Merging RDF Characteristic Sets to Optimize SPARQL Queries

Marios Meimaris and George Papastefanatos
Preliminaries

• RDF (Resource Description Framework)
  • Abstract Data model for Linked Data
  • Based on *Triples*: Subject-Predicate-Object
  • RDF datasets are *Directed Labelled Graphs*

• Characteristic Set (CS)
  • A CS is a set of properties with the same subject as source node
  • An RDF dataset can be described as a set of unique CSs
  • Each CS is an *implicit resource type*
Motivation

Web of Data

› Large Volumes
› Schema Diversity
› Loose Structure
› Complex analysis needs

The need for fast and efficient indexing and complex query processing methods on voluminous and diverse data arises...
State of the Art

 › Well-established approaches:
   › Exhaustive permutation indexing
     › (SPO, PSO, OPS, etc.)
     › Partial Permutated Indexes (SO, OS etc.)
     › Facilitation of merge joins (where possible)
   › Property tables
     › Each node type becomes a relational table
     › Fast grouping and retrieval by concept type
   › Vertical partitioning
     › Each property becomes a relational table
     › Very simple design
Challenge

We target **efficient SPARQL query processing**:

› **Query Characteristics:**
  › long chain patterns (object-subject joins)
  › descriptive star patterns (subject-subject joins)

› **Data Characteristics:**
  › large volume
  › semi-structured (loosely-defined schema)
SELECT DISTINCT *
WHERE {
  ?x lubm:researchInterest ?o1 .
  ?x lubm:mastersDegreeFrom ?o2 .
  ?x lubm:doctoralDegreeFrom ?o3 .
  ?x lubm:memberOf ?y .
  ?z lubm:hasAlumnus ?o7
}
Approach

› Use Characteristic Sets (CSs) and their links in order to store and index triples

› **Characteristic Sets** (Neumann & Moerkotte, ICDE 2011)
  › A *Characteristic Set* (CS) $S_c$ of a node $x$ is defined as the set of properties emitting from $x$ (i.e., $x$ as subject)

$$S_c(John) = \{\text{name, origin, worksAt, type}\}$$

$$S_c(Alice) = \{\text{name, origin, studiesAt, type}\}$$
Approach

› Derive a relational representation of an RDF dataset
› Use CSs as tables and links between CSs as relationships
  › SS joins (star patterns) are very easy to compute
  › SO / OS joins (chain patterns) become simple semi-joins between tables
  › Combined SS star/chain patterns are answered efficiently

› Problems
  › many CSs with small numbers of triples
  › few CSs with large numbers of triples
  › huge schema overlaps between them
Observations

› Based on previous findings:
  › CS number is generally low but exhibits skewed distribution
    › E.g., many CSs with very few (<10) subjects
    › CS number affects number of joins

› Merging closely related CSs helps storage & querying
  › Less CSs means less joins
  › Less CSs means less I/O costs in disk-based systems
  › Compact schema easier to understand and maintain

› CSs are hierarchical, i.e., their property sets can be super/subsets of each other

› Challenge: exploit the hierarchical structure in order to merge together closely related CSs
Challenge

› Each CS defines a relational table \((s, p_1, p_2, \ldots, p_k)\)
› Merging of CS tables results in NULL values for non-shared attributes
› Challenge: merge CSs and reduce NULL value effect

e.g.:
\(c_0 = \{\text{name, age}\}\)
\(c_1 = \{\text{name, age, marriedTo}\}\)
\(c_2 = \{\text{name, age, marriedTo, worksAt}\}\)
Approach

› Use a **dense child** table and merge its parents into it
  › **Why dense?** -> # of NULLs is proportional to # of records of table to be merged
  › **Why child?** -> more specialized, thus will contain columns of parents

› Identify dense CSs
  › if \(|c_i| > m \times |c_{max}|\) parameter => \(c_i\) is dense
  › Every resulting (merged) table will contain **exactly one dense node** (and several non-dense)

› Find optimal merging of ancestors to dense child CSs

  e.g.,
  \(c_1\): \{name, age, address\}, \(c_2\): \{name, age\}: \(c_1\) child of \(c_2\)
  \[\text{hier\_merge}(c_1, c_2) = c_{12}\]
  \(c_{12}\): \{name, age, address\}
Approach - Example
Approach – Loading and Merging

› Finding the optimal solution is equivalent to enumerating all possible sub-graphs -> **exponential**

› **Greedy approximation**
  › At each step, merge parent CS and dense child CS that minimize objective cost function
  › Cost function minimizes the number of NULL values introduced by the merge

› Tuning of $m$ parameter
Approach – Querying

› Parse incoming SPARQL queries
  › Identify query CSs that match merged CSs in the dataset
  › Rewrite query as an SQL statement with UNIONs between matched CSs
  › In case of SO/OS joins, prune off CSs that are not linked

› Pass final query to relational optimizer

› Build and output results
Implementation & Evaluation

› Implemented *raxonDB* on top of relational backbone
› Evaluated on real & synthetic data:
  › Geonames (~150m triples)
  › Reactome (~15m triples)
  › LUBM (up to 350m triples)
  › WatDiv (up to 100m triples)
› Measured Loading Time, Disk Size, Query Processing
› Compared with plain ECS indexing and state of the art competitors
# Implementation & Evaluation (Loading)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Size (MB)</th>
<th>Time</th>
<th>Tables (CSs)</th>
<th># of ECSs</th>
<th>Dense CS Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reactome Simple</td>
<td>781</td>
<td>3min</td>
<td>112</td>
<td>346</td>
<td>100%</td>
</tr>
<tr>
<td>Reactome (m=0.05)</td>
<td>675</td>
<td>4min</td>
<td>35</td>
<td>252</td>
<td>97%</td>
</tr>
<tr>
<td>Reactome (m=0.25)</td>
<td>865</td>
<td>4min</td>
<td>14</td>
<td>73</td>
<td>77%</td>
</tr>
<tr>
<td>Geonames Simple</td>
<td>4991</td>
<td>69min</td>
<td>851</td>
<td>12136</td>
<td>100%</td>
</tr>
<tr>
<td>Geonames (m=0.0025)</td>
<td>4999</td>
<td>70min</td>
<td>82</td>
<td>2455</td>
<td>97%</td>
</tr>
<tr>
<td>Geonames (m=0.05)</td>
<td>5093</td>
<td>91min</td>
<td>19</td>
<td>76</td>
<td>87%</td>
</tr>
<tr>
<td>Geonames (m=0.1)</td>
<td>5104</td>
<td>92min</td>
<td>6</td>
<td>28</td>
<td>83%</td>
</tr>
<tr>
<td>LUBM Simple</td>
<td>591</td>
<td>3min</td>
<td>14</td>
<td>68</td>
<td>100%</td>
</tr>
<tr>
<td>LUBM (m=0.25)</td>
<td>610</td>
<td>3min</td>
<td>6</td>
<td>21</td>
<td>90%</td>
</tr>
<tr>
<td>LUBM (m=0.5)</td>
<td>620</td>
<td>3min</td>
<td>3</td>
<td>6</td>
<td>58%</td>
</tr>
<tr>
<td>WatDiv Simple</td>
<td>4910</td>
<td>97min</td>
<td>5667</td>
<td>802</td>
<td>100%</td>
</tr>
<tr>
<td>WatDiv (m=0.01)</td>
<td>5094</td>
<td>75min</td>
<td>67</td>
<td>99</td>
<td>77%</td>
</tr>
<tr>
<td>WatDiv (m=0.1)</td>
<td>5250</td>
<td>75min</td>
<td>25</td>
<td>23</td>
<td>63%</td>
</tr>
<tr>
<td>WatDiv (m=0.5)</td>
<td>5250</td>
<td>77min</td>
<td>16</td>
<td>19</td>
<td>55%</td>
</tr>
</tbody>
</table>
Implementation & Evaluation (Querying)

(a) Execution time (seconds) for LUBM
(b) Execution time (seconds) for Geonames
(c) Execution time (seconds) for Reactome

(a) Execution time (seconds) for LUBM2000
(b) Execution time (seconds) for Geonames
(c) Execution time (seconds) for Reactome
Papers and more info

› Ongoing (technical report):

› Previous works:
Future Work

- Optimal merging in polynomial time
- Better cost functions
- Distributed version of raxonDB
  - Exploit well-established relational backbones
    - Impala (SQL Engine) over Hive or Kudu
Thank you

Questions?

This research is funded by the project VisualFacts (#1614) - 1st Call of the Hellenic Foundation for Research and Innovation Research Projects for the support of post-doctoral researchers.