Challenges in Graph Data Processing

Thomas Neumann

Technische Universität München

March 31, 2014

Motivation

Graph data is quite common, and people start using **graph databases** for graph/RDF data.

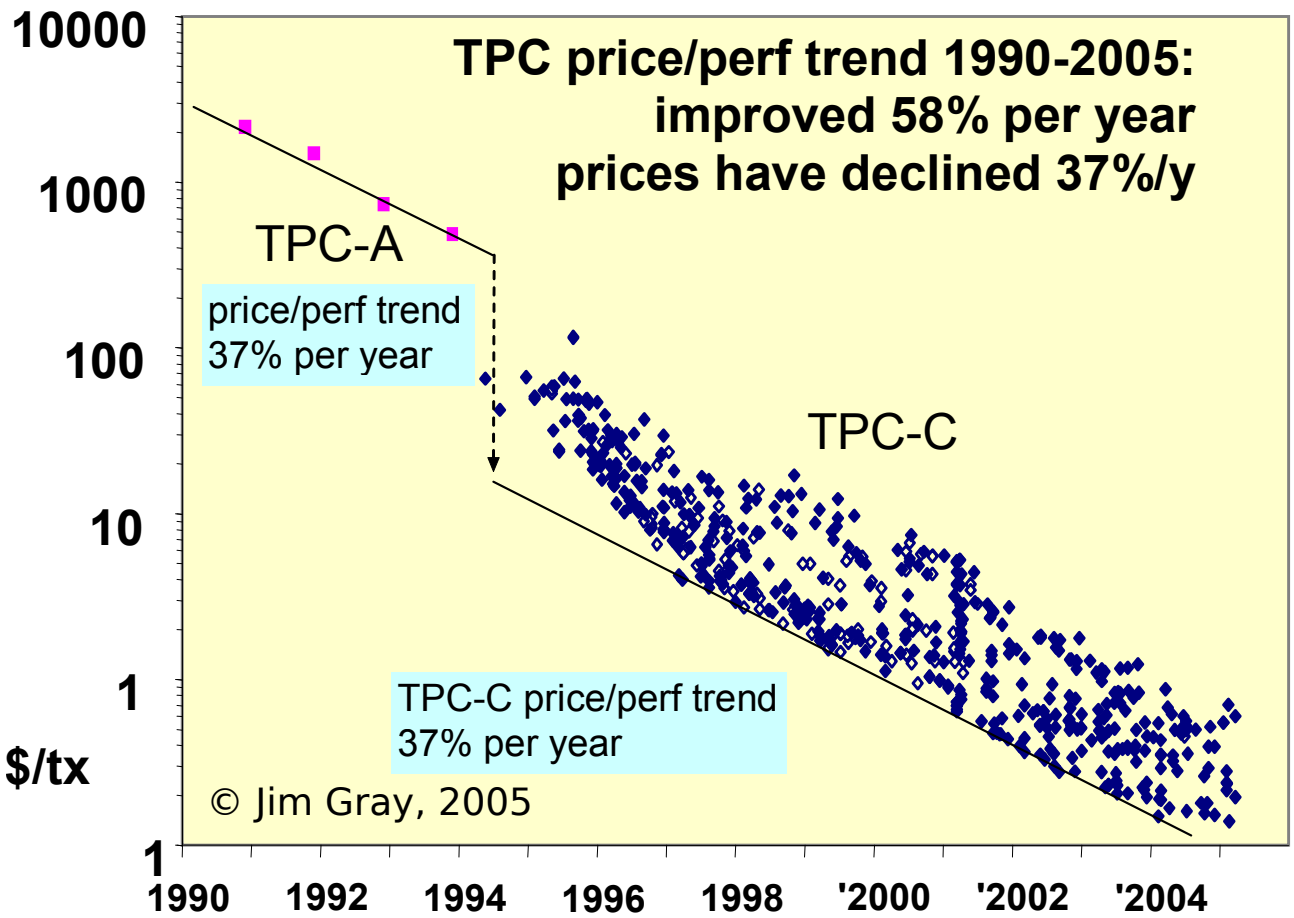
- which is a good idea
- use something existing instead of reinventing the wheel

but: are the **systems** good, too?

- is the system fast?
- is it correct?
- does it use sufficiently advances techniques?
- **hard to evaluate** in isolation
- whole field much less mature than, e.g., relation databases
- need for benchmarking and experiments

Note: This talk does not properly distinguish RDF and (property) graphs. Most examples are RDF-style, but problems are similar in both communities.

Why Benchmarking



- make competing products comparable
- accelerate progress, make technology viable

LDBC
Linked Data Benchmark Council

What is the LDBC?

Linked Data Benchmark Council = a **benchmarking organization** (`www.ldbc.eu`)

- Industry entity similar to TPC
- Focusing on graph and RDF store benchmarking

An **EU project** (STREP) in FP7

- Runs from sept 2012 – march 2015
- 8 project partners:
  Universitat Politècnica De Catalunya, Foundation for Research and Technology - Hellas, Neo Technology, OntoText AD, OpenLink Group Limited, Technische Universität München, Vrije Universiteit Amsterdam, Universität Innsbruck

EU Project Goals

- make sure the LDBC becomes a **strong entity** and will **continue to operate after the project**
- equip de LDBC with a good initial set of **benchmarks**, and benchmark results

Benchmark Task Forces

- Committee that works on a new benchmark
  - Technical Experts (choke point analysis)
  - TUC members (use cases)
- Benchmark Development Process
  - Specification, Implementation, Roll-Out

Task Force Activities

- Benchmark **Specification**
  - dataset selection (and/or data generator design),
  - workload
  - metrics
  - reporting format

  Benchmark **Implementation**
  - tool development
  - test evaluations (i.e.the running of the preliminary benchmark on a number of systems and an analysis of the results).

  Benchmark **Roll-out**
  - auditing guide + training the Auditors
  - producing the first reference results

Choke Points

- Choke Points are the pain points in current technology
  - insights from technology experts
  - ensure these pain points are part of benchmarks

  Choosing choke points well
  - Aim: stimulate innovation in key areas
  - Setting realistic goals

Examples of Choke Points

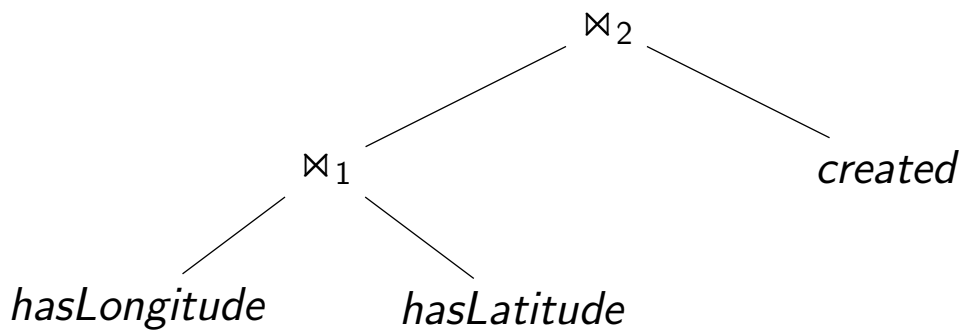
- Optimization strategies
  - Large search space
  - Simple greedy heuristics do not work
  - Non-inner joins (OPTIONALs in SPARQL)
- Real data
  - Correlations, skewed value distributions
  - Cardinality estimation becomes an issue even in simple queries

Following example numbers derived from YAGO2

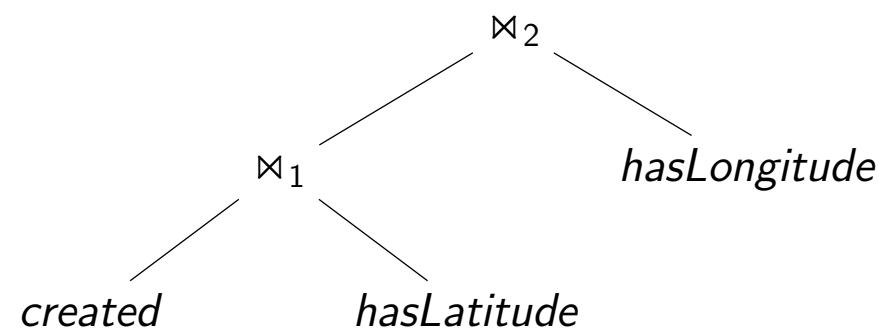
- 100 Mln triples
- 6 Gb when loaded into RDF-3X

Example: Cardinality estimation


```
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

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:

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

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

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

Queries with Bounded Objects

Characteristic sets handle bounded predicates. For bounded objects we use

$$sel(?o = o_1 | ?p = p_1) = \frac{sel(?o = o_1 \wedge ?p = p_1)}{sel(?p = p_1)}$$

- conditional selectivity
- when multiple objects are bound, consider the most selective
- others are assumed to be dependent (e.g., *title*)
- adjust occurrences to 1

Even better estimates with slightly more space:

- include number of distinct values in characteristic set
- conditional selectivity for the whole set

Handling Diverse Sets

- number of characteristic sets is usually very low
- but Billion Triples Challenge has 484,586 (union of data sets)
- becomes unwieldy

There, most sets occur very rarely. Individual impact is low.

- we can *merge* S_1 into S_2 if $S_1 \subseteq S_2$
- we can *split* S into S_1 and S_2

Introduces only a modest error, reduces as much as needed (e.g., 10,000).

Example

$S_1 = \{(author, 120), 100\}$, $S_2 = \{(title, 230), 200\}$
$S_3 = \{(author, 2300), (title, 1001), (year, 1000), 1000\}$
$S_4 = \{(author, 30), (title, 20), 20\}$
merge S_4 into S_3: $S_3 = \{(author, 2330), (title, 1021), (year, 1000), 1020\}$
or split S_4: $S_1 = \{(author, 150, 120)\}, S_2 = \{(title, 250, 220)\}$

Using Characteristic Sets

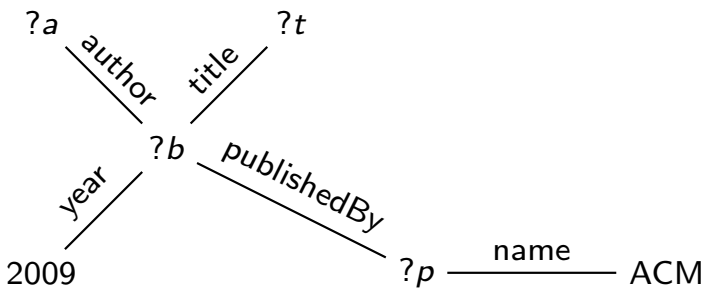
- characteristic sets accurately describe individual subjects
- but a query touches more than one subject
- combine characteristics sets to form whole queries

General strategy:

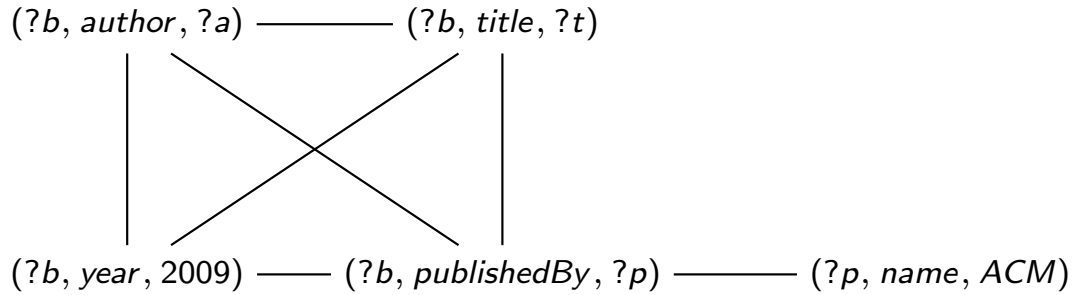
- exploit as much information about correlation as possible
- ignore the joins order ("holistic" estimates)
- avoids "fleeing to ignorance"
- *cover* the query with characteristic sets

Example

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



RDF query graph

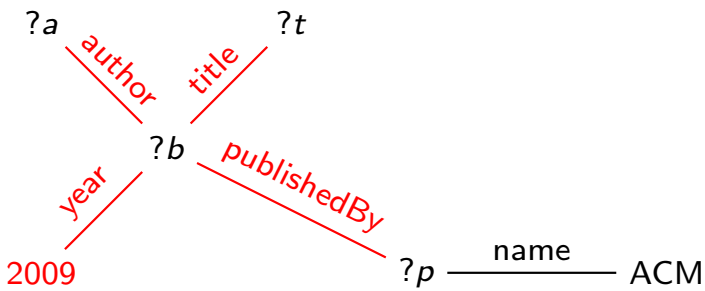


traditional query graph

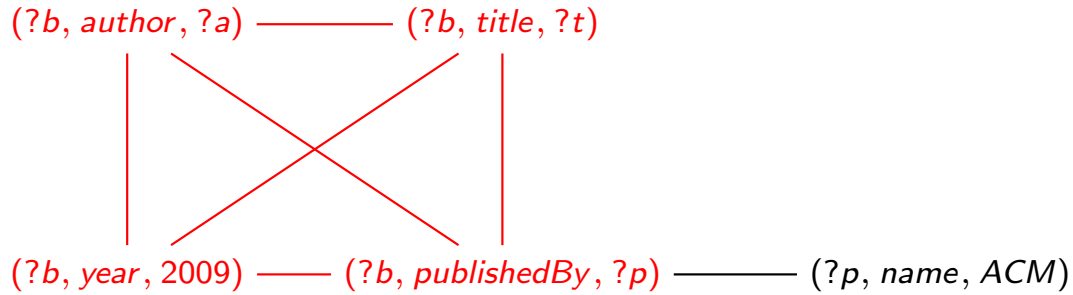
- we *cover* the query with characteristic sets

Example

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



RDF query graph

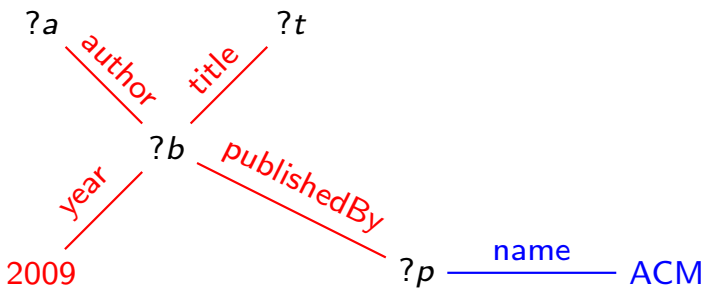


traditional query graph

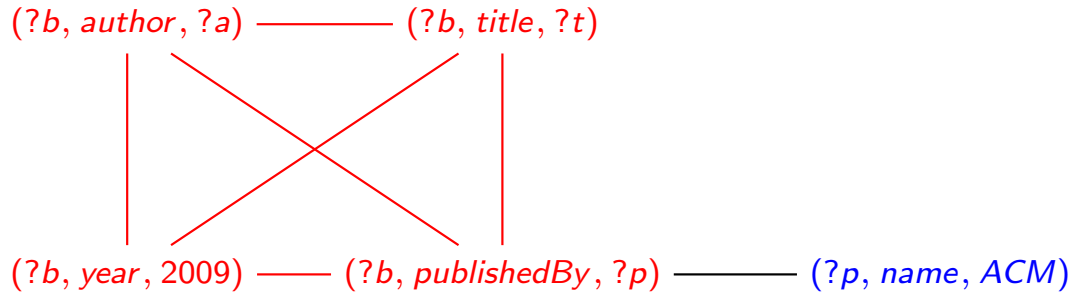
- we *cover* the query with characteristic sets
- prefer large sets over small sets

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

Evaluation

We compared different approaches:

- three large commercial DBMS (DB A, DB B, DB C)
- pair co-occurrence, Stocker et al. [WWW08]
- graph summarization, Maduko et al. [ESWC08]
- conditional selectivities, RDF-3X [VLDBJ10]
- characteristic sets

We consider two settings:

- single join
  - all predicate combinations
  - simple, but complete
- complex queries
  - up to six joins

Single Join Queries

Distribution of q-error ($\frac{\max(c,\hat{c})}{\min(c,\hat{c})}$, 1 is perfect)

	LibraryThing						
q-error	DB A	DB B	DB C	Stocker	Maduko	RDF-3X	CS
≤ 2	15.0	23.2	22.9	0	26.7	30.2	100
≤ 5	10.5	27.3	27.8	0	40.6	30.9	0
≤ 10	10.3	17.2	17.7	0	19.7	16.6	0
≤ 100	35.3	28.8	27.8	0.1	12.0	19.9	0
≤ 1000	20.7	3.1	3.3	0	0.8	2.2	0
> 1000	8.1	0.4	0.6	99.9	0.1	0.2	0
max	28,367,552	1,416,363	7,140,611	$1.3 * 10^{15}$	17,471	2,909,310	1.01

Computed by deriving the estimates for all queries of the form
?s <p_1> ?a. ?s <p_2> ?b.

Complex Queries

Cardinality estimates for complex queries: (absolute cardinalities)

	Yago							
query	exact	CS	RDF-3X	Stocker	Maduko	DB A	DB B	DB C
Q1	520,294	480,018	26,223	1	484,432	158	$4.3 * 10^8$	24,967
Q2	1,334	1,102	58	1	4	214	1	1
Q3	19,042	18,839	445	1	26,330	1	$1.0 * 10^7$	43,289
Q4	266	113	3	1	1	1	1	1
Q5	278,267	248,081	410	1	225,378	1	$1.9 * 10^{10}$	8,773
Q6	11,548	4,893	15	1	3	1	1	1
Q7	408	30	1	1	1	1	1	1
Q8	264	263	1	1	180	1	440	325
Q9	30	103	1	1	1	1	1	1
Q10	7	8	1	1	1	1	1	1

- most approaches severely underestimate (independence assumption)
- characteristic sets produce very accurate estimates

Challenges of SPARQL query optimization

Query Optimization:

Query Compilation ⇒ Query Execution
(dominated by query optimization)

Challenges of SPARQL query optimization

Technische Universität München

Query Optimization:

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

Challenges of SPARQL query optimization

Technische Universität München

Query Optimization:

	Query Compilation (dominated by query optimization)	⇒	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:

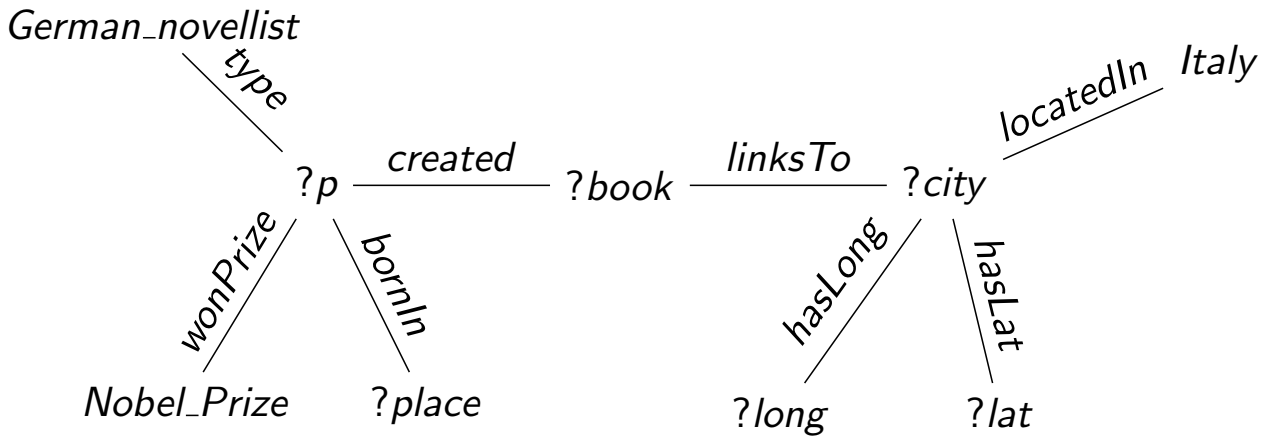
- RDF is a very verbose format
- TPC-H Q5: 5 joins in SQL vs 26 joins in SPARQL (assuming a triple store storage)
- Dynamic Programming (RDF-3X) becomes too expensive

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

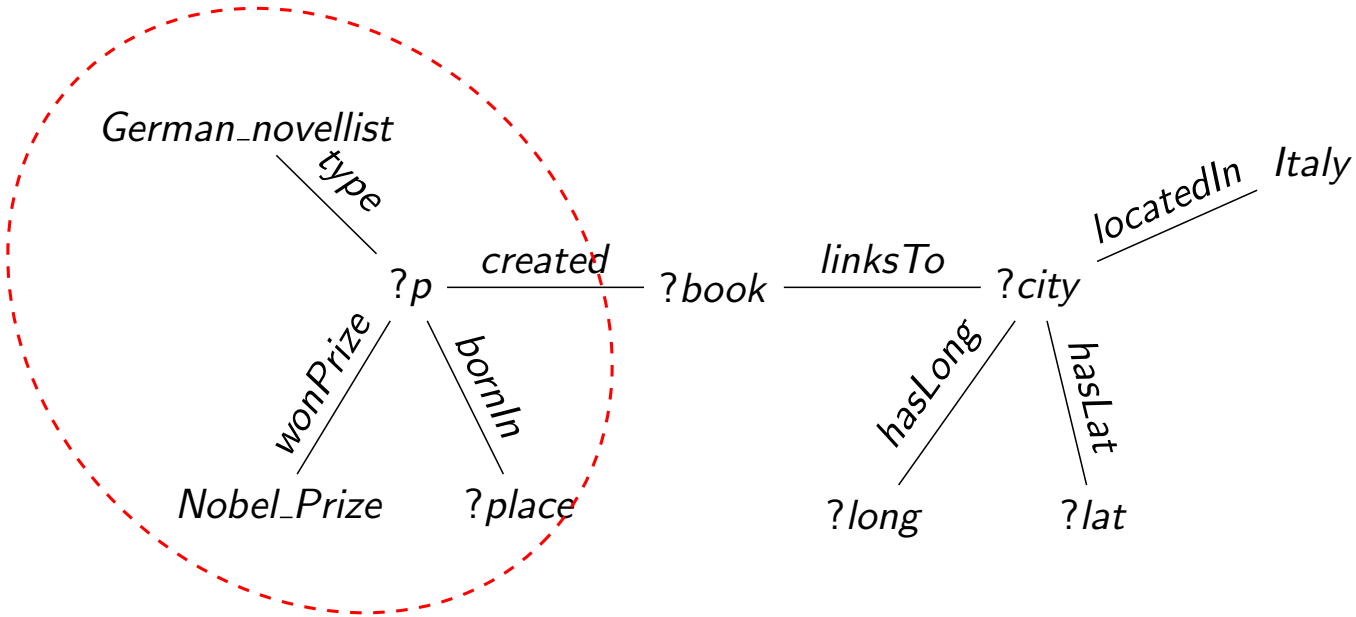
Combining Estimation and Optimization

Given a SPARQL query:



Combining Estimation and Optimization

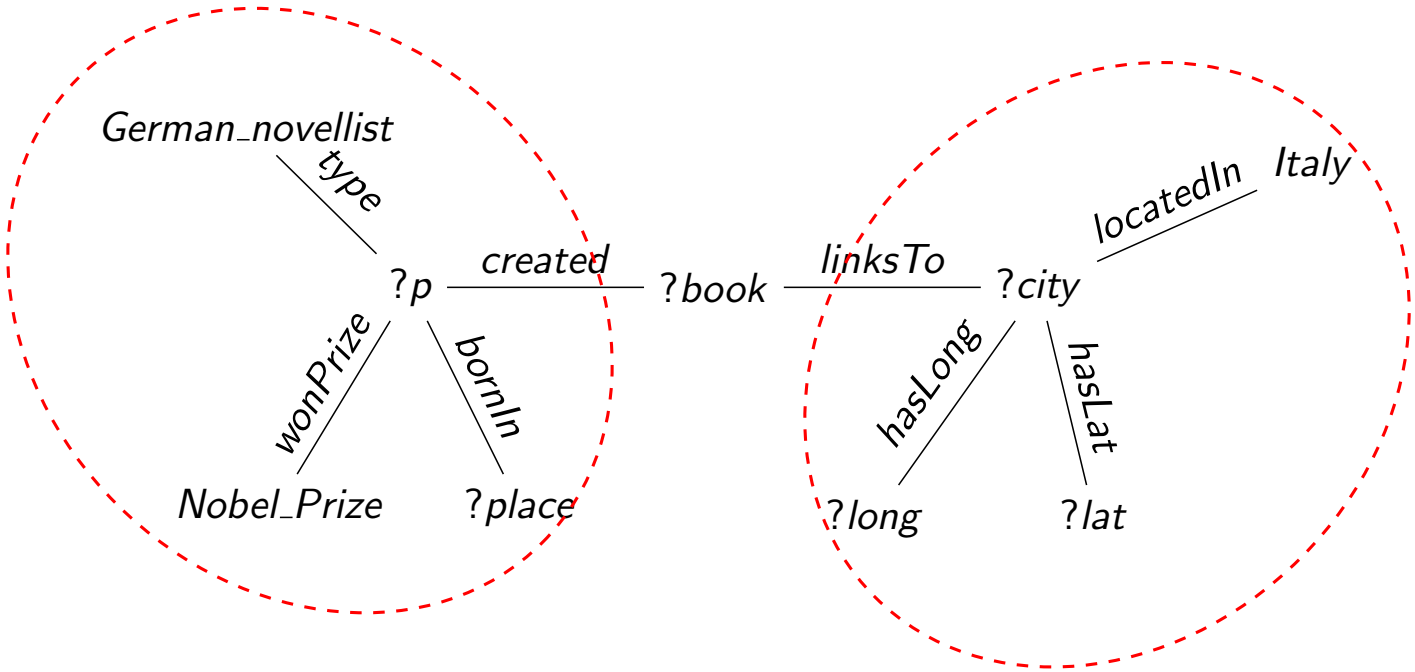
Given a SPARQL query:



- How to optimize star-shaped subqueries?

Combining Estimation and Optimization

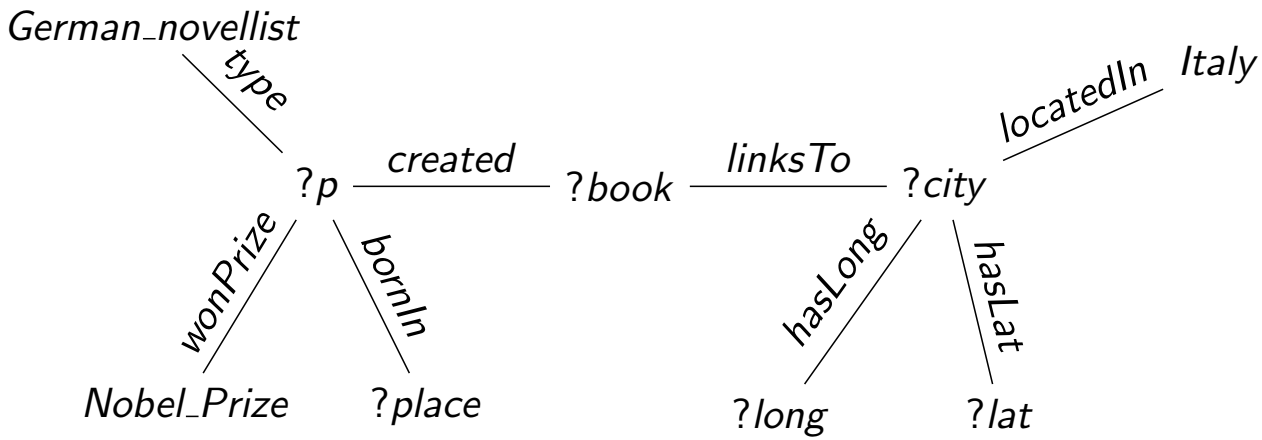
Given a SPARQL query:



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

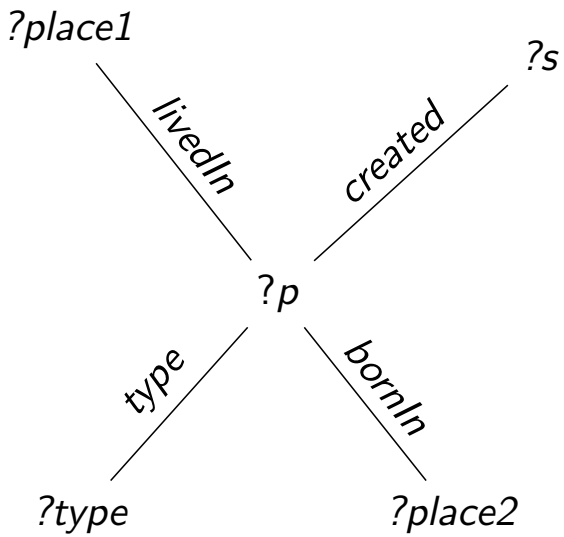
Combining Estimation and Optimization

Given a SPARQL query:



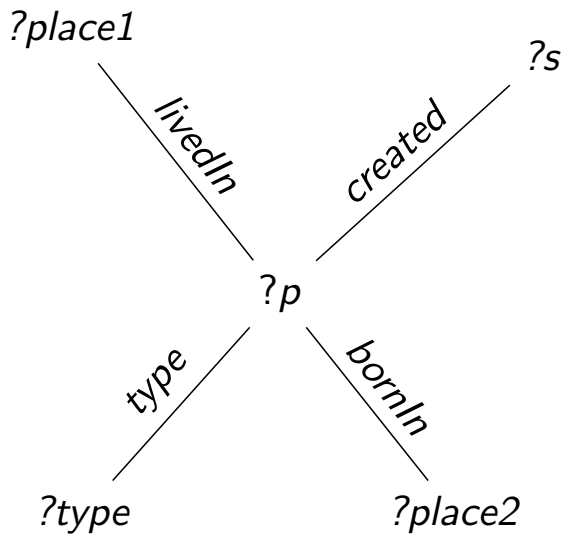
- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
- How to optimize arbitrary-shaped queries?

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

Optimizing star-shaped subqueries: Indexing

$\{type, livedIn, bornIn, created\}$

|

$\{livedIn, bornIn, created\}$

|

$\{livedIn, created\}$

- For every Char.Set keep its rarest subset
- Computing that could be expensive, but...
- Most of the CS in real data are already subsets of each other
- The total number of CS is very modest
  - less than 1000 for Uniprot
  - less than 10000 for YAGO

Optimizing star-shaped subqueries: Procedure

$\{type, livedIn, bornIn, created\}$

|

$\{livedIn, bornIn, created\}$

|

$\{livedIn, created\}$

1. $S \leftarrow$ the CS that corresponds to the subquery
2. $U \leftarrow$ the rarest subset of S
3. Get the difference $S \setminus U$, put it as a join
4. $S \leftarrow U$
5. If $S == \emptyset$ **return**; else goto 2.

Example

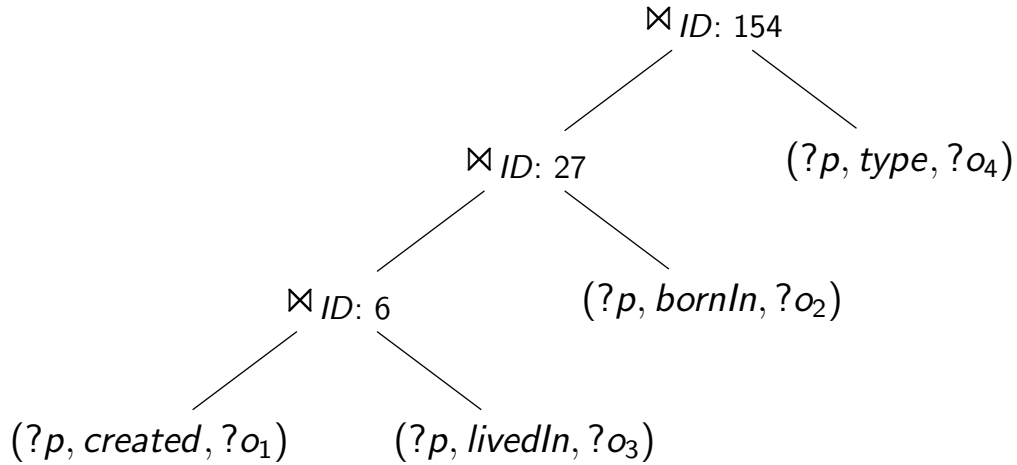
$\{type, livedIn, bornIn, created\}, ID : 154$

$|$

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

$|$

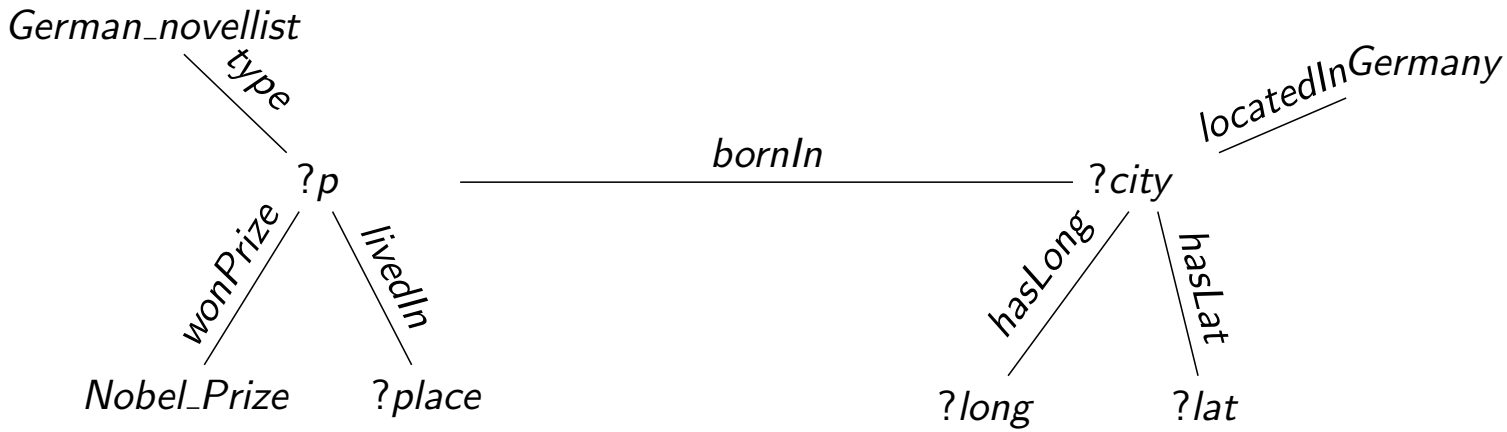
$\{livedIn, created\}, ID : 6$



Properties of the algorithm

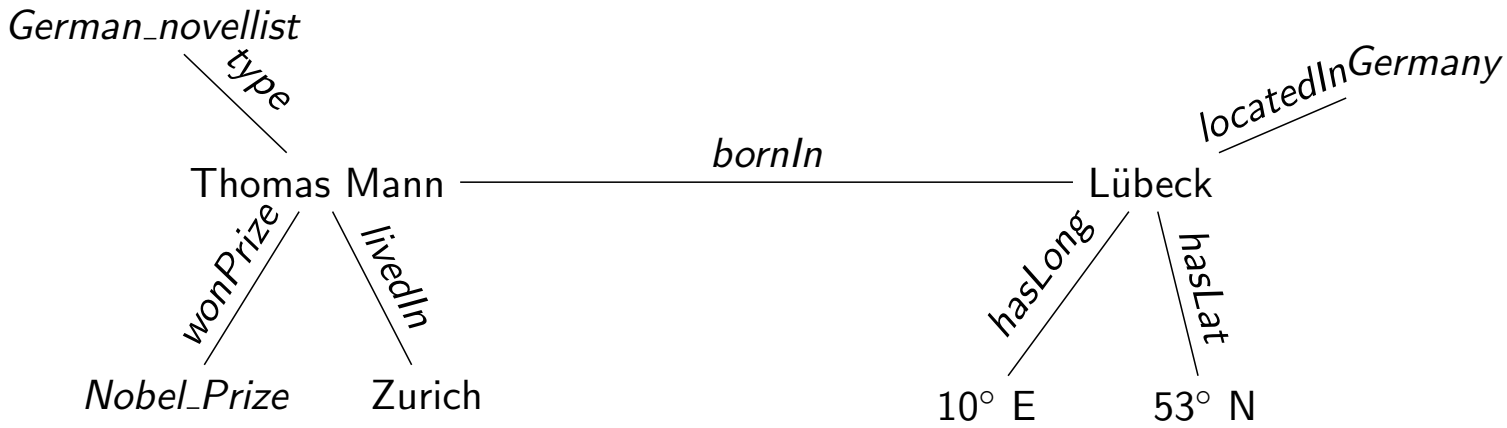
- Linear time, top-down, greedy
- Does not assume independence between predicates (unlike bottom-up greedy)

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

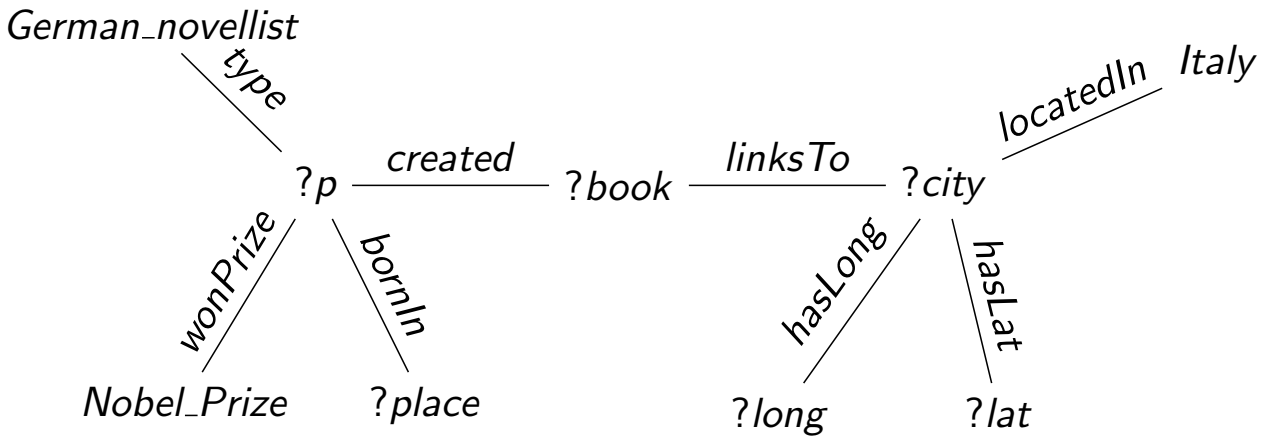
select distinct ?s ?o
where { ?s p_1 ?x_1.
?s p_2 ?x_2.
?s p_3 ?o.
?o p_4 ?y_1. }

- $\{S_i\} \leftarrow$ 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

Technische Universität München

Given a SPARQL query:

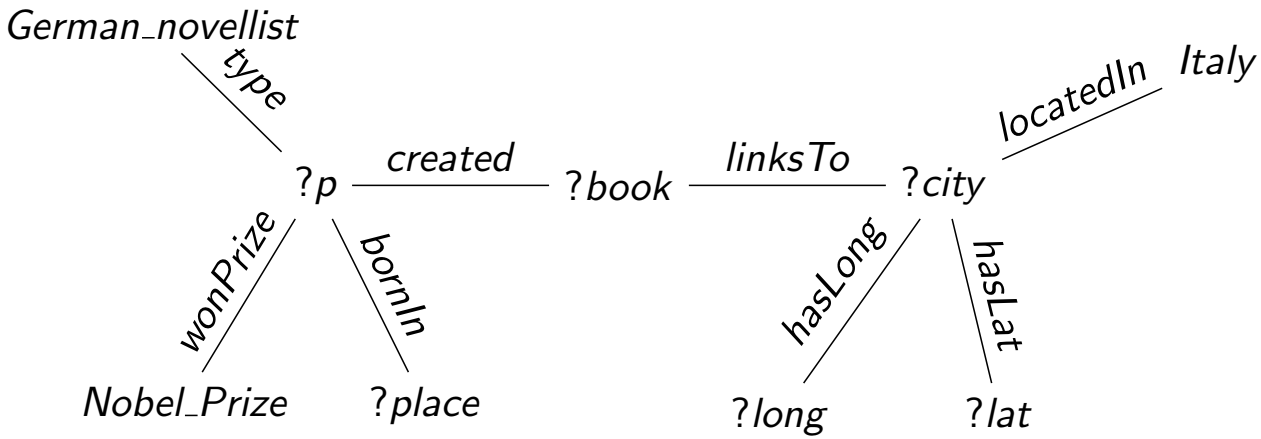


- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?

Outline

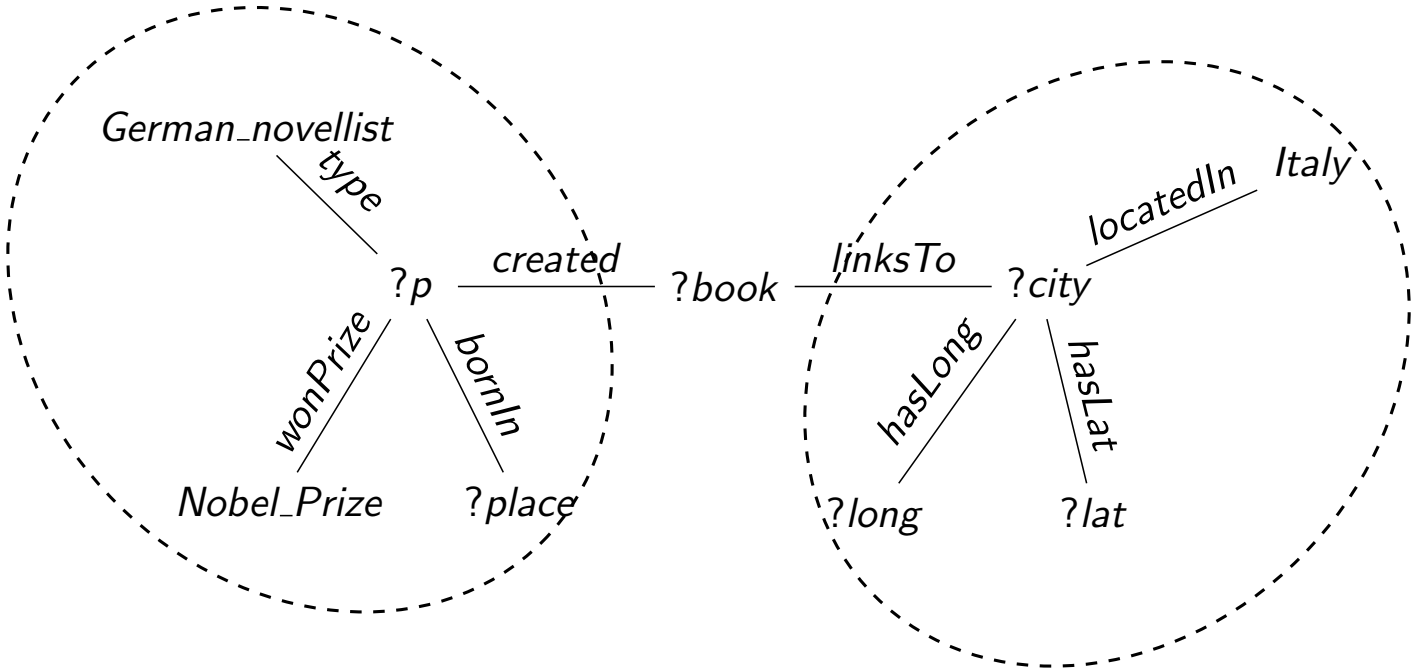
Technische Universität München

Given a SPARQL query:



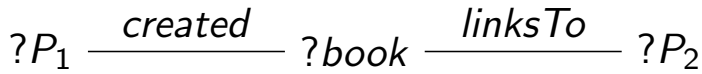
- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
- How to optimize arbitrary-shaped queries?

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

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

- 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

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

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

Applicability

- We have assumed the data is stored in a triple store
- Triple stores are industry standard
- Other approach: Predicate tables (and similar)
  - RDF@MonetDB, DB2
- Query simplification and Characteristic Pairs are applicable there as well

Experiments: Setup

- Three large datasets: YAGO, LibraryThing, Uniprot
- Generate random queries for them, varying size and structure
  - Pick central nodes in the graph
  - Try to connect them via paths
  - Add some star-shaped subqueries along the paths
- Compare with:
  - Greedy
  - DP
  - DP-CS: DP with Char.Sets (ICDE'11)
  - HSP: Heuristic SPARQL Planner (EDBT'12)
- Engine: RDF-3X
  - We implemented all the query optimization algorithms with the same backend (indexes, runtime techniques)

Precomputation: Time and Space

- Char.Sets (with subsets)
  - less than 5% of total loading time
  - less than 0.5% of total space
- Char.Pairs
  - less than 0.03% of total loading time
  - less than 0.01% of space

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.
- query optimization is essential

Needs Benchmarking to

- compare systems
- drive future development

LDBC targets that

- both an industry consortium and a (bootstrapping) EU project
- aims to become the standard graph benchmark council, similar to TPC

`www.ldbc.eu`