Querying Graph-Structured Data

Thomas Neumann

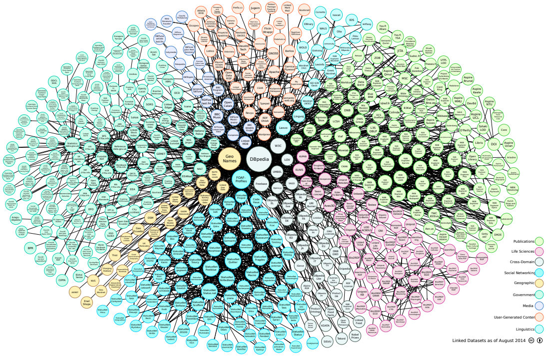
Technische Universität München

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.

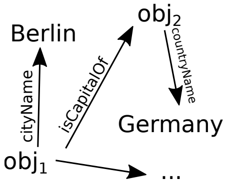
Graph-structured data

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:

- $<obj_1>$ $<cityName>$ 'Berlin'
- $<obj_1>$ $<isCapitalOf>$ $<obj_2>$
- $<obj_2>$ $<countryName>$ 'Germany'



Everything is encoded as triples, queries operate on triples.

SPARQL Protocol and RDF Query Language

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: $(obj_1, pred, ?x)$
$\Rightarrow$ Read range $(obj_1, pred, -\infty)$-$(obj_1, pred, \infty)$ in B^+-tree

Which sort order to choose?

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

e.g. `?x <cityName> ?capital.?x <isCapitalOf> ?y.` $\Rightarrow$
$(cityName, ?x, ?capital)_{PSO} \bowtie (isCapitolOf, ?x, ?y)_{PSO}$

Runtime Improvements

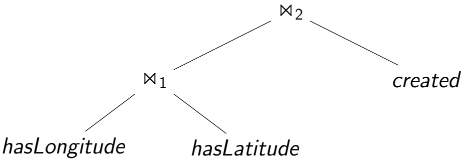
RDF-3X uses many techniques to improve runtime performance:

- 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
- ...

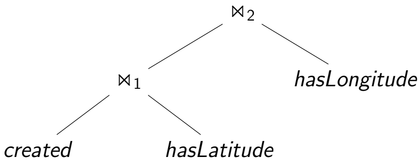
Optimize performance and minimize disk I/O.

Indexing is Not Enough

```
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:

$sel(\sigma_{P=\text{isCitizenOf}})$	$1.06 * 10^{-4}$
$sel(\sigma_{O=\text{United\_States}})$	$6.41 * 10^{-4}$
$sel(\sigma_{P=\text{isCitizenOf} \wedge O=\text{United\_States}})$	$4.86 * 10^{-5}$
$sel(\sigma_{P=\text{isCitizenOf}}) * sel(\sigma_{\text{O=United\_States}})$	$6.80 * 10^{-8}$

- 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

$< o_1 >$ <title> "The Tree and I".
$< o_1 >$ <author> <R. Pecker>.
$< o_1 >$ <author> <D. Owl>.
$< o_1 >$ <year> "1996".

Why Not Sampling?

RDF is very unfriendly for sampling

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

Yago sample

<wikicategory_Wilderness_Areas_of_Illinois> rdfs:label "Wilderness Areas of Illinois" .
<Telephone_numbers_in_Cameroon> rdfs:label "\u002b237" .
<Washington_Park_Race_Track> rdfs:label "Washington Park" .
<Seth_R.J.J._High_School> rdfs:label "Sett R\u002eJ\u002eJ\u002e High School" .
<Tengasu> rdfs:label "Tengasu" .
<Immaculate_Heart_Academy> rdfs:label "Immaculate Heart Academy" .
<Sion,_Switzerland> rdfs:label "Sion\u002c Switzerland" .
<wordnet_heroism_104857738> rdfs:label "gallantry" .
<Khyber_Pakhtunkhwa> rdfs:label "Khyber\u002dPakhtunkhwa" .
<J%C3%A1nos_Pap> rdfs:label "Janos Pap" .
<wikicategory_Jan_Smuts> rdfs:label "Jan Smuts" .
...

Sample would have to be huge to be useful.

Capturing Correlations

We classify the tuples using *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

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

$$\mathcal{S}_C(R) := \{S_C(s) | \exists p, o : (s, p, o) \in R\}.$$

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>.

$$S_C(o_1) = \{title, author, year\}$$

$$S_C(o_2) = \{title, author, year\}$$

$$\mathcal{S}_C = \{\{title, author, year\}^2, \{hasName, bornIn\}^1\}$$

Estimating Distinct Subjects

We can use characteristic sets for cardinality estimation

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

cardinality: $\sum_{S \in \{S | S \in \mathcal{S}_C(R) \wedge \{author, title\} \subseteq S\}} count(S)$

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

Number of Characteristic Sets

	triples	characteristic sets
Yago	40,114,899	9,788
LibraryThing	36,203,751	6,834
UniProt	845,074,885	613

Occurrence Annotations

Without *distinct* we need occurrence annotations

distinct	$\lvert\{s \mid \exists p, o : (s, p, o) \in R \wedge S_C(s) = S\}\rvert$
count(p_1)	$\lvert\{(s, p_1, o) \mid (s, p_1, o) \in R \wedge S_C(s) = S\}\rvert$
count(p_2)	$\lvert\{(s, p_2, o) \mid (s, p_2, o) \in R \wedge S_C(s) = S\}\rvert$
...	...

Example

select ?a ?t where { ?e <author> ?a. ?e <title> ?t. }

distinct	author	title	year
1000	2300	1010	1090

Estimate: $1000 * \frac{2300}{1000} * \frac{1010}{1000} = 2323$

- no longer exact, but very accurate in practice

Using Characteristic Sets

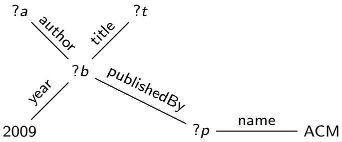
- characteristic sets accurately describe individual subjects
- but a query touches more than one subject
- combine characteristics sets to form whole queries

General strategy:

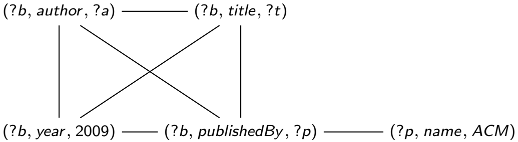
- exploit as much information about correlation as possible
- ignore the joins order ("holistic" estimates)
- avoids "fleeing to ignorance"
- *cover* the query with characteristic sets

Example

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



RDF query graph

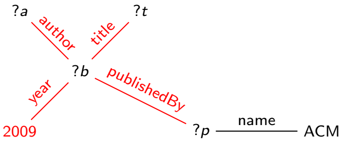


traditional query graph

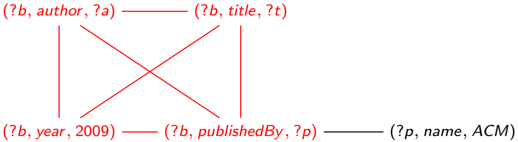
- we *cover* the query with characteristic sets

Example

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



RDF query graph

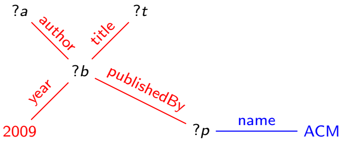


traditional query graph

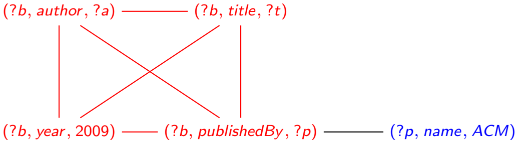
- we *cover* the query with characteristic sets
- prefer large sets over small sets

Example

select ?a ?t where { ?b <author>?a. ?b <title>?t. ?b <year>"2009". ?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
- assume independence for the rest

Challenges of SPARQL query optimization

Query Optimization:

Query Compilation $\Rightarrow$ Query Execution
(dominated by query optimization)

Challenges of SPARQL query optimization

Query Optimization:

	Query Compilation (dominated by query optimization)	$\Rightarrow$	Query Execution
RDF-3X	**78** s		2 s
Virtuoso 7	1.3 s		**384** s

Challenges of SPARQL query optimization

Query Optimization:

	Query Compilation (dominated by query optimization)	$\Rightarrow$	Query Execution
RDF-3X	**78** s		2 s
Virtuoso 7	1.3 s		**384** s
(next slides)	1.2 s		2 s

We ran a query with 17 joins on YAGO dataset (100 Mln triples)

Why does it happen?

Properties of the model:

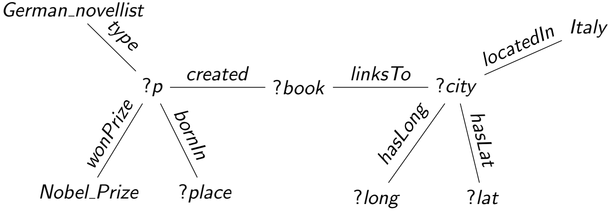
- RDF is a very verbose format
- TPC-H Q5: 5 joins in SQL vs 26 joins in SPARQL (assuming a triple store storage)
- Dynamic Programming (RDF-3X) becomes too expensive

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

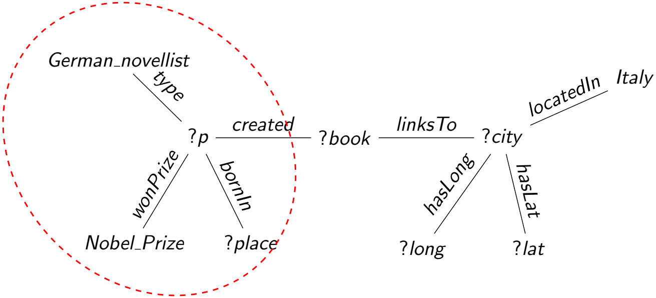
Combining Estimation and Optimization

Given a SPARQL query:



Combining Estimation and Optimization

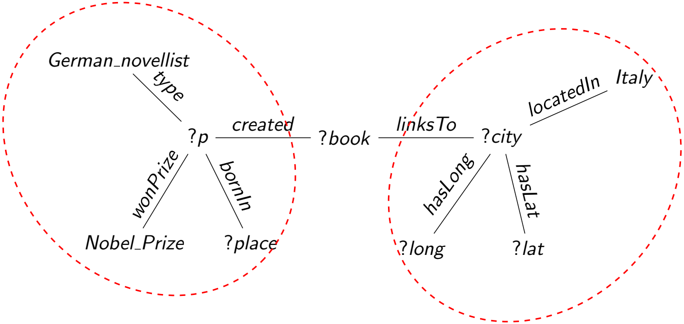
Given a SPARQL query:



- How to optimize star-shaped subqueries?

Combining Estimation and Optimization

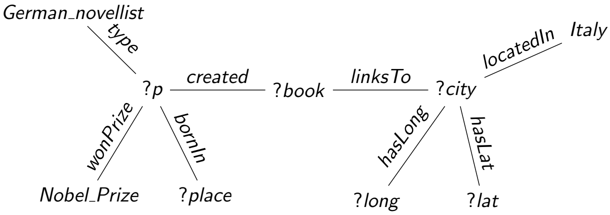
Given a SPARQL query:



- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?

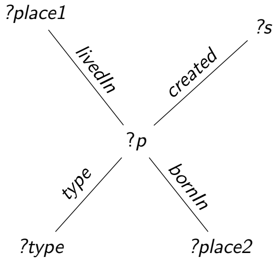
Combining Estimation and Optimization

Given a SPARQL query:



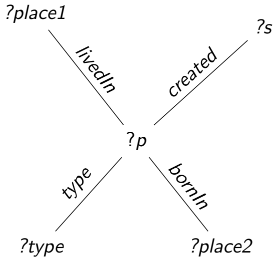
- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
- How to optimize arbitrary-shaped queries?

Optimizing star-shaped subqueries



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

Optimizing star-shaped subqueries



- $\{type, livedIn, bornIn, created\} \rightarrow$ 1025 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?
  - $\{type, livedIn, bornIn\} \rightarrow$ 13304 entities
  - $\{type, livedIn, created\} \rightarrow$ 6593 entities
  - $\{type, bornIn, created\} \rightarrow$ 6800 entities
  - $\{livedIn, bornIn, created\} \rightarrow$ **2399** entities
- *type* is not present in the rarest subset; we want to join it the last

Example

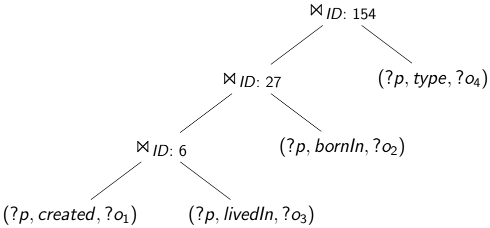
$$\{type, livedIn, bornIn, created\}, ID : 154$$

$$|$$

$$\{livedIn, bornIn, created\}, ID : 27$$

$$|$$

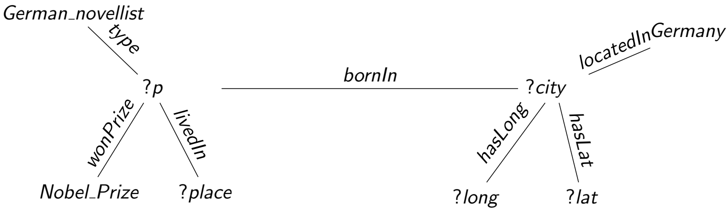
$$\{livedIn, created\}, ID : 6$$



Properties of the algorithm

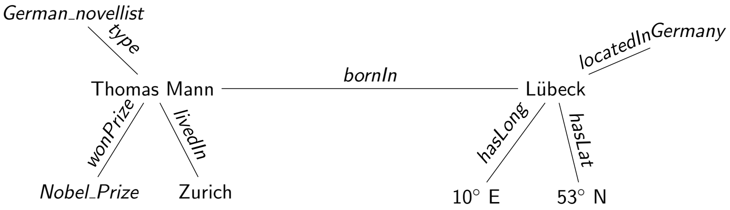
- Linear time, top-down, greedy
- Does not assume independence between predicates (unlike bottom-up greedy)

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

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
- 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

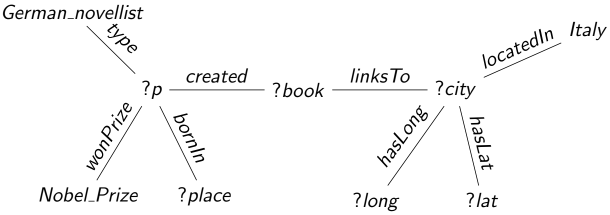
Char.Pairs: Estimating the cardinalities

select distinct ?s ?o
where { ?s p_1 ?x_1.
?s p_2 ?x_2.
?s p_3 ?o.
?o p_4 ?y_1. }

- $\{S_i\} \leftarrow$ Char.Sets with $\{p_1, p_2, p_3\}$
- $\{S_i'\} \leftarrow$ Char.Sets with $\{p_4\}$
- Form all the Char.Pairs between $\{S_i\}$ and $\{S_i'\}$
- Get their counts, sum up

Outline

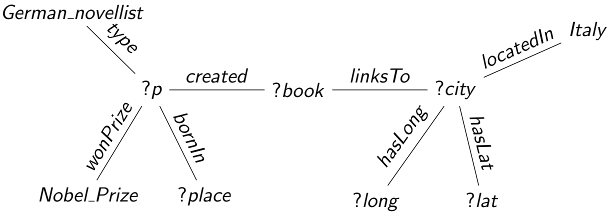
Given a SPARQL query:



- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?

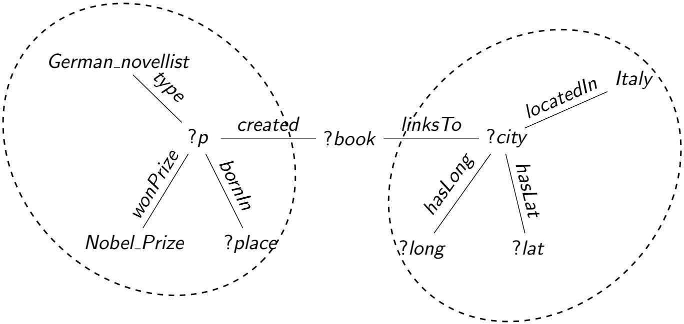
Outline

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

$$?P_1 \;\underline{\quad \textit{created} \quad}\; ?\textit{book} \;\underline{\quad \textit{linksTo} \quad}\; ?P_2$$

- 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

$$?P_1 \xrightarrow[s_1]{created} ?book \xrightarrow[s_2]{linksTo} ?P_2$$

- 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

$$?P_1 \; \frac{\textit{created}}{s_1} \; ?\textit{book} \; \frac{\textit{linksTo}}{s_2} \; ?P_2$$

Entities	Partial Plan	Cost
$\{P_1\}$	$(\textit{wonPrize} \bowtie \textit{type}) \bowtie \textit{bornIn}$	3000
$\{P_2\}$	$(\textit{locatedIn} \bowtie \textit{hasLong}) \bowtie \textit{hasLat}$	5000
$\{\textit{book}\}$	$\text{IndexScan}(P = \textit{linksTo}, S =?\textit{book})$	4500
$\{P_1, \textit{book}\}$	$((\textit{wonPrize} \bowtie \textit{type}) \bowtie \textit{bornIn}) \bowtie \textit{wrote}$	7500
...	...	...

Compile and Runtime for YAGO

Algo	Query Size (number of joins) total runtime (optimization time) $[10, 20)$	$[20, 30)$	$[30, 40)$	$[40, 50]$
DP	7745(7130)	-	-	-
DP-CS	65767(65223)	-	-	-
Greedy	857 (133)	1236 (413)	2204 (838)	4145 (1194)
HSP	1025 (**2**)	3189 (**3**)	4102 (**4**)	10720 (**5**)
Char.Pairs	**660** (150)	**967** (315)	**1211** (348)	**2174** (890)

Other Challenges

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

Many hard problems, need careful analysis and tests.

Conclusion

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.