Querying Graph-Structured Data

Thomas Neumann

Technische Universität München

November 4, 2016
Motivation

Many interesting data sets of a graph structure.

- very flexible
- easy to model
- but difficult to query
- often very large
- no obvious structure
- how to store and process?

**Linked Open Data** cloud is use. Contains data sets with billions of entries.
Graph-structured data

One way to model graph-structured data is to use RDF (Resource Description Framework).

- conceptually a directed graph with edge labels
- each edge represents a fact (triple in RDF notation)
- triples have the form \((subject, predicate, object)\)

Example:

- \(<\text{obj}_1 > \text{cityName} > 'Berlin'\)
- \(<\text{obj}_1 > \text{isCapitalOf} > \text{obj}_2 >\)
- \(<\text{obj}_2 > \text{countryName} > 'Germany'\)

Everything is encoded as triples, queries operate on triples.
All capitals in Europe:

SELECT ?capital ?country
WHERE {
  ?x <cityName> ?capital.
  ?x <isCapitalOf> ?y.
  ?y <countryName> ?country.
  ?y <isInContinent> <Europe>.
}

- querying via pattern matching in RDF graph
- queries are sets of triple patterns
- variable occurrences imply joins

**Problem:** huge graph, many variable bindings possible
How to process SPARQL queries?

- we could use a (relational) database
- load the graph as triples into a table
- patterns form filters and joins
- produces the correct answer
- but very inefficient
- the database does not “understand” the graph structure
- a specialized RDF engine is more efficient
- I will talk about RDF-3X here (open source)
Indexing RDF Graphs

Primary data structure: clustered $B^+$-trees

- stores triples in lexicographical order
- allows for good compression (differences are small)
- sequential disk accesses, fast lookups

Example: Sort order $(S,P,O)$, triple pattern: $(\text{obj}_1, \text{pred}, ?x)$
⇒ Read range $(\text{obj}_1, \text{pred}, -\infty)$-$($$(\text{obj}_1, \text{pred}, \infty)$$)$ in $B^+$-tree

Which sort order to choose?

- index is heavily compressed, space consumption not that critical
- $3! = 6$ possible Orderings ⇒ 6 $B^+$-trees
- always the 'right' sort order available, efficient merge joins

e.g. $?x <\text{cityName}> ?\text{capital}. $?x <\text{isCapitalOf}> ?\text{y}$. ⇒
$(\text{cityName}, ?x, ?\text{capital})_{PSO} \Join (\text{isCapitalOf}, ?x, ?\text{y})_{PSO}$
Runtime Improvements

RDF-3X uses many techniques to improve runtime performance:

- compressed B-trees reduce size and improve I/O performance
- exhaustive indexing often allows for cheap merge joins
- sideways information passing skips over large parts of the data
- works on compressed/encoded data as much as possible
- ...

Optimize performance and minimize disk I/O.
Indexing is Not Enough

```sql
select *
where {
    ?s yago:created ?product.
    ?s yago:hasLatitude ?lat.
    ?s yago:hasLongitude ?long
}
```

Suboptimal: $|\bowtie_1| = 140$ Mln
Runtime: 65 ms

Optimal: $|\bowtie_1| = 14$ K
Runtime: 20 ms

Query optimization has a huge impact, sometimes orders of magnitudes.
Cardinality Estimation

Traditional estimating:
- estimates for individual predicates and joins
- combined assuming independence
- statistical synopses

Not well suited for RDF data
Why are Standard Histograms not Enough?

Some number from the Yago data set:

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sel(\sigma P=\text{isCitizenOf}))</td>
<td>(1.06 \times 10^{-4})</td>
</tr>
<tr>
<td>(sel(\sigma O=\text{United}_{\text{States}}))</td>
<td>(6.41 \times 10^{-4})</td>
</tr>
<tr>
<td>(sel(\sigma P=\text{isCitizenOf} \land O=\text{United}_{\text{States}}))</td>
<td>(4.86 \times 10^{-5})</td>
</tr>
<tr>
<td>(sel(\sigma P=\text{isCitizenOf}) \ast sel(\sigma O=\text{United}_{\text{States}}))</td>
<td>(6.80 \times 10^{-8})</td>
</tr>
</tbody>
</table>

- independence assumption does not hold
- leads to severe underestimation
- multi-dimensional histograms would help (expensive)
- looking at individual triples is not enough

For RDF data, **correlation is the norm!**
Why is Correlation a Problem?

Correlation occurs across triples:

- some triples are closely related
- independence does not hold

Very common:

- soft functional dependencies
- if we know bind triple pattern, the others become unselective
- not captured by attribute histograms

Example Triples

\[
\begin{align*}
&< o_1 > <\text{title}> "\text{The Tree and I}". \\
&< o_1 > <\text{author}> <\text{R. Pecker}>. \\
&< o_1 > <\text{author}> <\text{D. Owl}>. \\
&< o_1 > <\text{year}> "1996". \\
\end{align*}
\]
Why Not Sampling?

RDF is very unfriendly for sampling

- no schema
- one huge "relation"
- billions of tuples
- very diverse

Sample would have to be huge to be useful.
Capturing Correlations

We classify the tuples using \textit{characteristic sets}

- compact data structure
- groups triples by ”behavior”
- within a group, triples are more homogeneous
- groups are annotated with occurrence statistics
- allows for deriving estimates for whole query fragments
- captures correlations within tuples and across tuples

Allows for very accurate cardinality estimates.
Characteristic Sets

Observation: nodes are characterized by outgoing edges

\[ S_C(s) := \{ p | \exists o : (s, p, o) \in R \}. \]
\[ S_C(R) := \{ S_C(s) | \exists p, o : (s, p, o) \in R \}. \]

Example

\(< o_1 > <title> ”The Tree and I”. < o_1 > <author> <R. Pecker>. < o_1 > <author> <D. Owl>. < o_1 > <year> ”1996”. < o_2 > <title> ”Emma”. < o_2 > <author> <J. Austen>. < o_2 > <year> ”1815”. <J. Austen> <hasName> ”Jane Austen”. <J. Austen> <bornIn> <Steventon>.\]

\[ S_C(o_1) = \{ title, author, year \} \]
\[ S_C(o_2) = \{ title, author, year \} \]

\[ S_C = \{ \{ title, author, year \}^2, \{ hasName, bornIn \}^1 \} \]
Estimating Distinct Subjects

We can use characteristic sets for cardinality estimation

query: 
```
select distinct ?e
where { ?e <author> ?a. ?e <title> ?t. }
```

cardinality: 
```
\sum_{S \in \{ S \in S_C(R) \land \{ \text{author}, \text{title} \} \subseteq S \}} \text{count}(S)
```

- the computation is exact! (only for \textit{distinct}, though)
- can estimate a large number of joins in one step
- number of characteristic sets is surprisingly low

### Number of Characteristic Sets

<table>
<thead>
<tr>
<th></th>
<th>triples</th>
<th>characteristic sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yago</td>
<td>40,114,899</td>
<td>9,788</td>
</tr>
<tr>
<td>LibraryThing</td>
<td>36,203,751</td>
<td>6,834</td>
</tr>
<tr>
<td>UniProt</td>
<td>845,074,885</td>
<td>613</td>
</tr>
</tbody>
</table>
Occurrence Annotations

Without *distinct* we need occurrence annotations

| distinct | \(|\{|s| \exists p, o : (s, p, o) \in R \land S_C(s) = S\}|\) |
|----------|--------------------------------------------------|
| count\((p_1)\) | \(|\{(s, p_1, o) | (s, p_1, o) \in R \land S_C(s) = S\}|\) |
| count\((p_2)\) | \(|\{(s, p_2, o) | (s, p_2, o) \in R \land S_C(s) = S\}|\) |
| ... | ... |

Example

```
```

<table>
<thead>
<tr>
<th>distinct</th>
<th>author</th>
<th>title</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2300</td>
<td>1010</td>
<td>1090</td>
</tr>
</tbody>
</table>

Estimate: \(1000 \times \frac{2300}{1000} \times \frac{1010}{1000} = 2323\)

- no longer exact, but very accurate in practice
Using Characteristic Sets

- characteristic sets accurately describe individual subjects
- but a query touches more than one subject
- combine characteristics sets to form whole queries

General strategy:
- exploit as much information about correlation as possible
- ignore the joins order ("holistic" estimates)
- avoids "fleeing to ignorance"
- cover the query with characteristic sets
Example

```sparql
select ?a ?t where {
  ?b <title>?t.  
  ?b <year>"2009".  
  ?p <name>"ACM".  
}
```

RDF query graph                       traditional query graph

- we cover the query with characteristic sets
Example

select ?a ?t where 
?b <publishedBy>?p. ?p <name>"ACM". }

RDF query graph traditional query graph

- we cover the query with characteristic sets
- prefer large sets over small sets

RDF query graph
traditional query graph

- we cover the query with characteristic sets
- prefer large sets over small sets
- assume independence for the rest
Challenges of SPARQL query optimization

Query Optimization:

Query Compilation \Rightarrow \text{Query Execution}

(dominated by query optimization)
Challenges of SPARQL query optimization

Query Optimization:

Query Compilation  \(\Rightarrow\)  Query Execution

(-dominated by query optimization)

<table>
<thead>
<tr>
<th>Tool</th>
<th>Compilation</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDF-3X</td>
<td>78 s</td>
<td>2 s</td>
</tr>
<tr>
<td>Virtuoso 7</td>
<td>1.3 s</td>
<td>384 s</td>
</tr>
</tbody>
</table>
Challenges of SPARQL query optimization

Query Optimization:

Query Compilation \(\Rightarrow\) Query Execution

(dominated by query optimization)

<table>
<thead>
<tr>
<th>System</th>
<th>Compilation Time</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDF-3X</td>
<td>78 s</td>
<td>2 s</td>
</tr>
<tr>
<td>Virtuoso 7</td>
<td>1.3 s</td>
<td>384 s</td>
</tr>
<tr>
<td>(next slides)</td>
<td>1.2 s</td>
<td>2 s</td>
</tr>
</tbody>
</table>

We ran a query with 17 joins on YAGO dataset (100 Mln triples)
Why does it happen?

Properties of the model:

- RDF is a very verbose format
- TPC-H Q5: 5 joins in SQL vs 26 joins in SPARQL (assuming a triple store storage)
- Dynamic Programming (RDF-3X) becomes too expensive

Properties of the data:

- Lots of correlations, including structural
- If an entity has a LastName, it is likely to have a FirstName
- Greedy Algorithm (Virtuoso) often makes wrong choices in the beginning
Combining Estimation and Optimization

Given a SPARQL query:

\[
\text{German\_novellist} \quad \text{type} \quad \text{wonPrize} \quad \text{bornIn} \quad \text{Nobel\_Prize} \quad \text{created} \quad \text{linksTo} \quad \text{hasLong} \quad \text{hasLat} \quad \text{hasLat} \quad \text{locatedIn} \quad \text{Italy} \\
\text{?p} \quad \text{?book} \quad \text{?city} \quad \text{?place} \quad \text{?long} \quad \text{?lat} \\
\]
Combining Estimation and Optimization

Given a SPARQL query:

\[
\text{German\_novellist} \quad \text{type} \quad ?p \quad \text{created} \quad ?book \quad \text{linksTo} \quad ?city \quad \text{locatedIn} \quad \text{Italy}
\]

\[
\text{Nobel\_Prize} \quad \text{wonPrize} \quad ?p \quad \text{bornIn} \quad \text{place} \quad \text{hasLong} \quad ?long \quad \text{hasLat} \quad ?lat
\]

- How to optimize star-shaped subqueries?
Combining Estimation and Optimization

Given a SPARQL query:

- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
Combining Estimation and Optimization

Given a SPARQL query:

```sparql
?p German_novellist
\^type
wonPrize
Nobel_Prize
\^bornIn
?place
?p created ?book
\^linksTo
?city
\^locatedIn
Italy
?book hasLong ?long
?book hasLat ?lat
?long
\^hasLong
\^hasLat
?city
\^hasLong
\^hasLat
?p
\^created
?book
\^linksTo
?city
\^locatedIn
Italy
?long
\^hasLong
\^hasLat
?lat
```

- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
- How to optimize arbitrary-shaped queries?
Optimizing star-shaped subqueries

- \{\text{type, livedIn, bornIn, created}\} \rightarrow 1025 \text{ entities}
- \text{Characteristic Set}
  - Count all distinct Char.Sets with number of occurrences
  - Accurate estimation of cardinalities of star-shaped queries
Optimizing star-shaped subqueries

- \{\text{type}, \text{livedIn}, \text{bornIn}, \text{created}\} \rightarrow 1025 \text{ entities}
- **Characteristic Set**
  - Count all distinct Char.Sets with number of occurrences
  - Accurate estimation of cardinalities of star-shaped queries
- One step beyond: what is the rarest subset of the given CS?
  - \{\text{type}, \text{livedIn}, \text{bornIn}\} \rightarrow 13304 \text{ entities}
  - \{\text{type}, \text{livedIn}, \text{created}\} \rightarrow 6593 \text{ entities}
  - \{\text{type}, \text{bornIn}, \text{created}\} \rightarrow 6800 \text{ entities}
  - \{\text{livedIn}, \text{bornIn}, \text{created}\} \rightarrow 2399 \text{ entities}
- \text{type} is not present in the rarest subset; we want to join it the last
Example

\{type, livedIn, bornIn, created\}, ID : 154
\{livedIn, bornIn, created\}, ID : 27
\{livedIn, created\}, ID : 6
Properties of the algorithm

- Linear time, top-down, greedy
- Does not assume independence between predicates (unlike bottom-up greedy)
Cardinality estimates in arbitrary queries

- How to estimate the cardinality of this query?
- Two subqueries depend on each other: every person is likely to have one birthplace in the data
- Just multiplying their frequencies is a big underestimation
How to estimate the cardinality of this query?

Two subqueries depend on each other: every person is likely to have one birthplace in the data.

Just multiplying their frequencies is a big underestimation.

We will construct a lightweight statistics of the dataset.

Count how frequently these two star-shaped subgraphs appear together.
Characteristic Pairs

- Characteristic Pair: Two Characteristic Sets that appear connected via an edge in the dataset
- Identifying CP: one scan over the data once the Char.Sets are computed
- In the worst case, the number of CP grows quadratically with different Char.Sets
- But we are only interested in very frequent ones
- If the pair is rare, the independence assumption holds
Char.Pairs: Estimating the cardinalities

```
select distinct ?s ?o
where {
  ?s p1 ?x1.
  ?s p2 ?x2.
  ?s p3 ?o.
}
```

- \( \{ S_i \} \leftarrow \text{Char.Sets with } \{ p_1, p_2, p_3 \} \)
- \( \{ S'_i \} \leftarrow \text{Char.Sets with } \{ p_4 \} \)
- Form all the Char.Pairs between \( \{ S_i \} \) and \( \{ S'_i \} \)
- Get their counts, sum up
Given a SPARQL query:

- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
Given a SPARQL query:

- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
- How to optimize arbitrary-shaped queries?
Query simplification

- We start with identifying optimal plans for subqueries
Query simplification

- We start with identifying optimal plans for subqueries
- Now, we remove them from the SPARQL query graph, and run the Dynamic Programming algo
Query simplification

- We start with identifying optimal plans for subqueries
- Now, we remove them from the SPARQL query graph, and run the Dynamic Programming algo
- We know the selectivities between the subqueries
Query simplification

\[
?P_1 \xrightarrow{\text{created}} s_1 \xrightarrow{\text{book}} \xrightarrow{\text{linksTo}} s_2 \xrightarrow{\text{?P}_2}
\]

<table>
<thead>
<tr>
<th>Entities</th>
<th>Partial Plan</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>{P_1}</td>
<td>(wonPrize &amp; type) &amp; bornIn</td>
<td>3000</td>
</tr>
<tr>
<td>{P_2}</td>
<td>(located&amp; hasLong) &amp; hasLat</td>
<td>5000</td>
</tr>
<tr>
<td>{book}</td>
<td>IndexScan(P = linksTo, S = ?book)</td>
<td>4500</td>
</tr>
<tr>
<td>{P_1, book}</td>
<td>((wonPrize &amp; type) &amp; bornIn) &amp; wrote</td>
<td>7500</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
## Compile and Runtime for YAGO

<table>
<thead>
<tr>
<th>Algo</th>
<th>Query Size (number of joins)</th>
<th>total runtime (optimization time)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[10, 20)</td>
<td>[20, 30)</td>
</tr>
<tr>
<td>DP</td>
<td>7745 (7130)</td>
<td>-</td>
</tr>
<tr>
<td>DP-CS</td>
<td>65767 (65223)</td>
<td>-</td>
</tr>
<tr>
<td>Greedy</td>
<td>857 (133)</td>
<td>1236 (413)</td>
</tr>
<tr>
<td>HSP</td>
<td>1025 (2)</td>
<td>3189 (3)</td>
</tr>
<tr>
<td>Char.Pairs</td>
<td><strong>660</strong> (150)</td>
<td><strong>967</strong> (315)</td>
</tr>
</tbody>
</table>
Other Challenges

- complex paths (transitivity etc.)
- complex aggregates
- updates
- transactions
- ...

Many hard problems, need careful analysis and tests.
Graph Data Processing is hard

- complex, not schema, correlations, etc.
- requires efficient storage and indexing
- query optimization is essential
- powerful techniques pay off very quickly

Many interesting problems still open.