Exploiting the query structure for efficient join ordering in SPARQL

Andrey Gubichev Thomas Neumann

Technische Universität München

March 2014

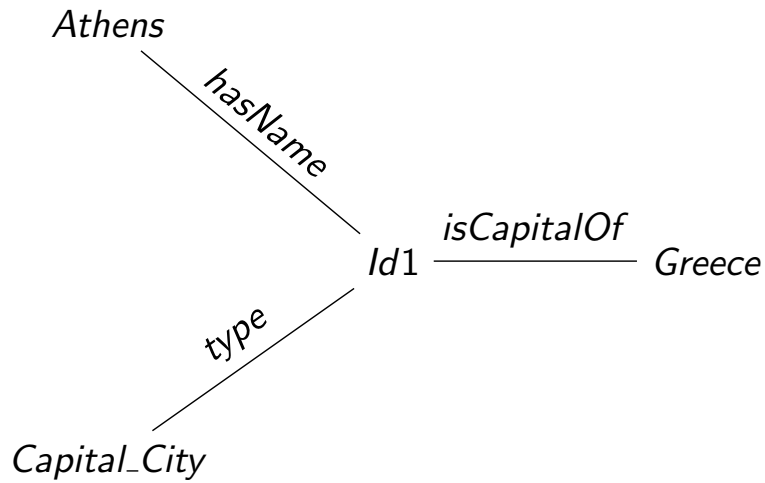
RDF and SPARQL

RDF is a graph-structured data model

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

Example:

- <Id1> <hasName> 'Athens'
- <Id1> <type> <Capital_City>
- <Id1> <isCapitalOf> <Greece>



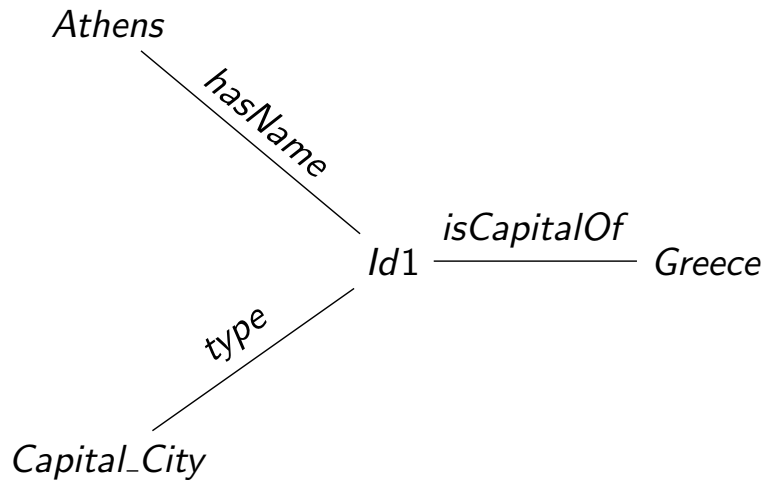
RDF and SPARQL

RDF is a graph-structured data model

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

Example:

- <Id1> <hasName> 'Athens'
- <Id1> <type> <Capital_City>
- <Id1> <isCapitalOf> <Greece>



SPARQL queries are sets of triple patterns to match against the graph

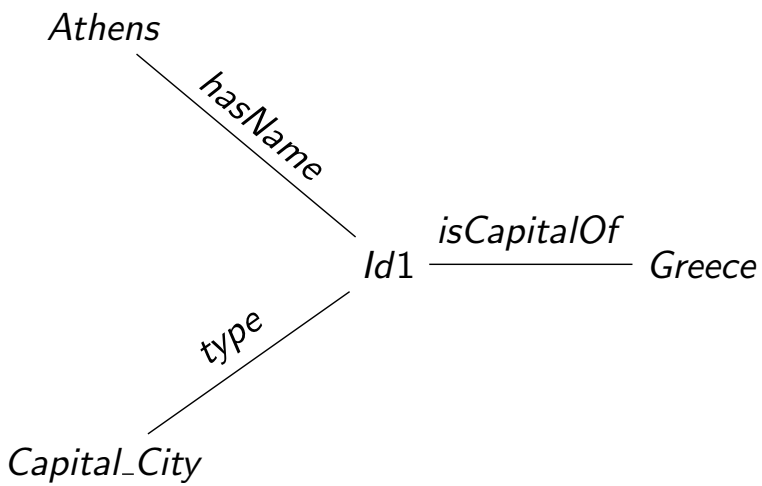
```
SELECT ?x WHERE { ?x <hasName> ?name.
?x <type> <Capital_City>.
?x <isCapitalOf> ?country. }
```

Evaluation strategy: find variable bindings for each pattern, and join them

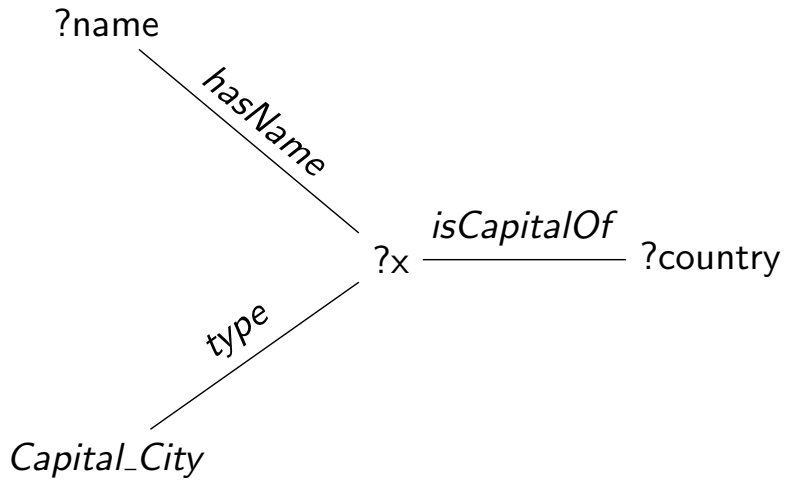
RDF Graph and SPARQL Query Graph

RDF Graph:

- <Id1> <hasName> 'Athens'
- <Id1> <type> <Capital_City>
- <Id1> <isCapitalOf> <Greece>

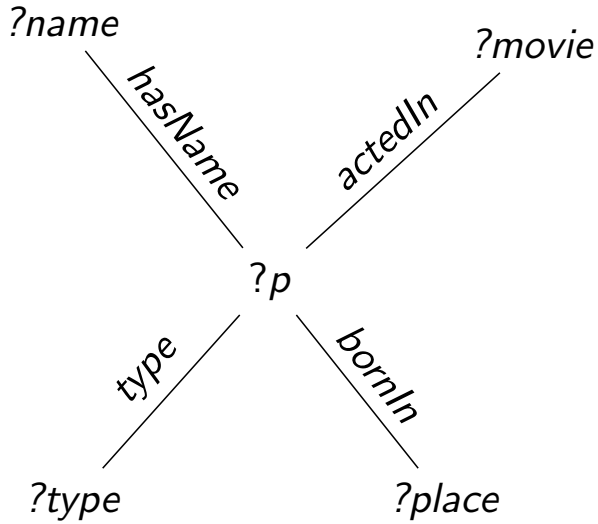



```
SELECT ?x WHERE {
?x <hasName> ?name.
?x <type> <Capital_City>.
?x <isCapitalOf> ?country. }
```

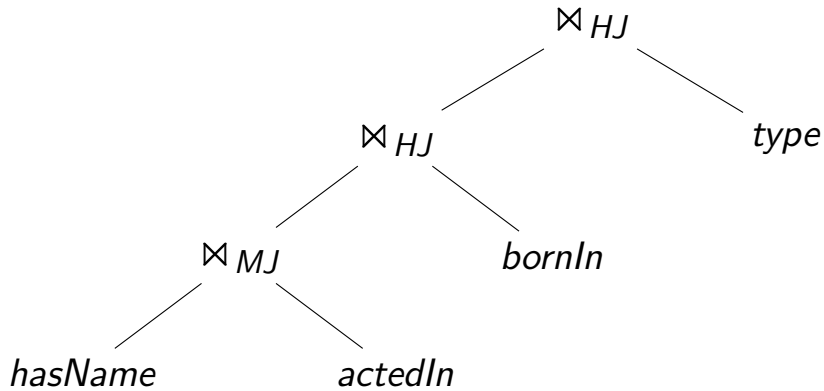


Query Optimization for SPARQL

Input: SPARQL Query Graph



Output: Join tree



Objective: minimize amount of intermediate results

Challenges of SPARQL query optimization

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

Query Processing:

Query Compilation ⇒ Query Execution
(dominated by query optimization)

Challenges of SPARQL query optimization

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

			Query Processing:
	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

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

	Query Processing:		
	Query Compilation (dominated by query optimization)	⇒	Query Execution
RDF-3X	78 s		2 s
Virtuoso 7	1.3 s		384 s
This work	1.2 s		2 s

Why does it happen?

Properties of the model:

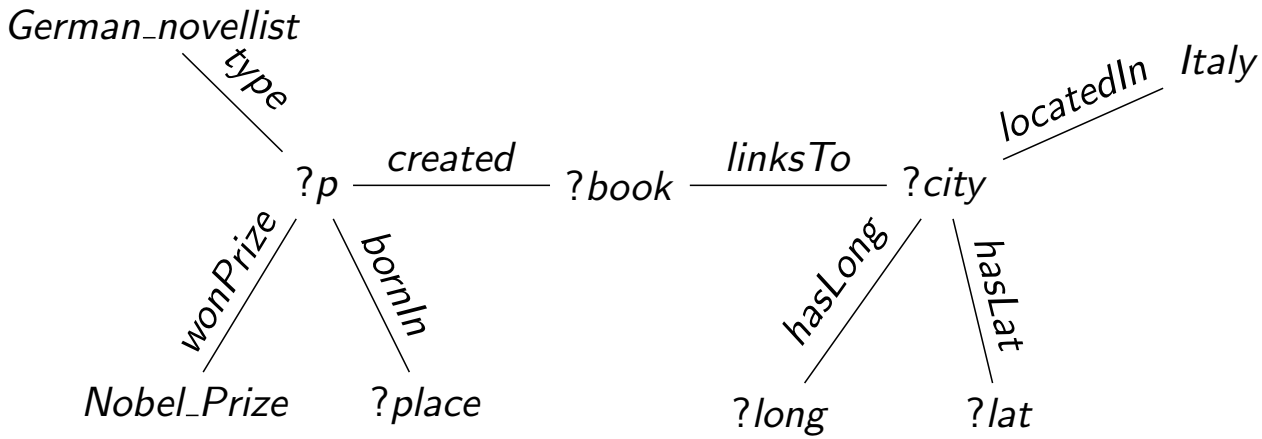
- RDF is a very verbose format
- TPC-H Q5: 6 joins in SQL vs 26 joins in SPARQL (assuming a triple store storage)
- Dynamic Programming (RDF-3X) becomes too expensive

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

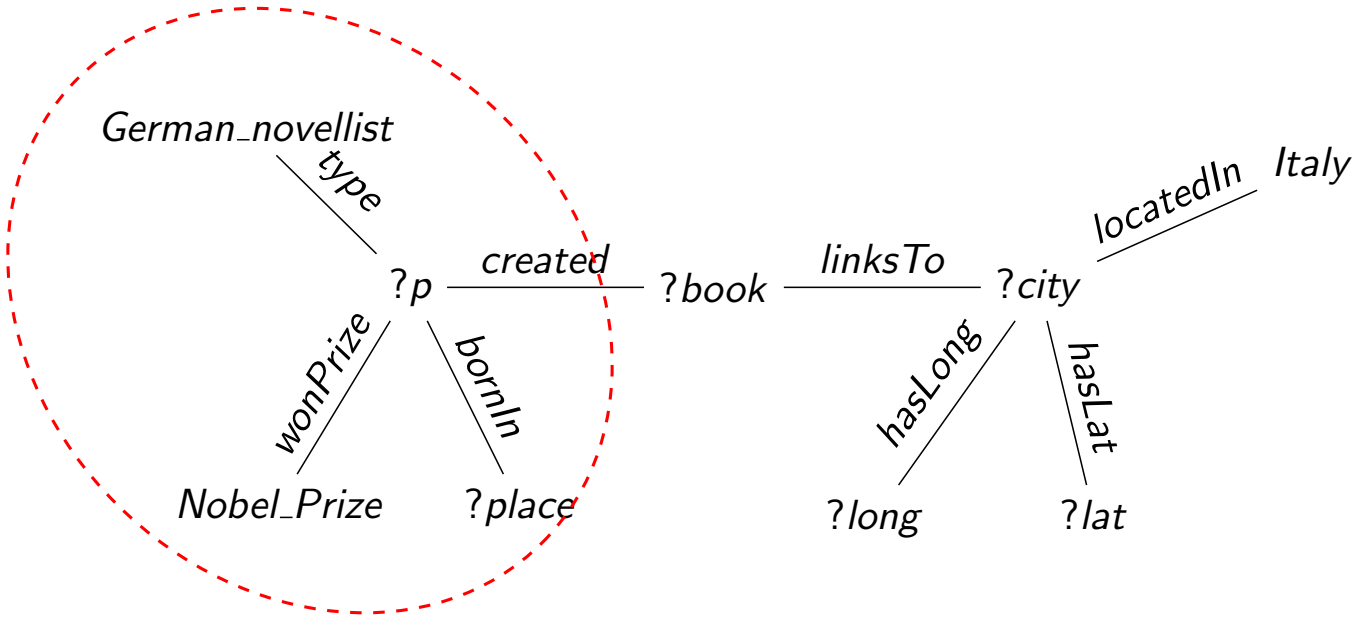
The goal of this work

Given a SPARQL query:



The goal of this work

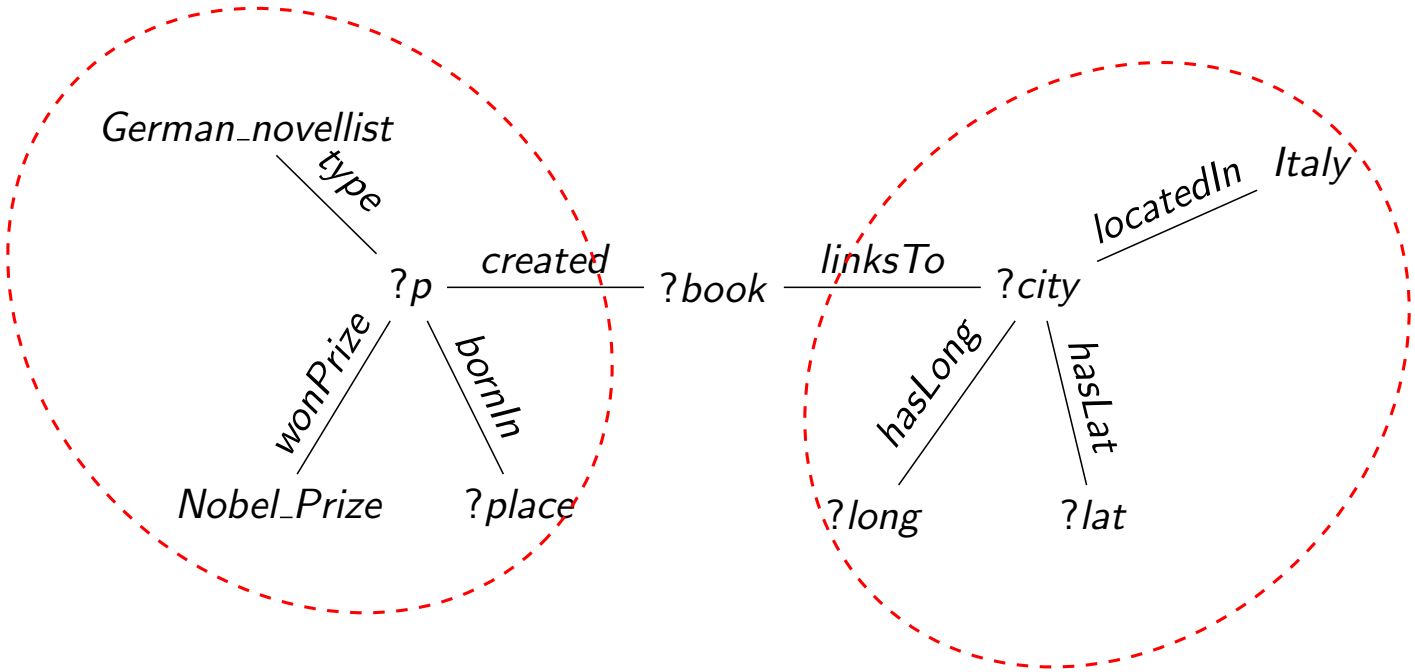
Given a SPARQL query:



- How to optimize star-shaped subqueries?

The goal of this work

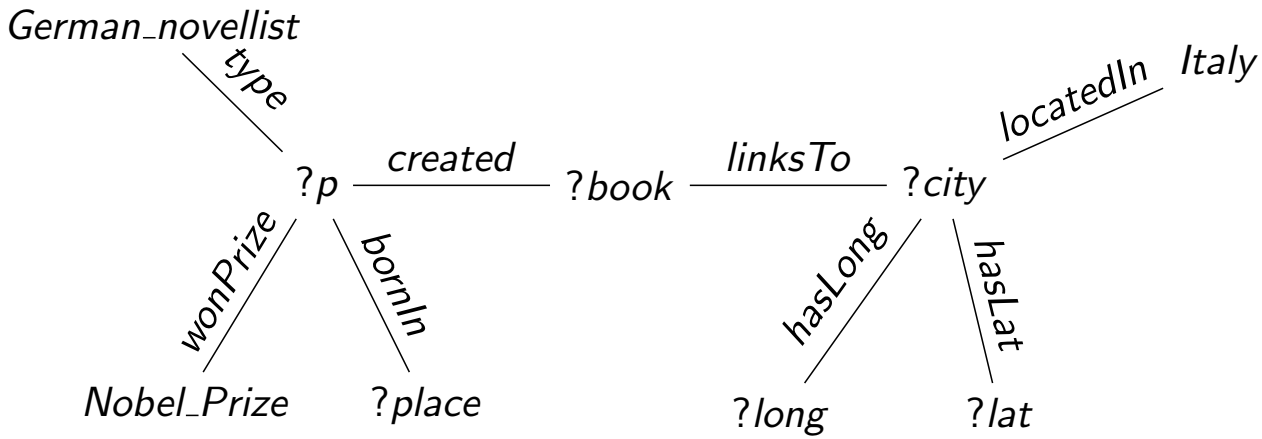
Given a SPARQL query:



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

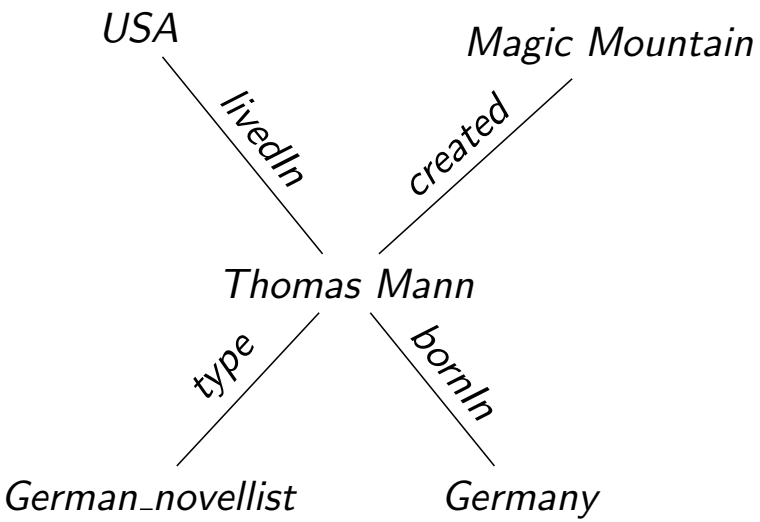
The goal of this work

Given a SPARQL query:



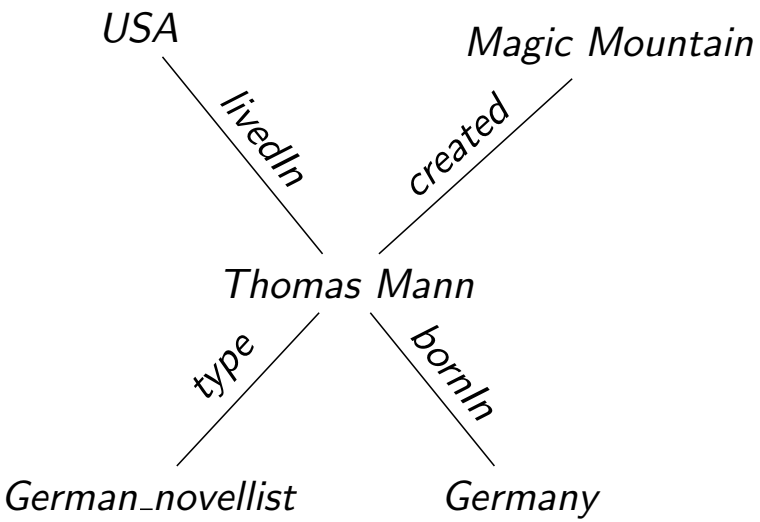
- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
- How to optimize arbitrary-shaped queries?

Related Work: Characteristic Sets (ICDE'11)



- Outgoing edges "characterise" the node
- $\{type, livedIn, bornIn, created\}$ – Characteristic Set of the node
- Different nodes may have the same CS (e.g., all writers)

Related Work: Characteristic Sets (ICDE'11)

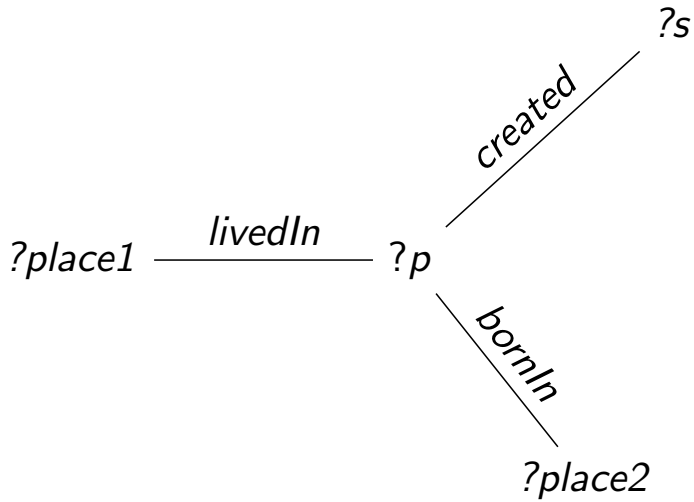


- Outgoing edges "characterise" the node
- $\{type, livedIn, bornIn, created\}$ – Characteristic Set of the node
- Different nodes may have the same CS (e.g., all writers)
- We mine all the distinct CSs from the data, and count the nodes for each CS
- $\{type, livedIn, bornIn, created\} \rightarrow 1025$ nodes

Related Work: Characteristic Sets (ICDE'11)

Given all the CSs, how to estimate the cardinalities of star-shaped queries?

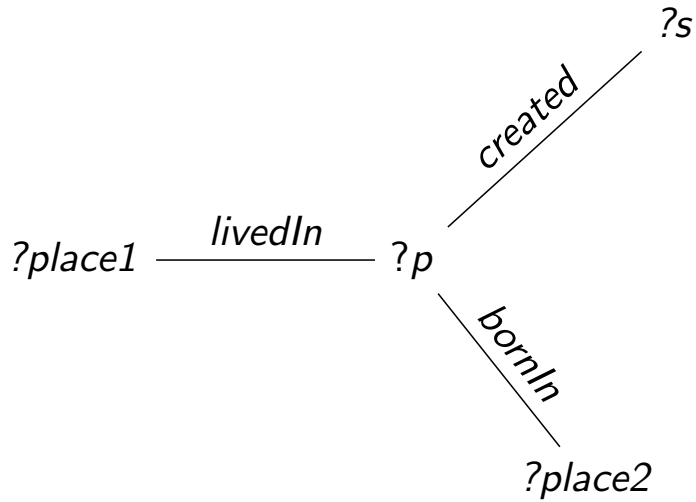
Char.Set	Count
{*livedIn*, *bornIn*, *created*}	2399
{*type*, *livedIn*, *bornIn*, *created*}	1025
...	...



Related Work: Characteristic Sets (ICDE'11)

Given all the CSs, how to estimate the cardinalities of star-shaped queries?

Char.Set	Count
{*livedIn*, *bornIn*, *created*}	2399
{*type*, *livedIn*, *bornIn*, *created*}	1025
...	...

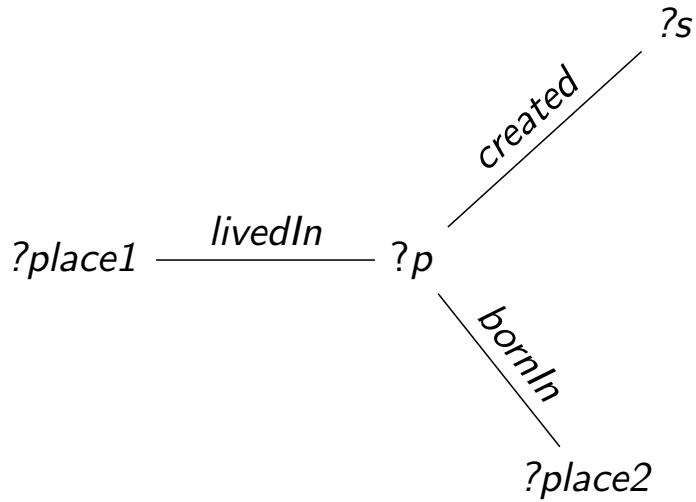


$$\begin{aligned}\text{Card} &= \text{Count}(\{\textit{livedIn}, \textit{bornIn}, \textit{created}\}) \\ &+ \text{Count}(\{\textit{type}, \textit{livedIn}, \textit{bornIn}, \textit{created}\}) \\ &+ \cdots \\ &= 2399 + 1025 + \cdots\end{aligned}$$

Related Work: Characteristic Sets (ICDE'11)

Given all the CSs, how to estimate the cardinalities of star-shaped queries?

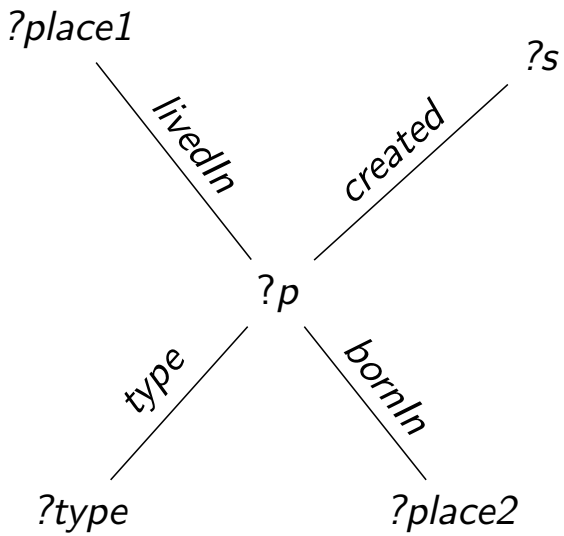
Char.Set	Count
$\{livedIn, bornIn, created\}$	2399
$\{type, livedIn, bornIn, created\}$	1025
$\dots$	$\dots$



$$\begin{aligned}\text{Card} &= \text{Count}(\{livedIn, bornIn, created\}) \\ &+ \text{Count}(\{type, livedIn, bornIn, created\}) \\ &+ \cdots \\ &= 2399 + 1025 + \cdots\end{aligned}$$

This estimation can be used in Dynamic Programming algorithm
The plan gets better (no independence assumption).
But compile time increases.

Optimizing star-shaped subqueries



- $\{type, livedIn, bornIn, created\} \rightarrow$ 1025 entities
- 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
  - <1000 for Uniprot (700 Mln triples)
  - <10000 for YAGO (100 Mln triples)

Optimizing star-shaped subqueries: Procedure

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

|

$\{livedIn, bornIn, created\}$

|

$\{livedIn, created\}$

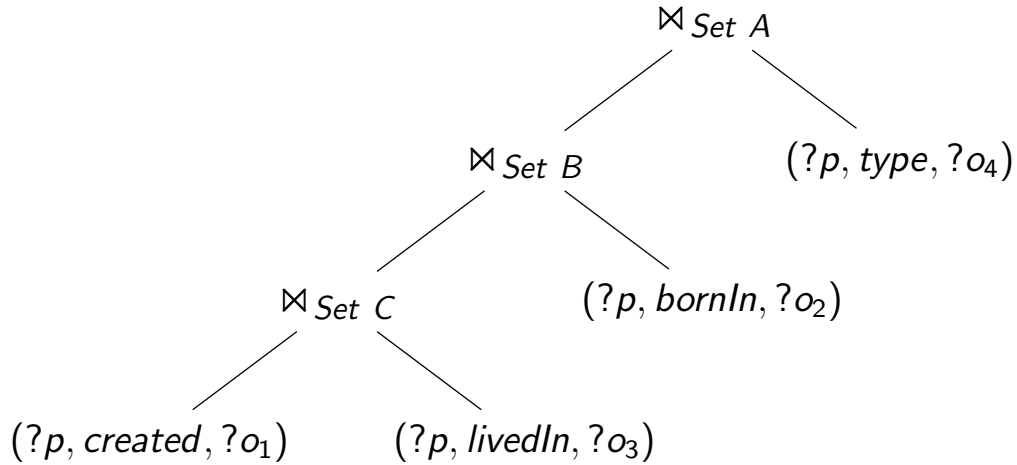
1. $S \leftarrow$ the CS that corresponds to the subquery
2. $U \leftarrow$ the rarest subset of S
3. Get the difference $S \setminus U$, put it as a join
4. $S \leftarrow U$
5. If $S == \emptyset$ **return**; else goto 2.

Example

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

$\{livedIn, bornIn, created\}, Set\ B$

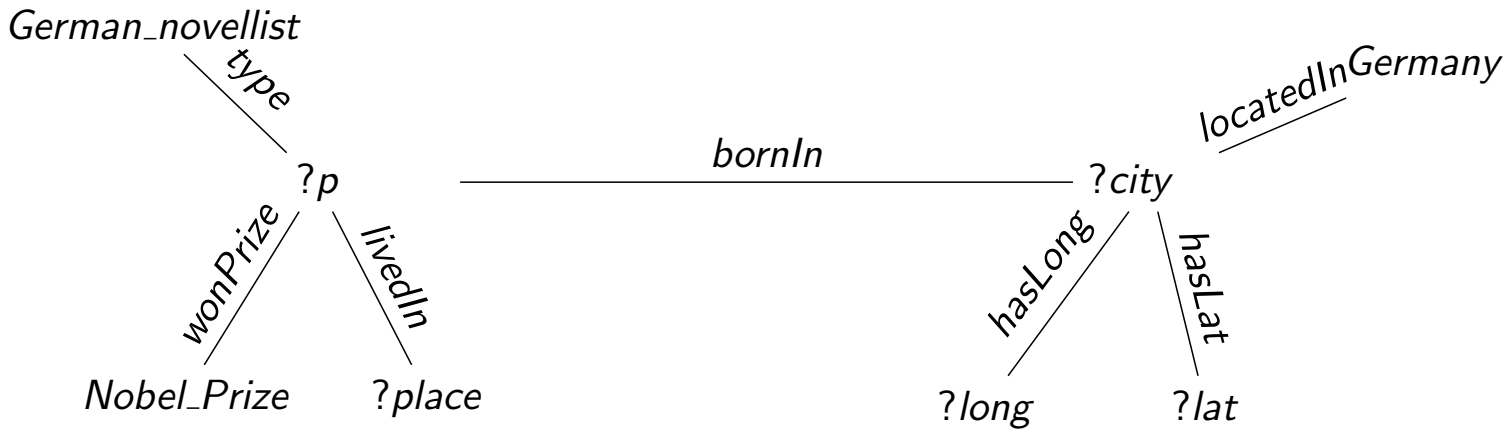
$\{livedIn, created\}, Set\ C$



Properties of the algorithm

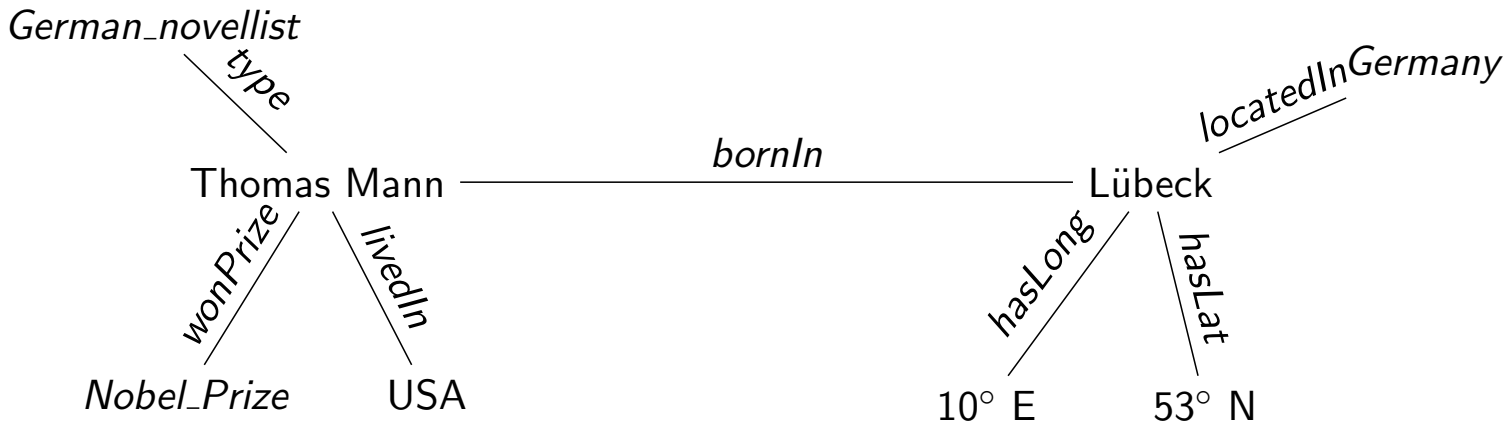
- Linear time, top-down, greedy
- Does not assume independence between predicates (unlike bottom-up greedy)

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



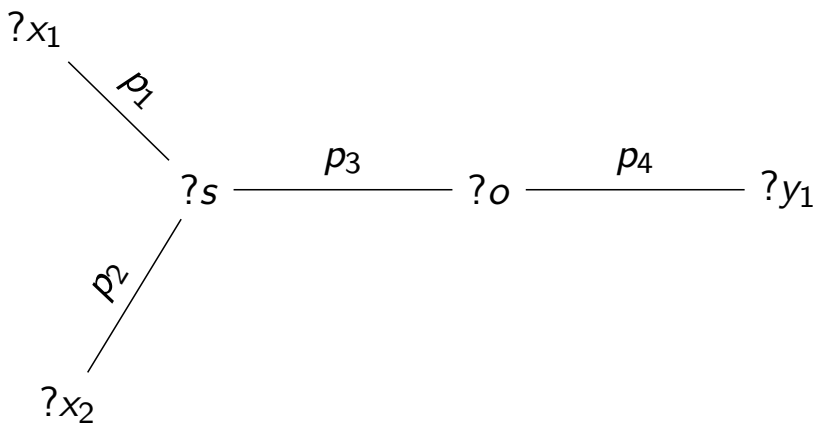
- How to estimate the cardinality of this query?
- Two subqueries depend on each other: every person is likely to have one birthplace in the data
- Just multiplying their frequencies is a big underestimation
- We will construct a lightweight statistics of the dataset
- Count how frequently these two 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

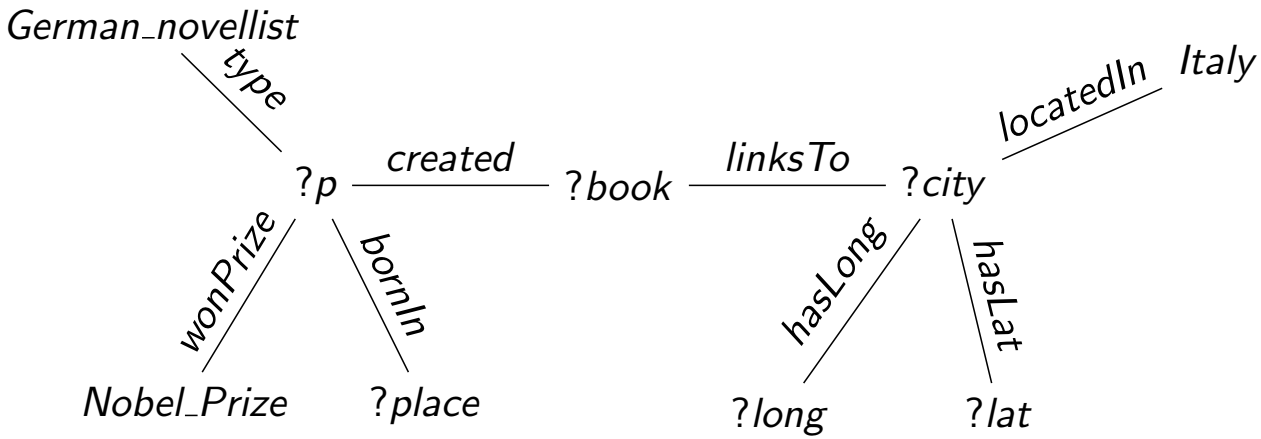
select distinct ?s ?o
where { ?s p_1 ?x_1.
?s p_2 ?x_2.
?s p_3 ?o.
?o p_4 ?y_1. }



- $\{S_i\} \leftarrow$ 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

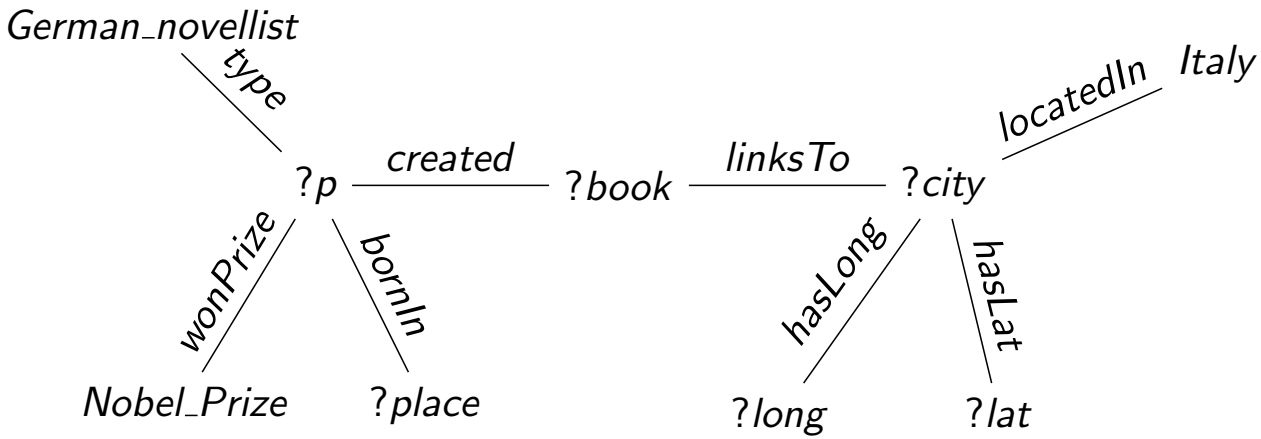
Given a SPARQL query:



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

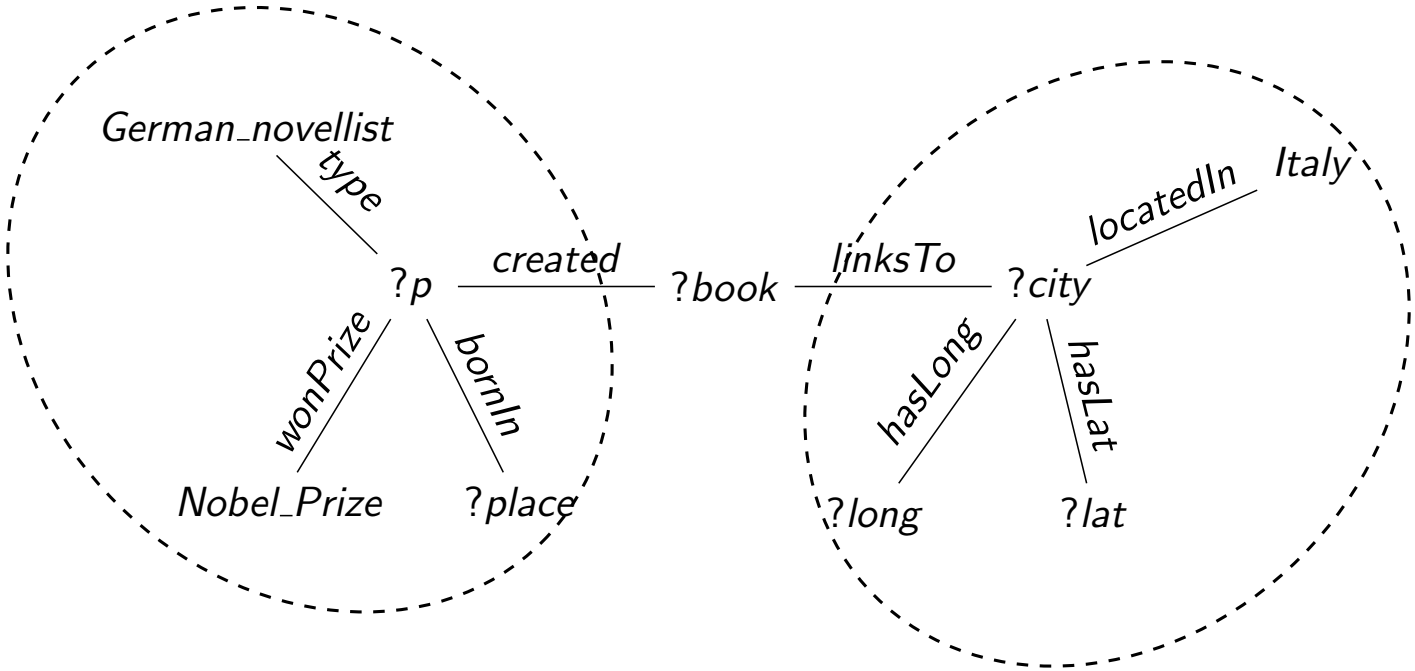
Outline

Given a SPARQL query:



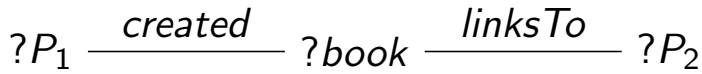
- How to optimize star-shaped subqueries?
- How to capture selectivities between subqueries?
- How to optimize arbitrary-shaped queries?

Query simplification



- We start with identifying optimal plans for subqueries

Query simplification



- 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

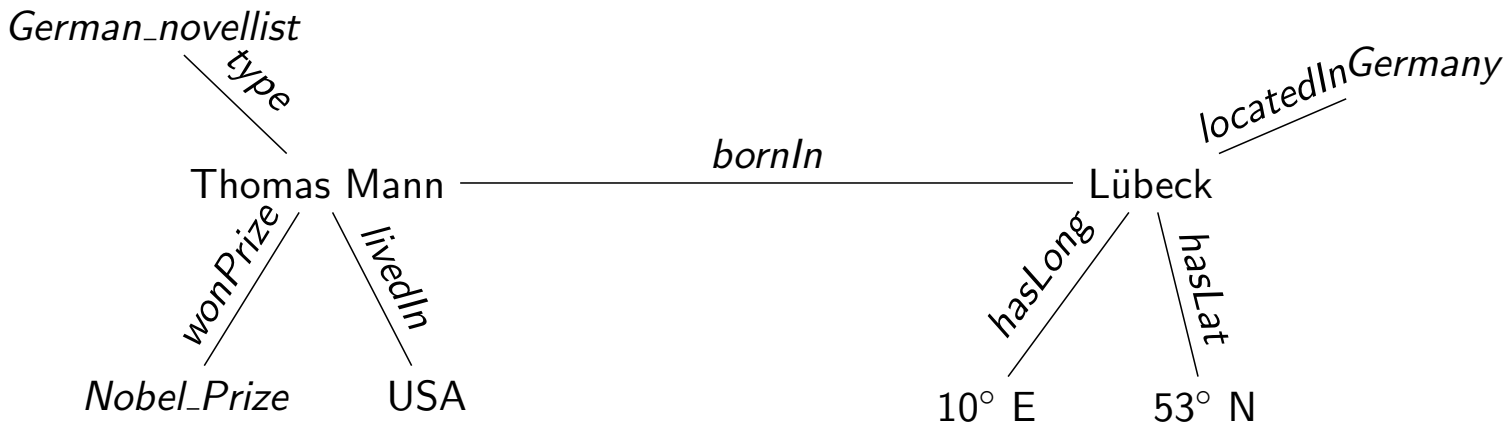
$$?P_1 \xrightarrow[s_1]{created} ?book \xrightarrow[s_2]{linksTo} ?P_2$$

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
...	...	...

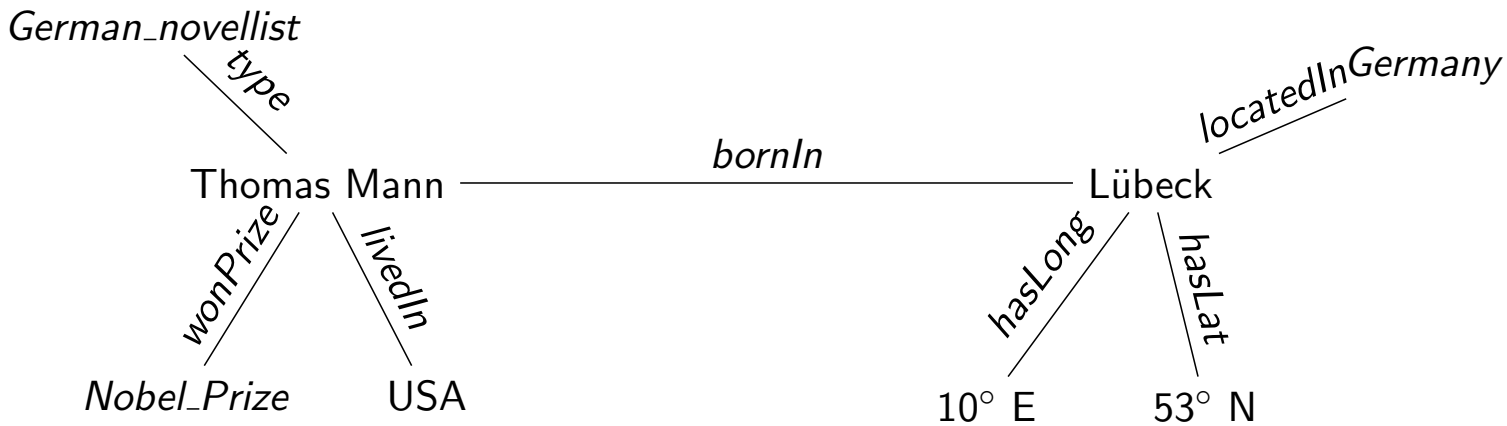
Applicability

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

Outline


German_novellist
type
Thomas Mann
bornIn
Lübeck
locatedIn
Germany
wonPrize
livedIn
hasLong
hasLat
Nobel_Prize
USA
10° E
53° N

Outline



Writer

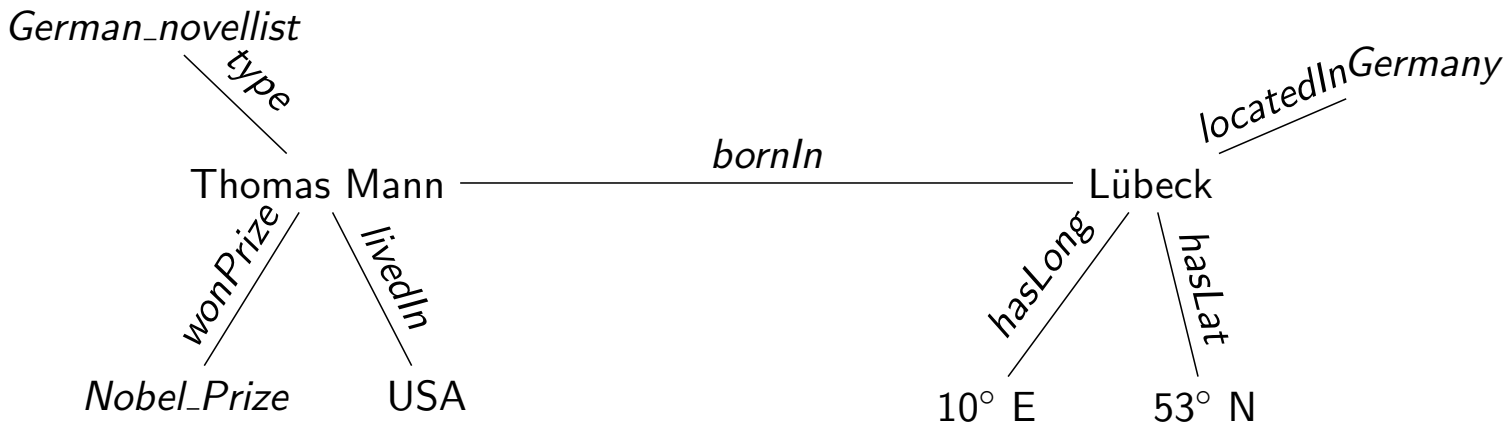
ID	Name	Type	Prize	BornIn
...	...	...	...	...

City

CityId	Name	Country	Long	Lat
...	...	...	...	...

Foreign-Key: Writer:BornIn

Outline



Writer

ID	Name	Type	Prize	BornIn
...	...	...	...	...

City

CityId	Name	Country	Long	Lat
...	...	...	...	...

Foreign-Key: Writer:BornIn

- Only conceptual view for the optimizer, the data can still be stored in triple

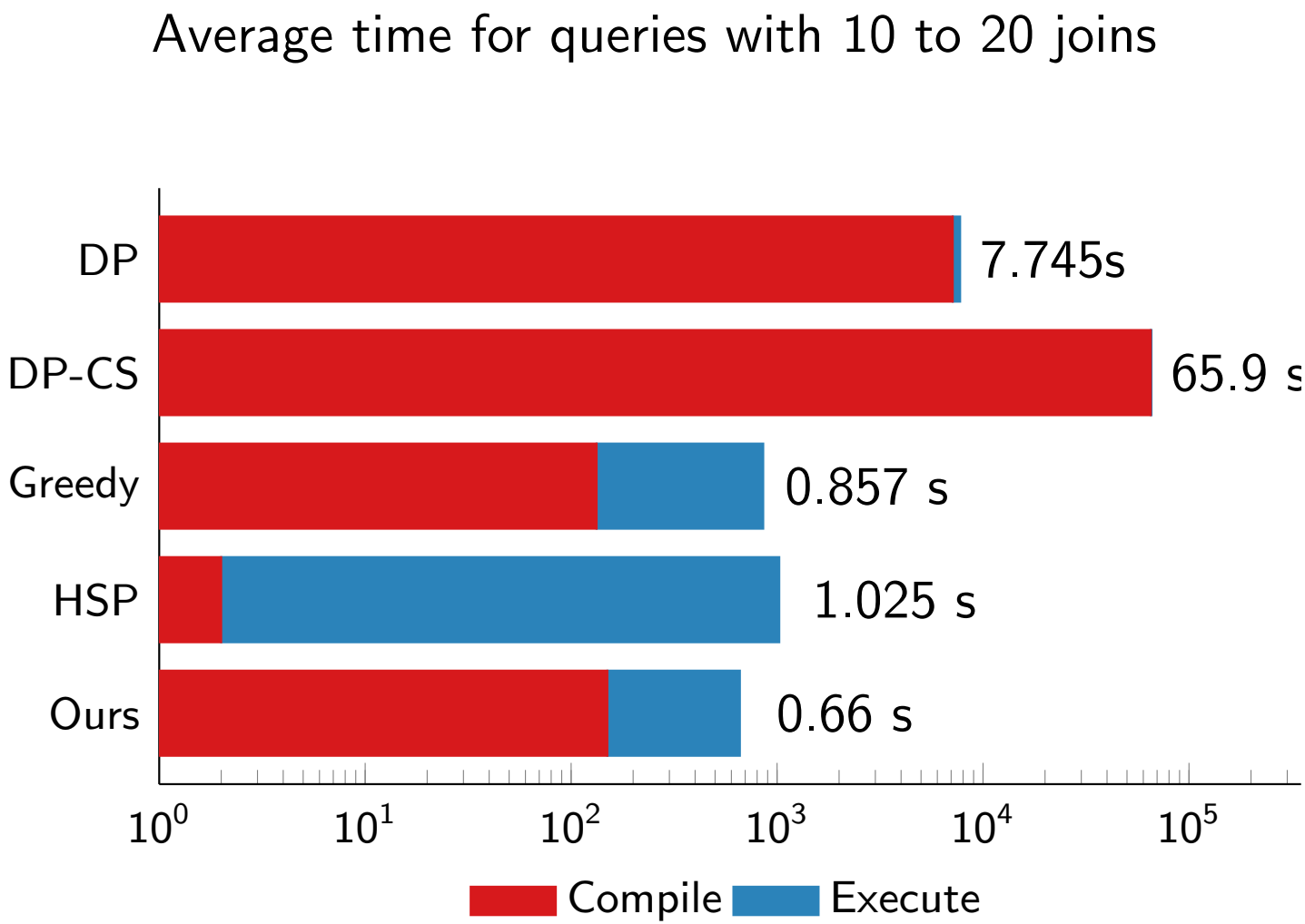
Experiments: Setup

- 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)

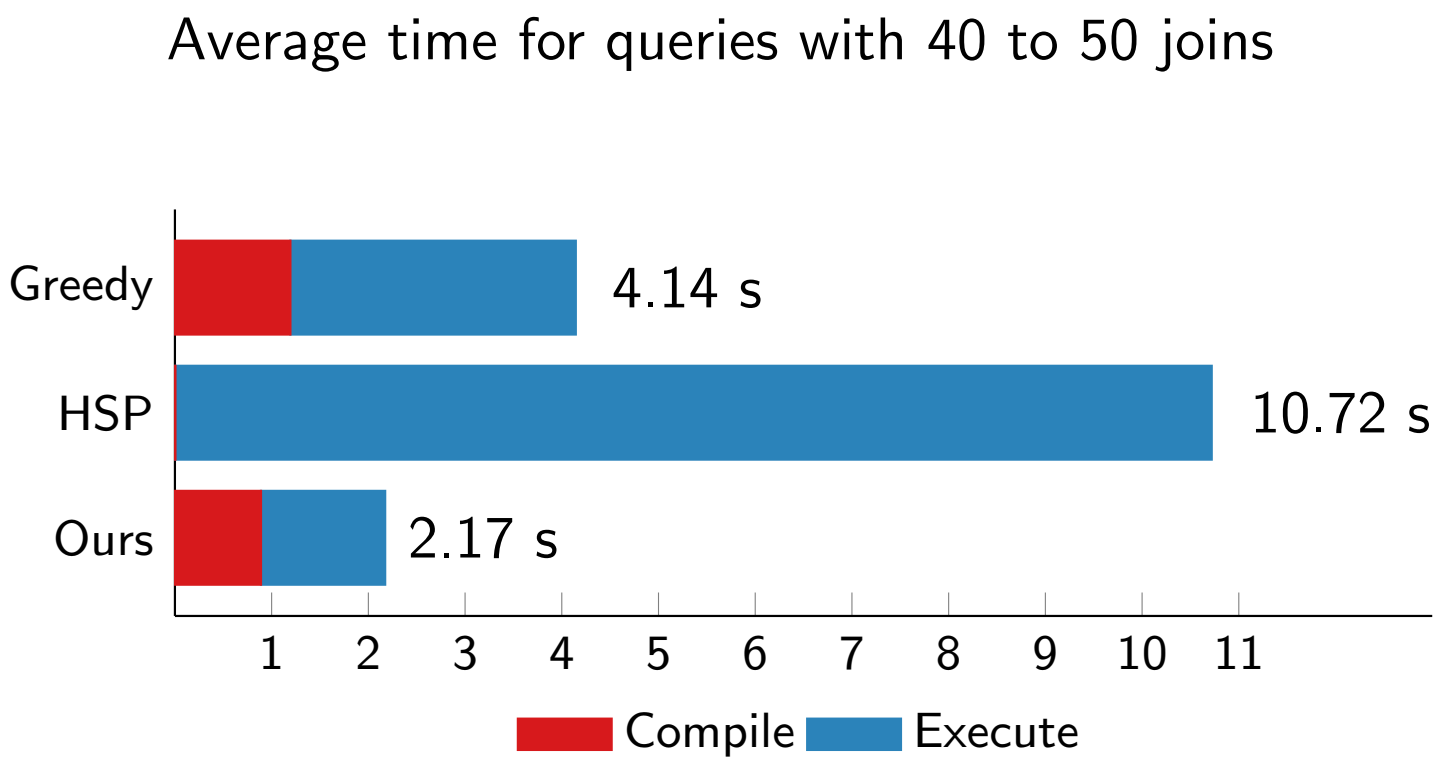
Precomputation: Time and Space

- 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

Query Optimization: Compile and Runtime for YAGO



Query Optimization: Compile and Runtime for YAGO



Conclusions

- Techniques for join ordering in complex SPARQL queries
- Extended the Characteristic Sets for join ordering
- Introduced Characteristic Pairs for cardinality estimation
- Structure-aware SPARQL query simplification

Conclusions

- Techniques for join ordering in complex SPARQL queries
- Extended the Characteristic Sets for join ordering
- Introduced Characteristic Pairs for cardinality estimation
- Structure-aware SPARQL query simplification

Paper also has:

- How to handle constants?
- What is the effect of individual techniques?

Open issues

- Do we capture all correlations? No
  - attributes can also be correlated with structure
- Updates