An Extensible Framework for Query Optimization on TripleT-Based RDF Stores

Bart Wolff[1], George Fletcher[1], and James Lu[2]

[1]Eindhoven University of Technology
[2]Emory University

LWDM 2015
Brussels
27 March 2015

Data

Resource description framework (RDF)

- W3C recommendation data model for linked data
- Graph-based data model, embodying lessons learned from previous data models
  - flexibly capture "things" and their relationships
  - don't force artificial data/metadata distinctions
  - support full spectrum between structured and unstructured data

Data

Resource description framework (RDF)

- W3C recommendation data model for linked data
- Graph-based data model, embodying lessons learned from previous data models
  - flexibly capture "things" and their relationships
  - don't force artificial data/metadata distinctions
  - support full spectrum between structured and unstructured data
- Uses web identifiers (URIs) to identify resources ("things")
- Uses triples to state relationships between things
- serialization formats: N-triples, N3, RDF/XML

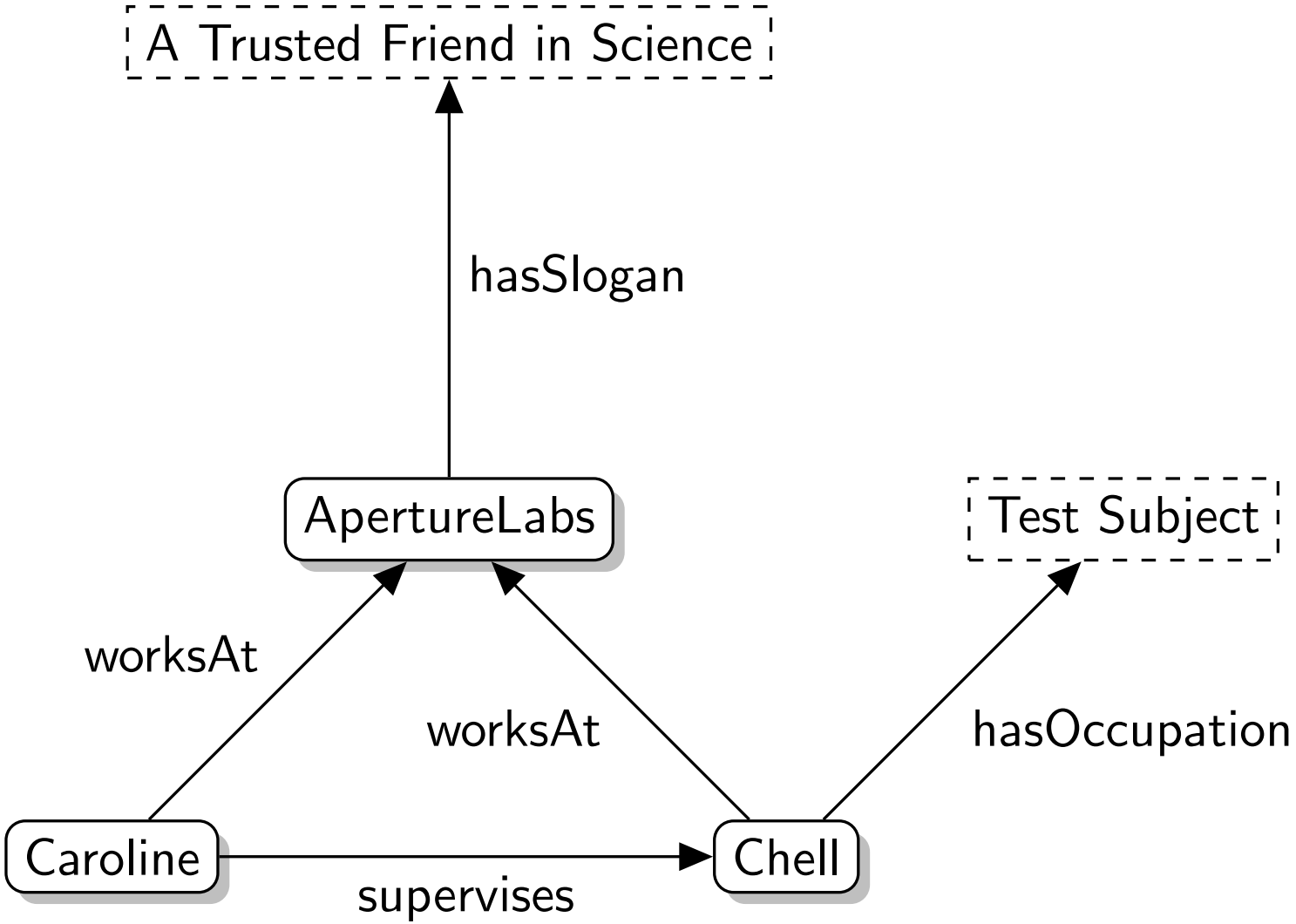
Data

RDF datasets (or: RDF graphs) consist of a finite number of triples:

```
(Chell, worksAt, ApertureLabs)
(Chell, hasOccupation, "Test Subject")
(ApertureLabs, hasSlogan, "A Trusted Friend in Science")
(Caroline, worksAt, ApertureLabs)
(Caroline, supervises, Chell)
```

Each triple is of the shape (s, p, o).

Data


A Trusted Friend in Science
hasSlogan
ApertureLabs
Test Subject
worksAt
worksAt
hasOccupation
Caroline
supervises
Chell

Queries

SPARQL is the W3C recommended language for querying RDF datasets.

```
PREFIX ex: <http://example.org/>
SELECT ?slogan
WHERE {
  ex:Chell ex:worksAt ?x.
  ex:Caroline ex:worksAt ?x.
  ?x ex:hasSlogan ?slogan.
}
```

Queries

We focus on the core of SPARQL: Basic Graph Patterns (BGPs).

Queries

We focus on the core of SPARQL: Basic Graph Patterns (BGPs).

BGPs are constructed from Simple Access Patterns (SAPs):

`(?x, worksAt, ?y)`

A conjunction of one or more SAPs makes up a BGP:

`{(?x, worksAt, ?y), (?y, hasSlogan, ?z)}`

Queries

Results of BGP queries are sets of variable bindings:

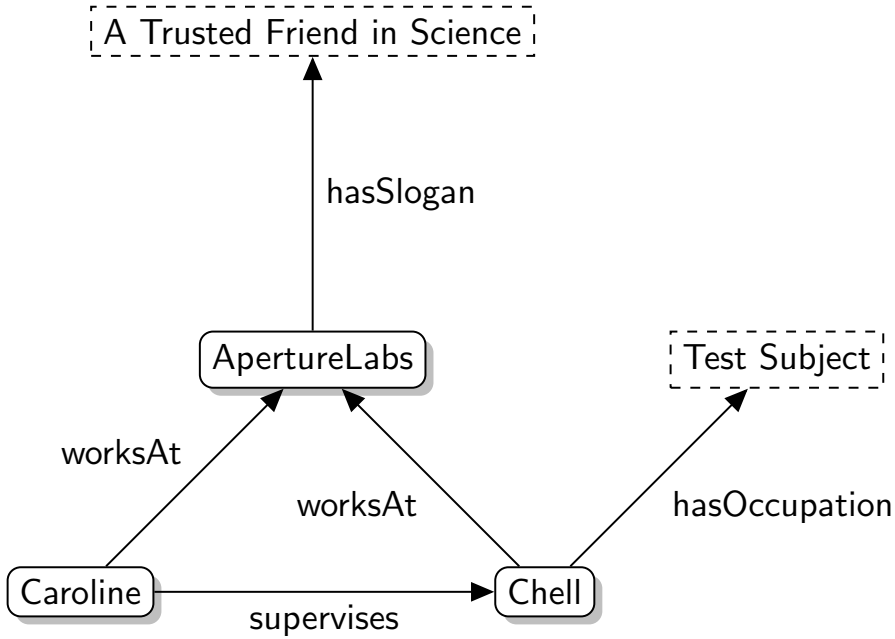
Queries

Results of BGP queries are sets of variable bindings:

```
(?x, worksAt, ?y)
```

On our example dataset, we get:

```
{(?x : Chell),
 (?y : ApertureLabs)}
{(?x : Caroline),
 (?y : ApertureLabs)}
```



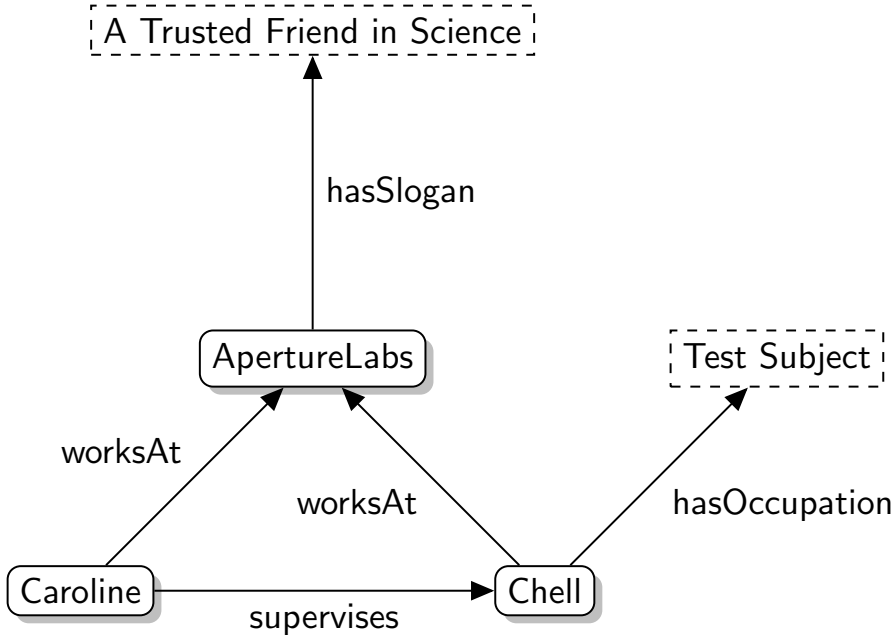
Queries

Results of BGP queries are sets of variable bindings:

```
{(Chell, worksAt, ?x),
 (?x, hasSlogan, ?y)}
```

On our example dataset, we get:

```
{(?x : ApertureLabs),
 (?y : "A Trusted...")}
```



Storage in external memory

Storing and querying RDF datasets is supported by RDF stores (also known as triple stores). As adoption of RDF grows, RDF stores have to deal with increasingly large datasets.

Storage in external memory

Storing and querying RDF datasets is supported by RDF stores (also known as triple stores). As adoption of RDF grows, RDF stores have to deal with increasingly large datasets.

Two basic storage alternatives

1. flat file of triples, one per line
   - ... sorted on some subset of positions (e.g., by subject values)
2. index data structure: maps a search key to a set of matching triples
   - B+tree, hash table

Storage: on the varieties of triple indexes

state of the art in the market: value-based indexing

- MAP (2007)
- HexTree (2008)
- TripleT (2009)

Storage: on the varieties of triple indexes

state of the art in the market: value-based indexing

- MAP (2007)
- HexTree (2008)
- TripleT (2009)

still in the research lab: structure-based indexing

- group triples both by the values they share and by the *structure* they share in the graph (e.g., similar neighborhood topology)
- structure-based indexing also successful in XML data management

Storage: on the varieties of (value-based) triple indexes

Let's focus on value-based indexes

For graph G, let

$$
\begin{aligned}
\mathcal{S}(G) &= \{s \mid (s,p,o) \in G\} \\
\mathcal{P}(G) &= \{p \mid (s,p,o) \in G\} \\
\mathcal{O}(G) &= \{o \mid (s,p,o) \in G\} \\
\mathcal{A}(G) &= \mathcal{S}(G) \cup \mathcal{P}(G) \cup \mathcal{O}(G)
\end{aligned}
$$

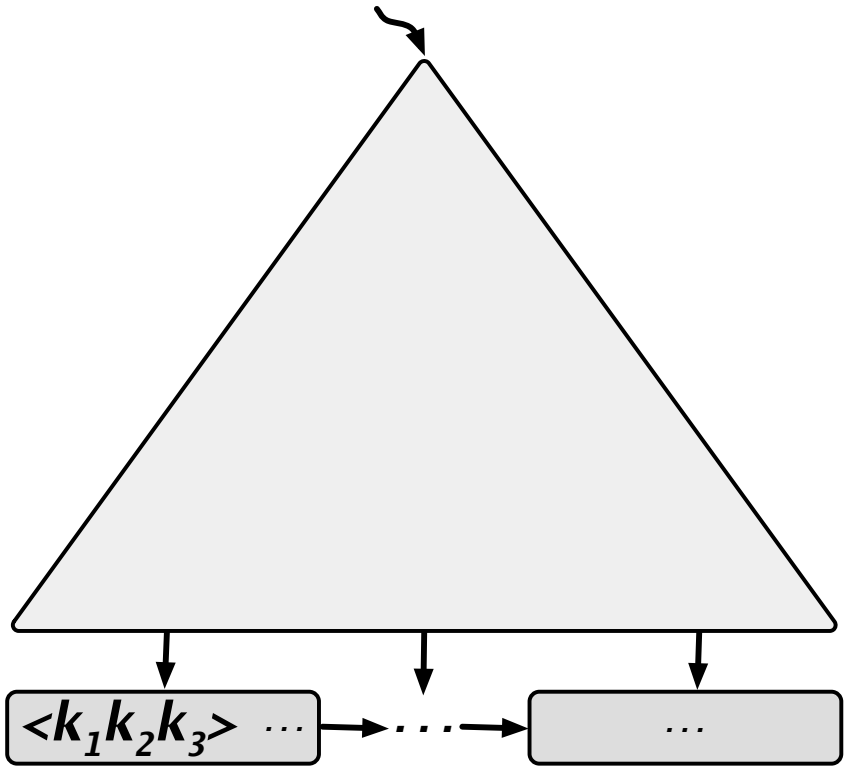
Storage: on the varieties of triple indexes

MAP: index all permutations of S, P, and O

$$
\begin{aligned}
SPO &= \{s\#p\#o \mid (s,p,o) \in G\} \\
SOP &= \{s\#o\#p \mid (s,p,o) \in G\} \\
PSO &= \{p\#s\#o \mid (s,p,o) \in G\} \\
POS &= \{p\#o\#s \mid (s,p,o) \in G\} \\
OSP &= \{o\#s\#p \mid (s,p,o) \in G\} \\
OPS &= \{o\#p\#s \mid (s,p,o) \in G\}
\end{aligned}
$$

Storage: on the varieties of triple indexes

MAP: index all permutations of S, P, and O



... times six

Storage: on the varieties of triple indexes

MAP: index all permutations of S, P, and O

Most popular approach to triple storage in practice

- RDF-3X: open source triple store
- Virtuoso: open source and commercial industrial-strength triple store
- Sesame/BigData: open source and commercial industrial-strength triple store

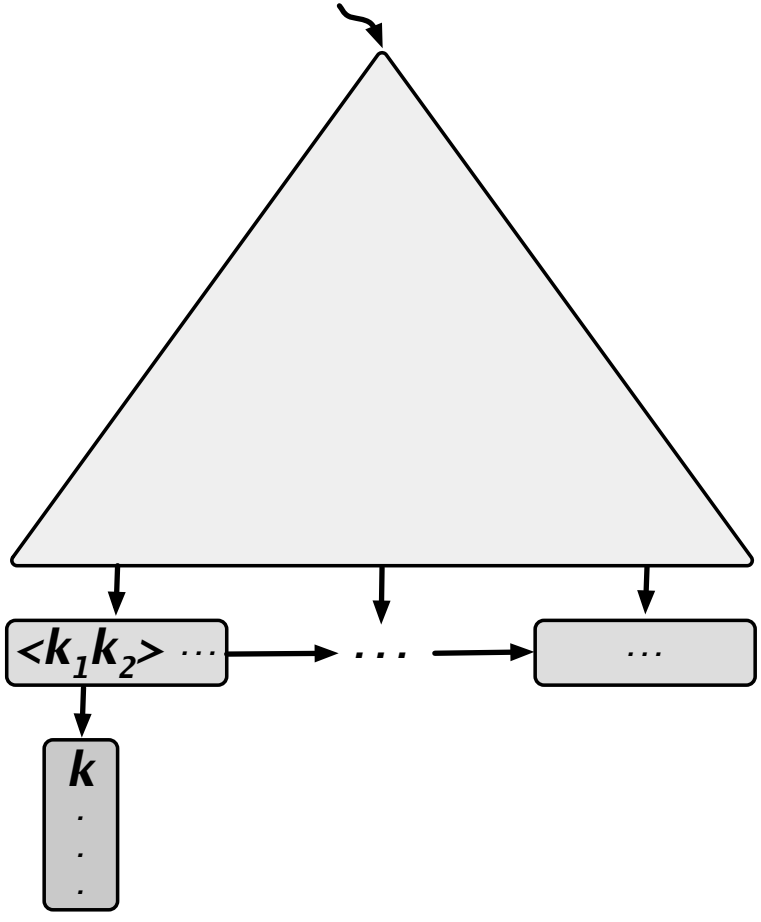
Storage: on the varieties of triple indexes

HexTree: index all pairs of S, P, and O

$$
\begin{aligned}
SP &= \{s\#p \mid (s,p,o) \in G\} \\
SO &= \{s\#o \mid (s,p,o) \in G\} \\
PS &= \{p\#s \mid (s,p,o) \in G\} \\
PO &= \{p\#o \mid (s,p,o) \in G\} \\
OS &= \{o\#s \mid (s,p,o) \in G\} \\
OP &= \{o\#p \mid (s,p,o) \in G\}
\end{aligned}
$$

Storage: on the varieties of triple indexes

HexTree: index all pairs of S, P, and O



... times six

Storage: on the varieties of triple indexes

HexTree: partners share payload

$$
\begin{aligned}
SP &\rightarrow \{o \in \mathcal{O}(G) \mid (s, p, o) \in G\} \\
PS &\rightarrow \{o \in \mathcal{O}(G) \mid (s, p, o) \in G\}
\end{aligned}
$$

Storage: on the varieties of triple indexes

MAP and HexTree

- *strengths:* support all native joins expressible in the SPARQL WHERE clause, using fast "merge" joins, with no constraints on "schema"

Storage: on the varieties of triple indexes

MAP and HexTree

- *strengths:* support all native joins expressible in the SPARQL WHERE clause, using fast "merge" joins, with no constraints on "schema"
- *weaknesses*
  - lots of (redundant) storage space
  - access to multiple data structures to perform joins

Storage: on the varieties of triple indexes

MAP and HexTree

- *strengths:* support all native joins expressible in the SPARQL WHERE clause, using fast "merge" joins, with no constraints on "schema"
- *weaknesses*
  - lots of (redundant) storage space
  - access to multiple data structures to perform joins
  - in general, weak data locality

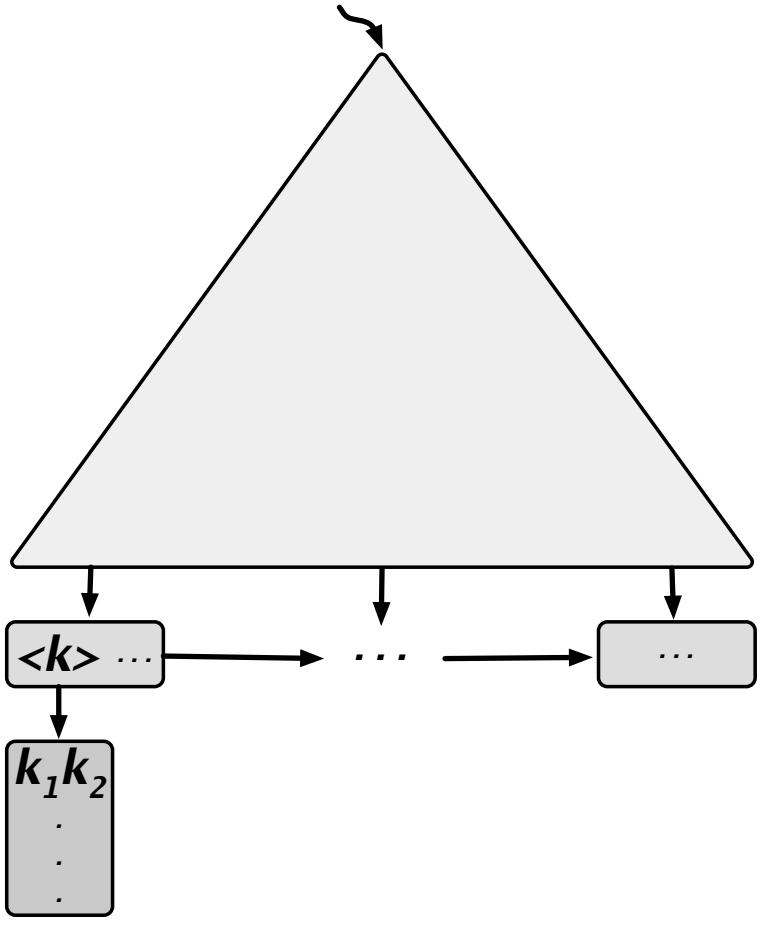
Storage: on the varieties of triple indexes

TripleT (Fletcher and Beck, CIKM 2009)

- index $\mathcal{A}(G)$
- just one data structure – a three-way triple tree

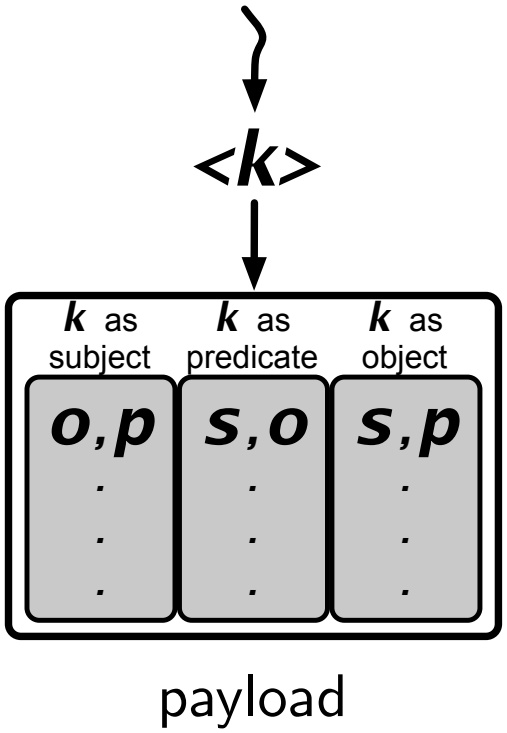
Storage: on the varieties of triple indexes

TripleT



Storage: on the varieties of triple indexes

TripleT



Storage: on the varieties of triple indexes

TripleT advantages

- retains strengths of MAP and HexTree

Storage: on the varieties of triple indexes

TripleT advantages

- retains strengths of MAP and HexTree
- reduced storage space
- reduced key size (i.e., shallower trees)

Storage: on the varieties of triple indexes

TripleT advantages

- retains strengths of MAP and HexTree
- reduced storage space
- reduced key size (i.e., shallower trees)
- single data structure to perform joins

Storage: on the varieties of triple indexes

TripleT advantages

- retains strengths of MAP and HexTree
- reduced storage space
- reduced key size (i.e., shallower trees)
- single data structure to perform joins
- in general, stronger data locality

Storage: on the varieties of triple indexes

TripleT advantages

- retains strengths of MAP and HexTree
- reduced storage space
- reduced key size (i.e., shallower trees)
- single data structure to perform joins
- in general, stronger data locality

Leads to significantly reduced storage and query processing costs

Storage: on the varieties of triple indexes

TripleT advantages

- retains strengths of MAP and HexTree
- reduced storage space
- reduced key size (i.e., shallower trees)
- single data structure to perform joins
- in general, stronger data locality

Leads to significantly reduced storage and query processing costs

idea appears in various guises in products

- open source TU/e implementation (this work)

  `https://github.com/b-w/TripleT`
- open source 4store engine
- and storage of RDF in so-called column stores
  - open source MonetDB
  - commercial Vertica

Storage: on the varieties of triple indexes

Also, relational DB-based RDF solutions are available in industrial-strength IBM DB2 and Oracle Database 11G products.

Both use various hybrids of the MAP, HexTree, and TripleT indexes

Goal

An RDF dataset consists entirely of triples. This results in queries with many self-joins, which provides a novel performance challenge.

Currently, BGP query optimization is an active area of research, where the challenge is to deliver interactive response times on BIG graphs.

Goal

Query optimization on TripleT has not been systematically studied up to this point.

This work: we focus on optimization of query plans for BGP queries on TripleT.

We investigate how the use of heuristics and external information (e.g. dataset statistics) can contribute towards a more intelligent way of generating query plans, minimizing query execution time over the TripleT RDF store.

Contributions

- We present an open source secondary-memory implementation of the TripleT RDF store and index.
- We propose a framework for query optimization in the form of a generic, rule-based algorithm used for generating query plans for given Basic Graph Pattern queries.
- We introduce a number of rules for use by our optimization framework.
- We implement this optimization framework on top of our TripleT store.
- We perform an extensive empirical study into the effectiveness of our rules in generating optimized query plans.

Query optimization

To answer a BGP query

$$(s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge \cdots \wedge (s_n, p_n, o_n)$$

we need to compute the natural join over the binding sets produced by its SAPs:

$$(s_1, p_1, o_1) \bowtie (s_2, p_2, o_2) \bowtie \cdots \bowtie (s_n, p_n, o_n)$$

Query optimization

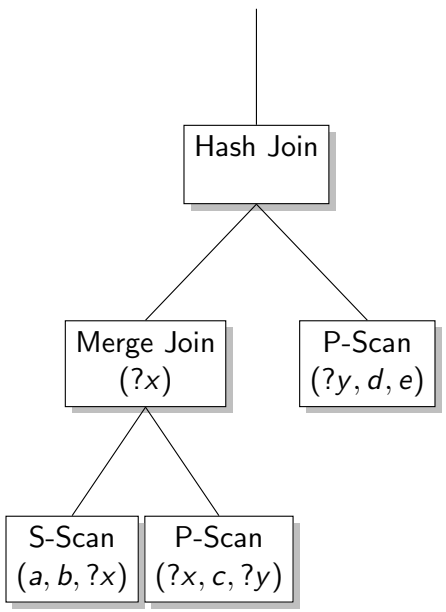
To answer a BGP query

$$(s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge \cdots \wedge (s_n, p_n, o_n)$$

we need to compute the natural join over the binding sets produced by its SAPs:

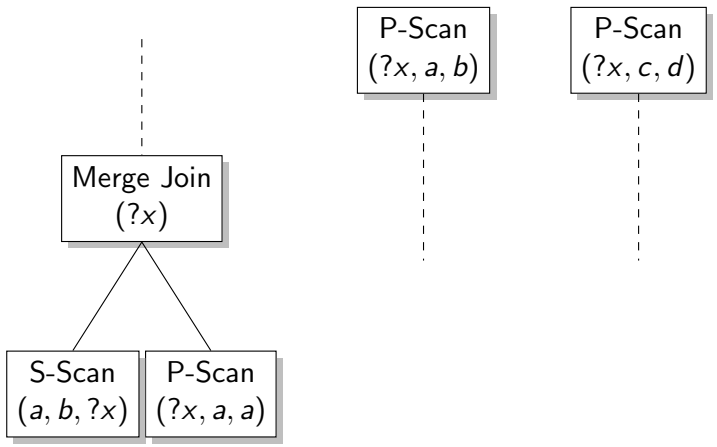
$$(s_1, p_1, o_1) \bowtie (s_2, p_2, o_2) \bowtie \cdots \bowtie (s_n, p_n, o_n)$$

Doing so requires a (physical) query plan.



Query optimization

Deciding in what order to perform these joins is crucial to query response time.



Different plans for the same query can have response times several orders of magnitude apart.

Query optimization

Goal: separation of algorithm and rules.

Solution: a generic algorithm for query plan generation, containing several *decision points* which are guided by a set of rules.

Query optimization

The steps of the algorithm are:

1. Compute the *atom collapse* of a given BGP.
2. From the atom collapse, compute a *join graph*.
3. From the join graph, compute a query plan.

Query optimization

The steps of the algorithm are:

1. Compute the *atom collapse* of a given BGP.
2. From the atom collapse, compute a *join graph*.
3. From the join graph, compute a query plan.

Several rules work to obtain two primary goals:

1. Maximize merge joins
2. Minimize (expected) intermediate results

Query optimization - Example

As an example, consider the following BGP:

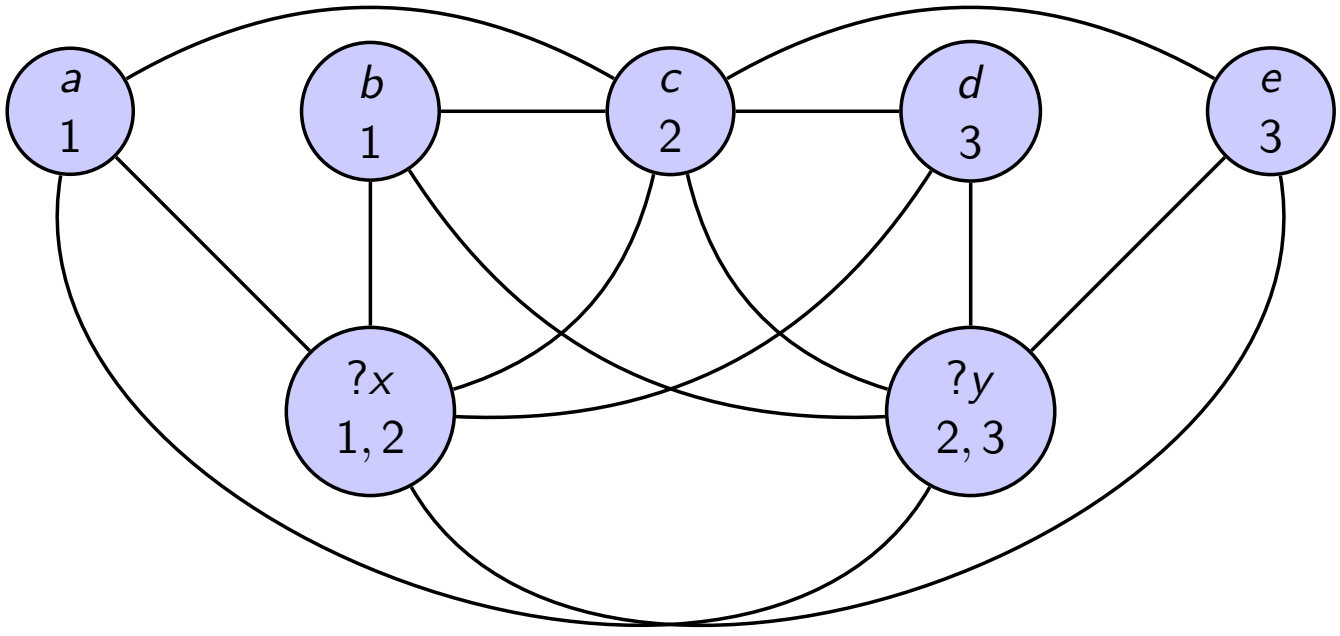
$$(a, b, ?x)_1 \wedge (?x, c, ?y)_2 \wedge (?y, d, e)_3$$

Query optimization - Example

As an example, consider the following BGP:

$$(a, b, ?x)_1 \wedge (?x, c, ?y)_2 \wedge (?y, d, e)_3$$

We first compute the atom collapse:

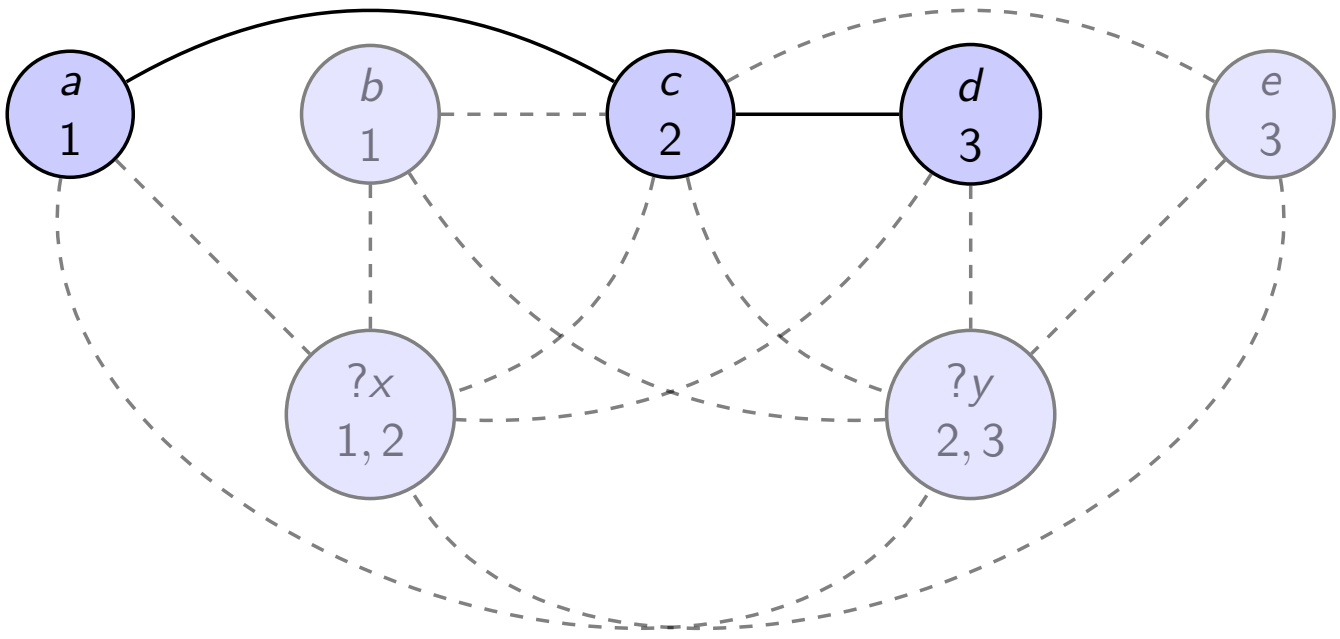


Query optimization - Example

As an example, consider the following BGP:

$$(a, b, ?x)_1 \wedge (?x, c, ?y)_2 \wedge (?y, d, e)_3$$

We then use this to obtain a join graph:

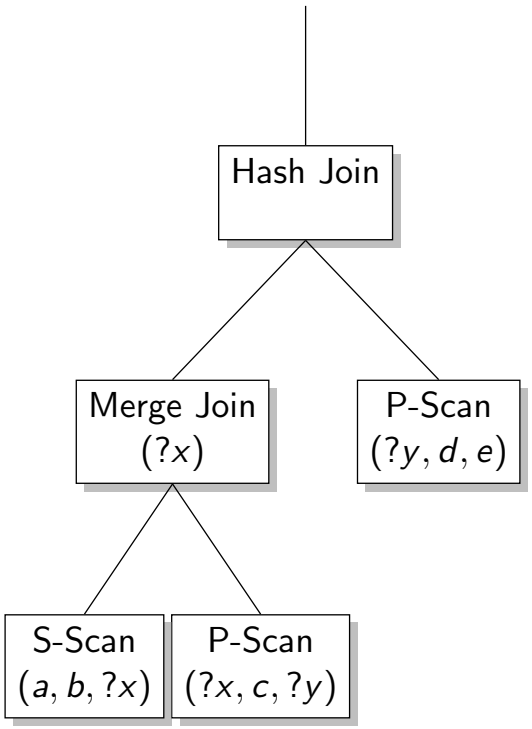


Query optimization - Example

As an example, consider the following BGP:

$$(a, b, ?x)_1 \wedge (?x, c, ?y)_2 \wedge (?y, d, e)_3$$

The join graph is then used to obtain a query plan:



Query optimization

Seed rules

- Seed rule 1: select one preferred seed for each distinct SAP in the input set based on the positions of the atoms in the SAP
  - following the ordering $s \succ o \succ p$.

Query optimization

Seed rules

- Seed rule 1: select one preferred seed for each distinct SAP in the input set based on the positions of the atoms in the SAP
  - following the ordering $s \succ o \succ p$.
- Seed rule 2: prioritizes the atoms in the SAP according to their selectivity, as indicated by dataset statistics.

Query optimization

Join rules

- Join rule 1: merge join prioritization.

Query optimization

Join rules

- Join rule 1: merge join prioritization.
- Join rule 2: prioritize joins involving the most selective SAPs
  - $(s, p, o) \succ (s, ?, o) \succ (s, p, ?) \succ (?, p, o) \succ (s, ?, ?) \succ (?, ?, o) \succ (?, p, ?) \succ (?, ?, ?)$

Query optimization

Join rules

- Join rule 1: merge join prioritization.
- Join rule 2: prioritize joins involving the most selective SAPs
  - $(s, p, o) \succ (s, ?, o) \succ (s, p, ?) \succ (?, p, o) \succ (s, ?, ?) \succ (?, ?, o) \succ (?, p, ?) \succ (?, ?, ?)$
- Join rule 3: prioritize joins between SAPs that have the most selective positioning of join variables
  - $s \bowtie p \succ o \bowtie p \succ s \bowtie o \succ s \bowtie s \succ o \bowtie o \succ p \bowtie p$.

Query optimization

Join rules

- Join rule 1: merge join prioritization.
- Join rule 2: prioritize joins involving the most selective SAPs
  - $(s, p, o) \succ (s, ?, o) \succ (s, p, ?) \succ (?, p, o) \succ (s, ?, ?) \succ (?, ?, o) \succ (?, p, ?) \succ (?, ?, ?)$
- Join rule 3: prioritize joins between SAPs that have the most selective positioning of join variables
  - $s \bowtie p \succ o \bowtie p \succ s \bowtie o \succ s \bowtie s \succ o \bowtie o \succ p \bowtie p$.
- Join rule 4: prioritize joins between SAPs which feature a literal value (e.g. "Sue") in one of its positions, over those featuring only URIs (e.g. "http://example.org/Sue")

Query optimization

Join rules

- Join rule 1: merge join prioritization.
- Join rule 2: prioritize joins involving the most selective SAPs
  - $(s, p, o) \succ (s, ?, o) \succ (s, p, ?) \succ (?, p, o) \succ (s, ?, ?) \succ (?, ?, o) \succ (?, p, ?) \succ (?, ?, ?)$
- Join rule 3: prioritize joins between SAPs that have the most selective positioning of join variables
  - $s \bowtie p \succ o \bowtie p \succ s \bowtie o \succ s \bowtie s \succ o \bowtie o \succ p \bowtie p$.
- Join rule 4: prioritize joins between SAPs which feature a literal value (e.g. "Sue") in one of its positions, over those featuring only URIs (e.g. "http://example.org/Sue")
- Join rule 5: expected output minimization. Using a precomputed statistics database, output sizes of joins are known in advance.
  - selectivity of atoms, pairs of atoms, and joins of SAPs

Experiments: goals

Goal is to answer a number of questions:

1. What is the contribution of each of the rules?
2. How effective are combinations or subsets of rules?
3. What is the added value of statistics?
4. How do the rules perform under different datasets and queries?

Experiments: data

9 different datasets, from 3 different sources:

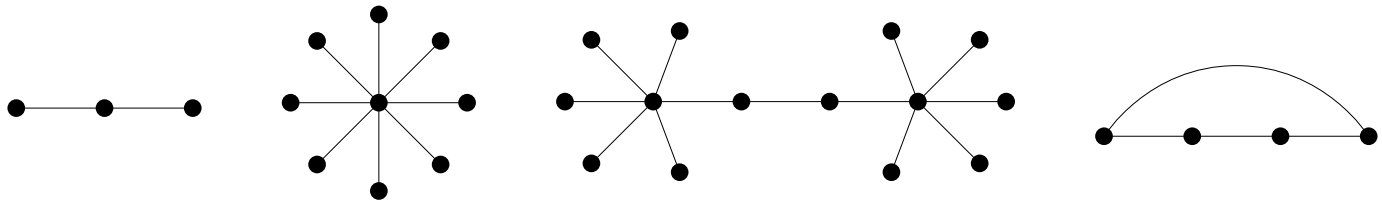
- DBpedia
- SP^2Bench
- UniProt

...and in 3 different sizes:

- 100.000 triples
- 1.000.000 triples
- 10.000.000 triples

Experiments: queries

Queries in different shapes:



...and of different sizes and selectivity.

Experiments: setup

We test with different configurations of rules.

		Rules						
		S1	S2	J1	J2	J3	J4	J5
Runs	A-1	1		1	2	3	4	
	A-2		1					1
	A-3	1	2	1	2	3	4	5
	A-4	2	1	2	3	4	5	1
	B-1	1			1	2	3	
	B-2	1		1		2	3	
	B-3	1		1	2		3	
	B-4	1		1	2	3		
	C-1	1		4	3	2	1	
	C-2	1		3	2	1	4	
	C-3	1		2	1	4	3	
	D-1		1	1	3			2
	D-2	1		2	1			
	D-3		1		1	2		3

Experiments: setup

Each of the runs has a particular purpose

- the **A-runs** are designed to test the heuristics rules against the statistics rules
- the **B-runs** aim to get a sense of the value of the individual heuristics rules
- the **C-runs** focus on the ordering of rules
- the **D-runs** are used to test different subset combinations of rules

For each configuration we test on all datasets with all queries, and observe performance.

Experiments: setup

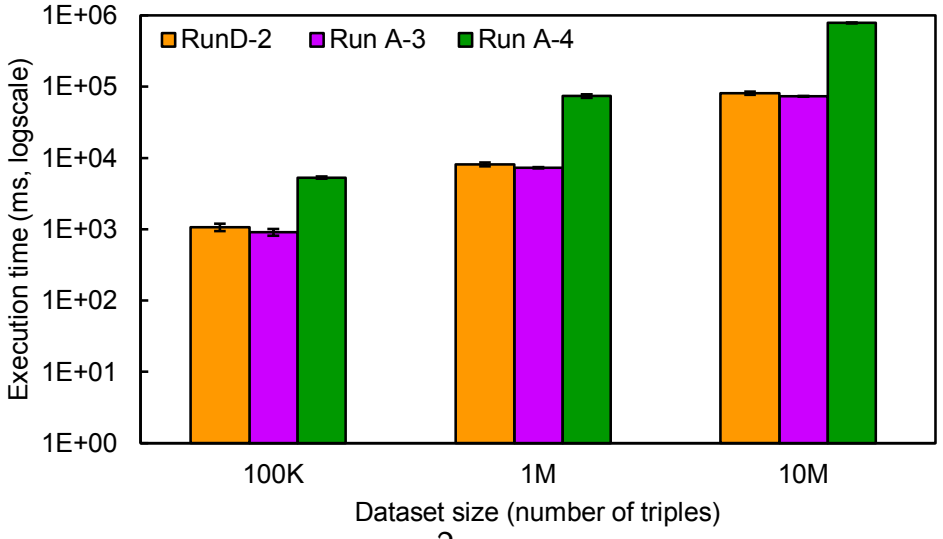
Each of the runs has a particular purpose

- the **A-runs** are designed to test the heuristics rules against the statistics rules
- the **B-runs** aim to get a sense of the value of the individual heuristics rules
- the **C-runs** focus on the ordering of rules
- the **D-runs** are used to test different subset combinations of rules

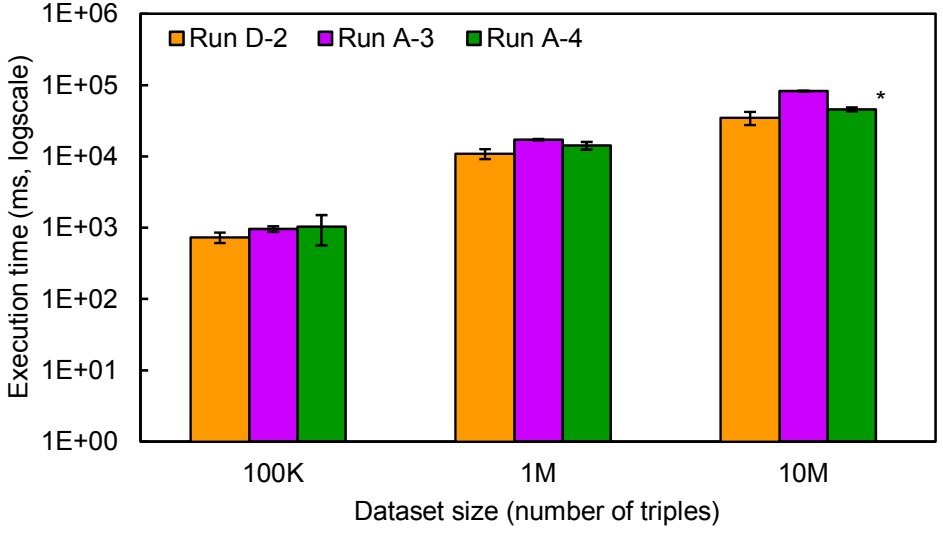
For each configuration we test on all datasets with all queries, and observe performance.

The B- and C-runs confirmed the observations in the other runs, so we only show the main observations from the A- and D-runs

Experiments: overview of runs A-3, A-4, and D-2



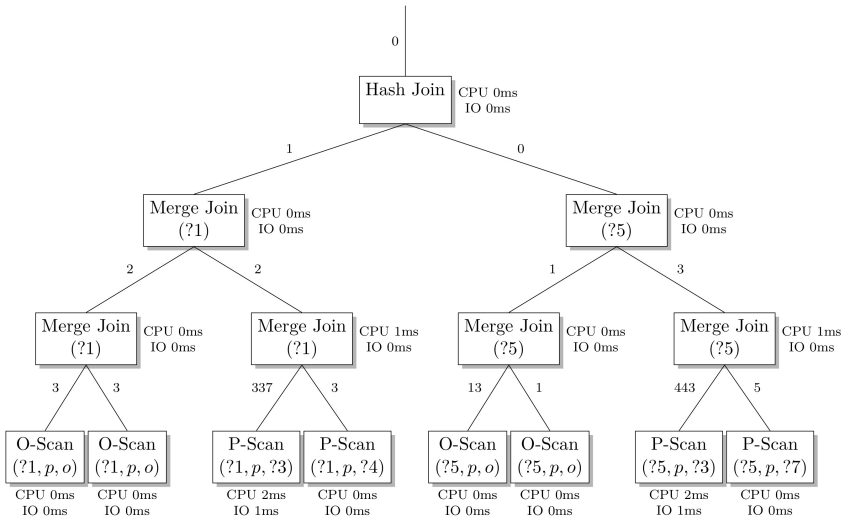
SP^{2}Bench



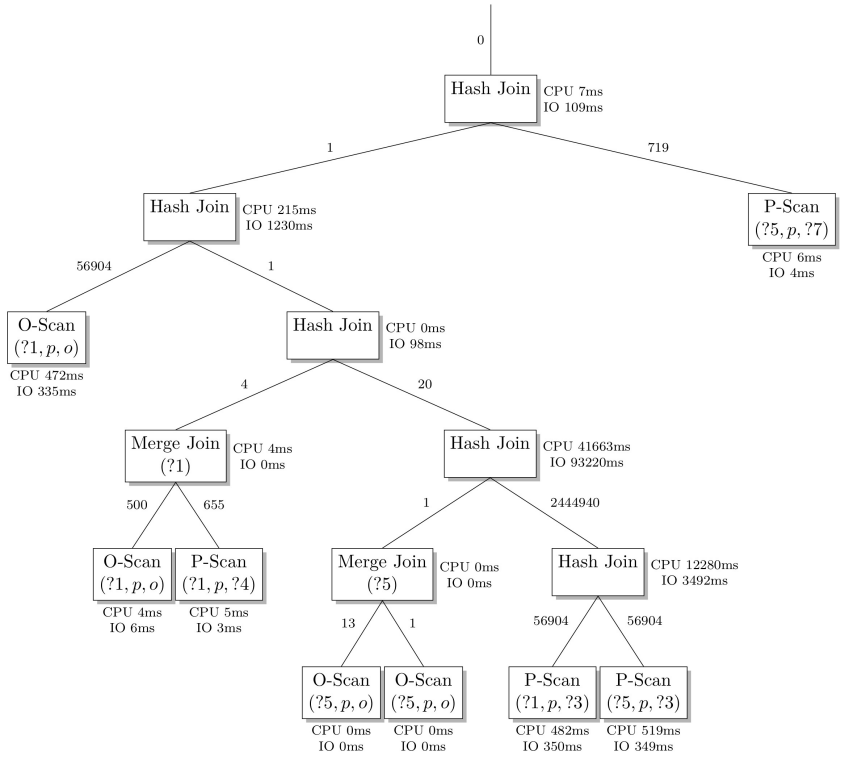
* Excluding 15 / 60 data points with values > 1E+06

UniProt

Experiments: A runs on SP2Bench



Run A-3



Run A-4

A BGP with 8 SAPs. Execution of A-4 is an order of magnitute slower.

Experiments: conclusions

Simple heuristics can consistently generate good query plans.

- these findings corroborate the results obtained by Tsialiamanis et al. EDBT 2012

Experiments: conclusions

Simple heuristics can consistently generate good query plans.

- these findings corroborate the results obtained by Tsialiamanis et al. EDBT 2012

The impact of statistics and the statistical prioritization join rule is measurable but limited.

Experiments: conclusions

Simple heuristics can consistently generate good query plans.

- these findings corroborate the results obtained by Tsialiamanis et al. EDBT 2012

The impact of statistics and the statistical prioritization join rule is measurable but limited.

Simple heuristic-driven rule sets such as A-3 and D-2 are the clear winners.

What's next?

Future directions:

- Extensively study the problem of statistics-based query optimization on TripleT.
- Study runtime optimization strategies in our framework, such as sideways information passing.
- Continue open-source development of TripleT into a fully-featured RDF data management solution.

What's next?

Future directions:

- Extensively study the problem of statistics-based query optimization on TripleT.
- Study runtime optimization strategies in our framework, such as sideways information passing.
- Continue open-source development of TripleT into a fully-featured RDF data management solution.

Thank you!