaustive but very space-efficient triple indexes
  - Avoids physical design tuning, generic storage
  - Fast runtime system, query optimization has a huge impact
Questions
Thank You