Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

RDF-3X: a RISC-style Engine for RDF
Scalable Join Processing on Very Large RDF Graphs

CS561 – Web Data Management
Professors: Christophides Vassilis

CSD - UoC (Computer Science Department, University of Crete) , 2012-2013
Presentation Date: 21/05/2013

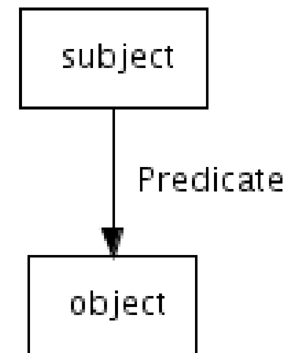
Lecturer: Alogdianaki Eleni 777

Overview

1. Preliminaries RDF – SPARQL
2. Why RDF-3X?
3. Aggregated Indices
4. Translating SPARQL queries
5. Optimizing Join Ordering
6. Selectivity Estimations
7. Evaluation
8. Scalable Join Processing
9. SIP - Sideways Information Parsing
10. SIP on Merge and Hash Joins
11. Run time Acceleration of Index Scans
12. Selectivity Estimation - SIP
13. Evaluation

Preliminaries

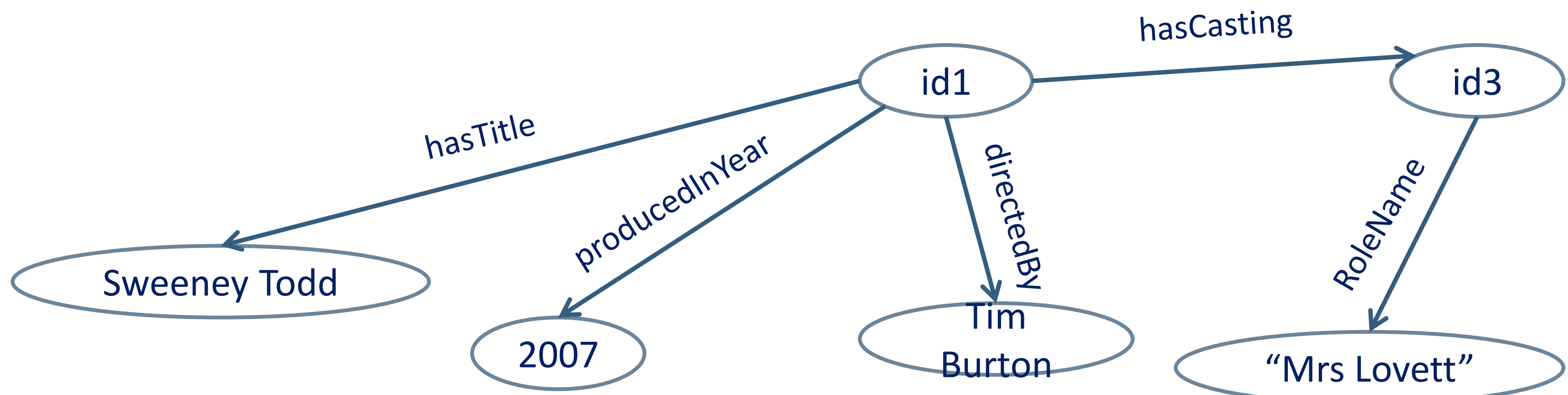
RDF (Resource Description Framework)



- conceptually a labeled graph
- graph is stored as a collection of facts
- each edge represents a fact (triple)
- items represented in the form (subject, predicate, object)

Examples:

- (id1; hasTitle; "Sweeney Todd"),
- (id1; producedInYear; "2007"),
- (id1; directedBy; "Tim Burton"),
- (id1; hasCasting; id3),
- (id3; RoleName; "Mrs: Lovett"), and so on.



Preliminaries
SPARQL Query Language

SPARQL: SPARQL Protocol and RDF Query Language

- ✓ query language for databases
- ✓ able to retrieve and manipulate data stored in RDF format

Example:

```
Select ?title WHERE {
?m <hasTitle> ?title.
?m <hasCasting> ?c.
?c <Actor> ?a.
?a <hasName> "Johnny Depp" }
```

Why RDF-3X?

RDF more and more popular

- ✓ Semantic-Web
- ✓ Life sciences
- ✓ Web 2.0 platforms – Social media Networks

RDF storage, indexing and query processing faces problems:

- Physical database design difficult
- No global schema
- Very fine grained data items
- Statistics (for the Execution plan optimization) hard to predict (due to no schema)

Solution: RDF-3X

- ✓ RISC – style engine
- ✓ based on a single triples table
- ✓ employs fast merge joins
- ✓ determines the best join ordering

Storage of RDF Data – RDF-3X

Based on a single giant "triples table"

Store all triples in a clustered B+ tree

- triples sorted lexicographically
- allows the conversion of SPARQL patterns into range scans

Replace all literals by ids using a mapping dictionary

Benefits:

- ✓ compresses triple store (now contains only ids)
- ✓ fast and efficient scan

After processing query:

- ids transformed into literals

Storage RDF Data

Example Dictionary Compression

Dictionary Compression

Triples		
object214	hasColour	blue
object214	belongsTo	object352
…	…	…

Triples		
Subject	Predicate	Object
0	1	2
0	3	4
…	…	…

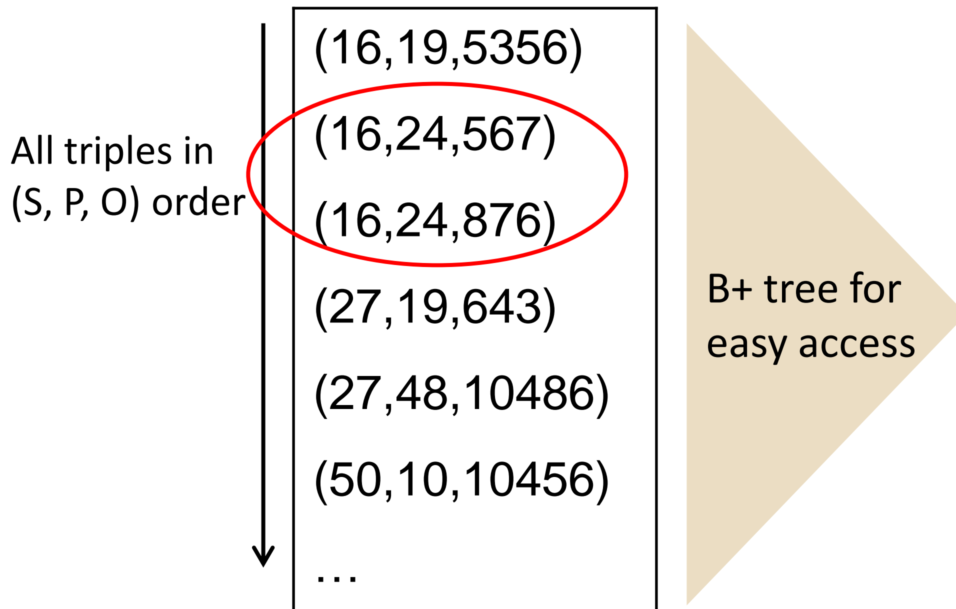
Mapping Dictionary	
ID	Value
0	object214
1	hasColour
2	blue
3	belongsTo
4	object352
…	…

Storage RDF Data

Example Query answering

Example: Albert_Einstein invented ?x

- Build clustered index over (S, P, O)



1. Lookup ids for constants:
   Albert_Einstein=16, invented=24
2. Lookup known prefix in index:
   (16,24,0)
3. Read results while prefix matches:
   (16,24,567), (16,24,876)
   come already sorted!

- Build similar clustered indexes for all six combinations:
  SPO, OPS, PSO, SOP, OSP, POS

Storage of RDF Data

Compressed Indexes

To guarantee every possible pattern can be answered with variables in any position of triple (with single index scan):

Indexes are kept for every sorting order combination on Subject, Property and Object (SPO, SOP, OSP, OPS, PSO, POS)

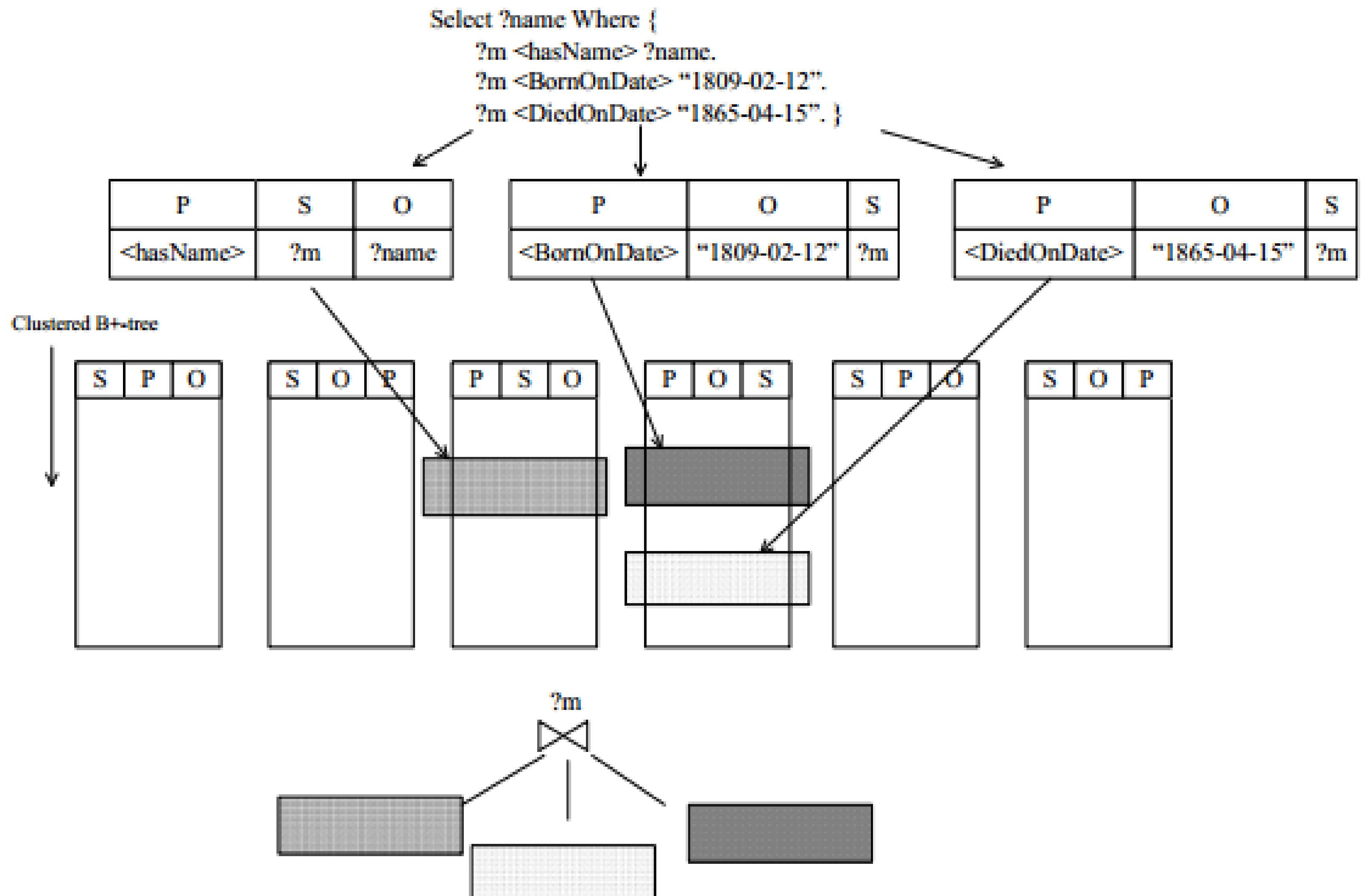
Use generic terminology: value1, value2, value3

Leads to:

- ✓ Compression scheme for triples
- ✓ Store only changes between triples (not full triples)

RDF-3X

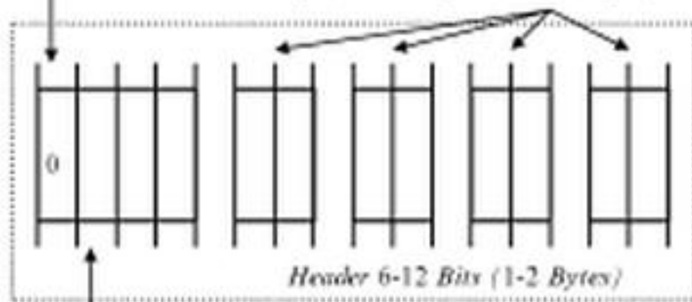
Indexing



Storage of RDF Data

Compressed Indexes

- It can help further compression of data in which sequential values occur often
- It stores values of bytes as differences (deltas) between sequential values, rather than values themselves



Storage of RDF Data
Compressed Indexes

0 for entry contains id triple
1 for the reference to next B+-tree leaf node

0 or 2bits for storing the number of bytes necessary to
store the difference of the three components and the
original representation of the object
00 = 1 Byte, 01 = 2 Bytes, 10 = 3 Bytes, 11 = 4 bytes

0

Header 6-12 Bits (1-2 Bytes)

0 to 4 Bytes
for subject

0 to 4 Bytes
for predicate

0 to 4 Bytes
for object

0 for original representation
of object must be stored
1 otherwise

How many leading components are the same in
comparsion to the last stored triple?
00 all components are different
01 one leading component is the same
10 two leading components are the same
11 no last triple

- It can help further compression of data in which sequential values occur often
- It stores values of bytes as differences (deltas) between sequential values, rather than values themselves

Storage of RDF data- Aggregated Indices

Index partial triples rather than full triples

Build aggregated indexes

- six addittional indexes (SP, PS, SO, OS, PO, OP)
- store only two out of the 3 columns of the triple
- store (value1, value2, count) instead of (value1, value2, value3)
- count used to produce the duplicates
- are organized in B+ -trees
- much smaller than full triple indexes

One value indexes (S, P, O) of the form (value1, count)

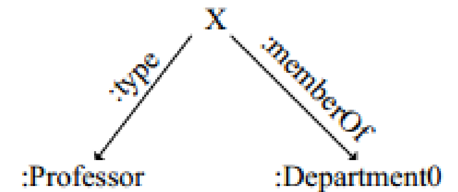
- rare but size very small

Examples:

1. select ?a ?c where {
   ?a ?b ?c }
2. (<a>,<b>,<C>) (<a>,<b>,<D>) aggregated as (<a>,<b>,2)

Translating SPARQL Queries

Construct query graph representation ⟶ suitable for later optimization



Query Graph

Each query can be parsed into a set of triple patterns

Each component either literal (mapped into ids) or variable

If query consists of:

- Single triple pattern
  Answer query with a singe range scan
- Multiple triple patterns
  Join results of individual patterns

Eliminate duplicates with DISTINCT option

Optimizing Join Ordering

Steps:

1. Build query graph
2. Fix query plan (most important issue in query processes) = Find the best join order
3. Execute selected query plan

Example:

```
select ?a where {
?a hasName "Robert".
?a hasGender "male".
?a receivedMedal "Purple Heart".
"Bob" nicknameOf ?a. }
```

Observation: If we first join those that are named Robert and won a Purple Heart, processing time reduced dramatically

- There are several kinds of plan in this query (different join order)

Which order gives the smallest query cost?

Selectivity Estimation

- ✓ give accurate cardinality of the joined triple pattern
- ✓ so we can compute the cost of the query plan precisely
- ✓ compute the cost for each one – choose smallest one as the final plan

Selectivity: ratio between the number of occurrences selected by an index and the size of the input data

Histograms

- ✓ Generic
- ✓ Can handle any triple patterns and joins

Frequent join paths in the data

- ✓ More accurate predictions for large joins

Selectivity Histograms

- similar data items are put in buckets
- buckets count the number of items they contain
- for each bucket (i.e. triple range) compute statistics
- space reduction achieved through fixed number of buckets

We use equi-depth histograms (good balance of items per bucket)

start (s,p,o)		end (s,p,o)	
number of triples			
number of distinct 2-prefixes			
number of distinct 1-prefixes			
join partners on subject			
	s=s	s=p	s=o
join partners on predicate			
	p=s	p=p	p=o
join partners on object			
	o=s	o=p	o=o

Estimate cardinality of single triple pattern (number of triples, number of distinct 2-prefixes, number of distinct 1-prefixes)

Estimate result cardinality (join partners on subject, predicate, object)

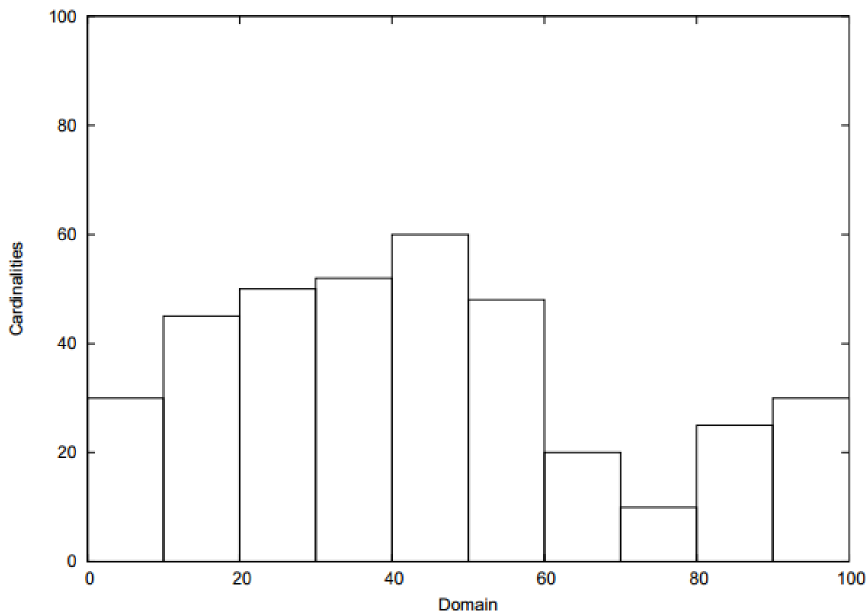
Selectivity Histograms

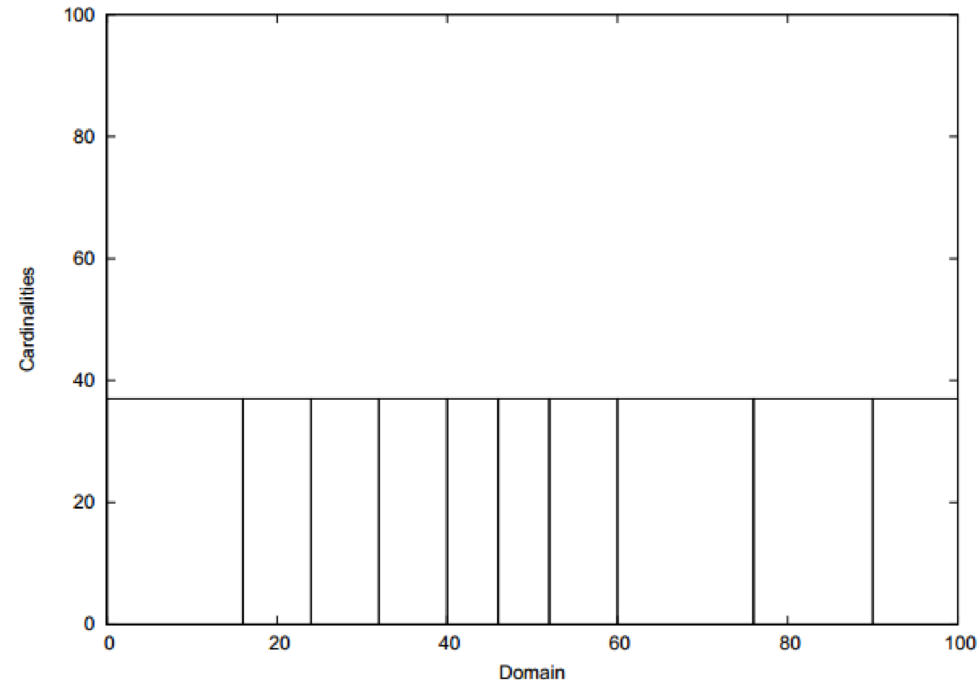
The selectivity of an object *O given* predicate *P can then be estimated as* $\frac{hc(P,Oc)}{|\mathrm{T}p|}$ *where*

hc: height of histogram class

Oc : equi-depth classes of valid values per predicate

Tp: is the (exact) number of triples matching predicate *P*





Weakness:
they assume independence between predicates

Frequent Paths

Computes frequent join paths

In SPARQL following predicates occur commonly:
navigation: {(?a,[],?b),(?b,[],?c),(?c,[],?d)} (chain)
selection: {(?a,[],?b),(?a,[],?c),(?a,[],?d)} (star)

Compute the most frequent paths:
Paths with the largest cardinalities
Materialize result cardinalities and path descriptions (predicates p1, …, pn)

- Not as easily applicable as histograms, but very accurate

Optimization

Cost-based Query Planning

Hardest problems in query optimization is to estimate the costs of query plans

- ✓ Optimizers cost query plans - estimate selectivities
- ✓ Choose smallest cost plan as the final one

Evaluation

RDF-3X compared with:

- MonetDB (column store approach)
- PostgreSQL (triple store approach)

Following datasets:

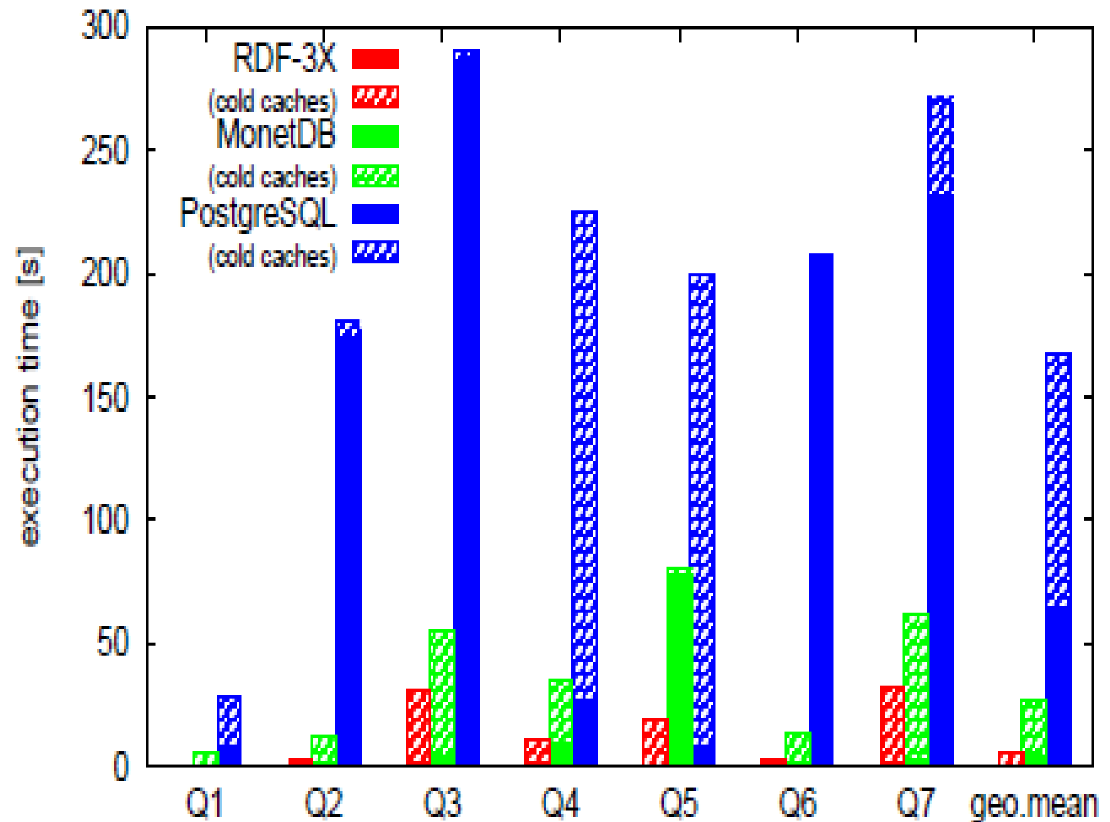
- Barton (library data)
- Yago (Wikipedia)
- LibraryThing

Equivalent Queries (SPARQL for RDF-3X, SQL for others)

Evaluation - Barton

sample query (Q5)

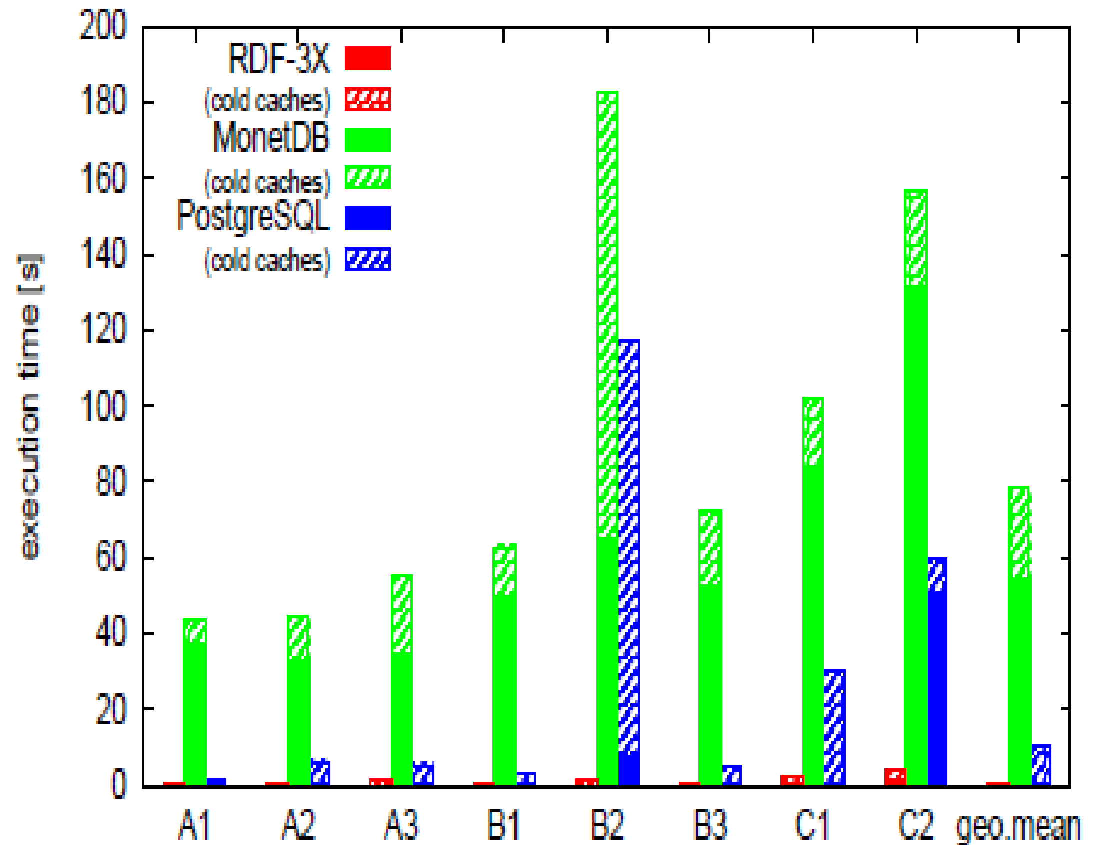
select ?a ?c where
{ ?a <origin> <marcorg/DLC>.
?a <records> ?b.
?b <type >?c.
filter (?c != <Text>) }



Evaluation - Yago

sample query(B2)

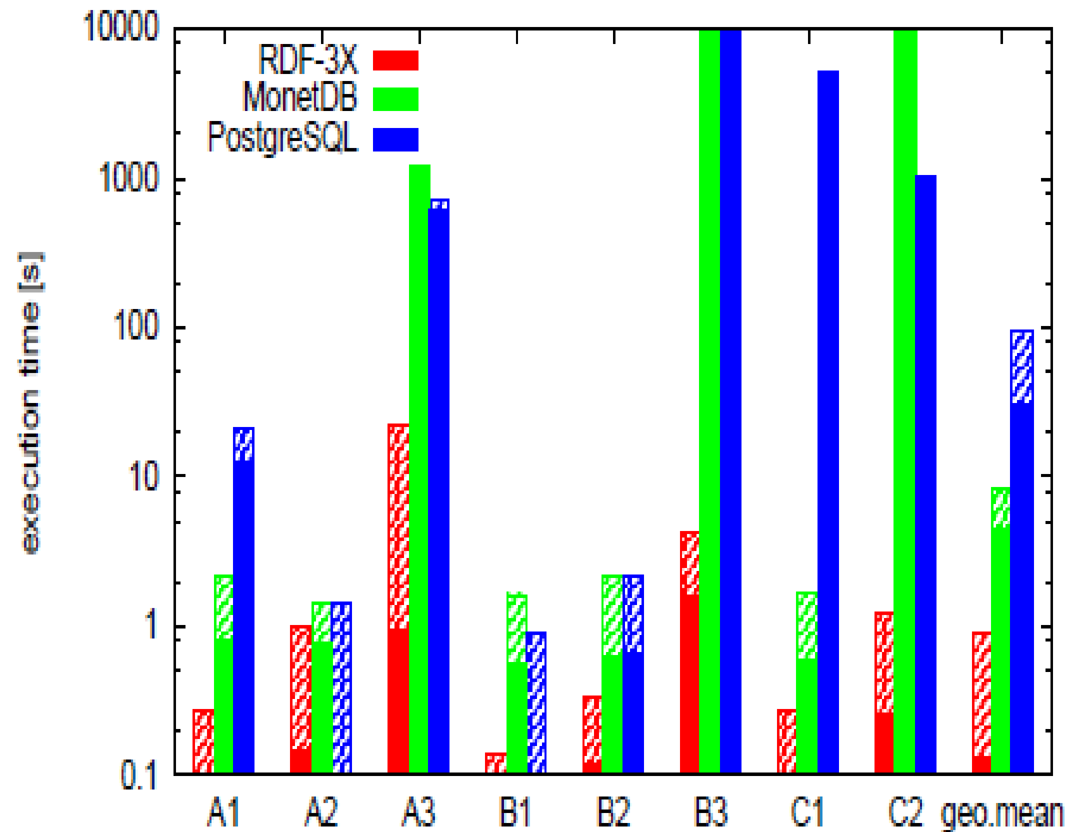
select ?n1 ?n2 where {
?p1 <isCalled> ?n1.
?p1 <bornInLocation> ?city.
?p1 <isMarriedTo> ?p2.
?p2 <isCalled> ?n2.
?p2 <bornInLocation> ?city }



Evaluation – Library Thing

sample query(B3)

select distinct ?u where {
?u [] ?b1.
?u [] ?b2.
?u [] ?b3.
?b1 [] <german> .
?b2 [] <french> .
?b3 [] <english>}



Scalable Join Processing

Very Large RDF Graphs are challenging:

- even simple scans can become too expensive
- only combinations of patterns are selective
- predicting the selectivity is hard

We improve join processing by two main techniques:

- sideways-information-passing drastically reduces scans costs (during both merge-joins and hash-joins)
- detailed join cardinalities of constants joined with the whole graph help a lot

RDF-3X : Probably fastest system between RDF engines

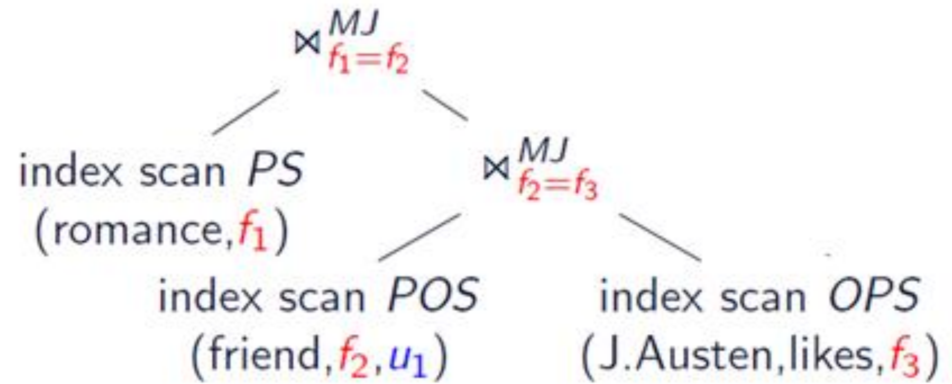
No matter how good optimizer-generated plan is ⟶ joins remain a costly operation

Sideways Information Passing
- ✓ Allows jumping over big gaps by accessing interior nodes of B+ -trees
- ✓ Avoids searching along a possibly long chain of leaves
- ✓ Improves selectivity estimates

✓ It passes information from one operand to the other sideways in the operator-graph

SIP – Example

- PS proceeds in parallel with merge of POS and OPS scans
- Sends valuable information to them
- Large parts of indexes are skipped



P	S
...	
2100	1003
2100	1119
2100	1127
2100	1255
2100	1266
2100	1418
2100	1429
2100	1683
...	

P	O	S
...		
2015	3007	1011
2015	3007	1055
2015	3007	1119
2015	3007	1127
2015	3007	1135
2015	3007	1339
2015	3007	1348
2015	3007	1418
...		

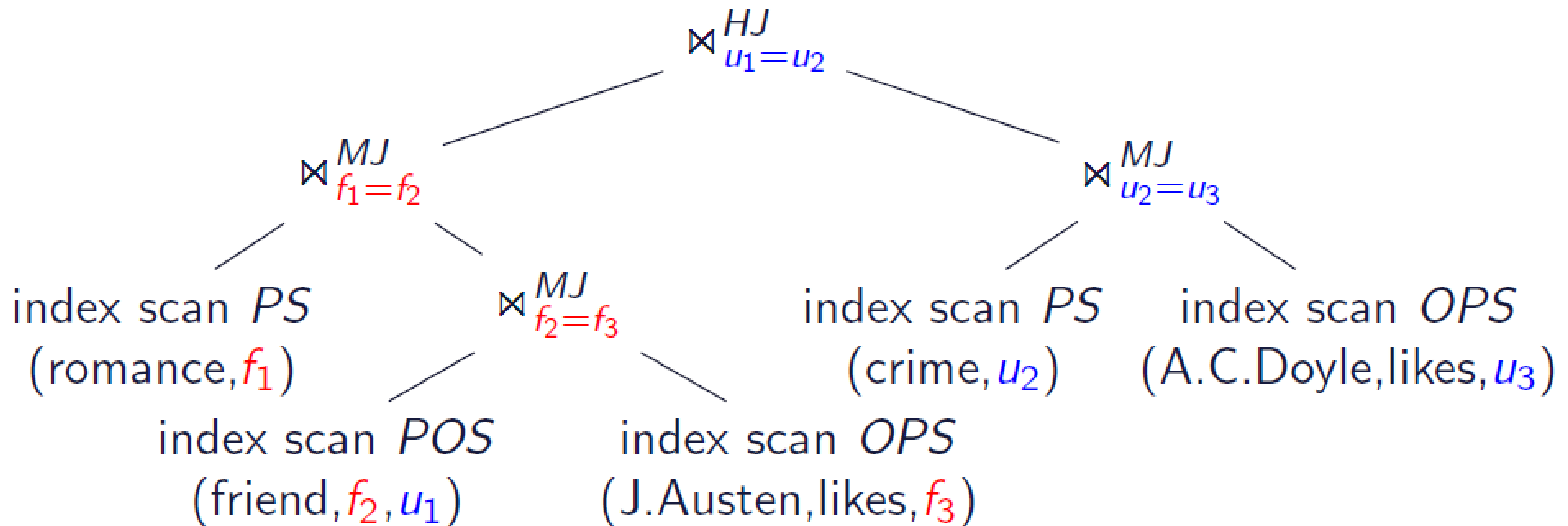
O	P	S
...		
3111	2003	1019
3111	2003	1055
3111	2003	1111
3111	2003	1127
3111	2003	1135
3111	2003	1199
3111	2003	1227
3111	2003	1243
...		

SIP

SIP

SIP – Execution Plan Example

select ?u where { ?u <crime> []. ?u <likes> "A.C.Doyle“. ?u<friend> ?f.
?f <romance> []. ?f <likes> "J.Austen". }

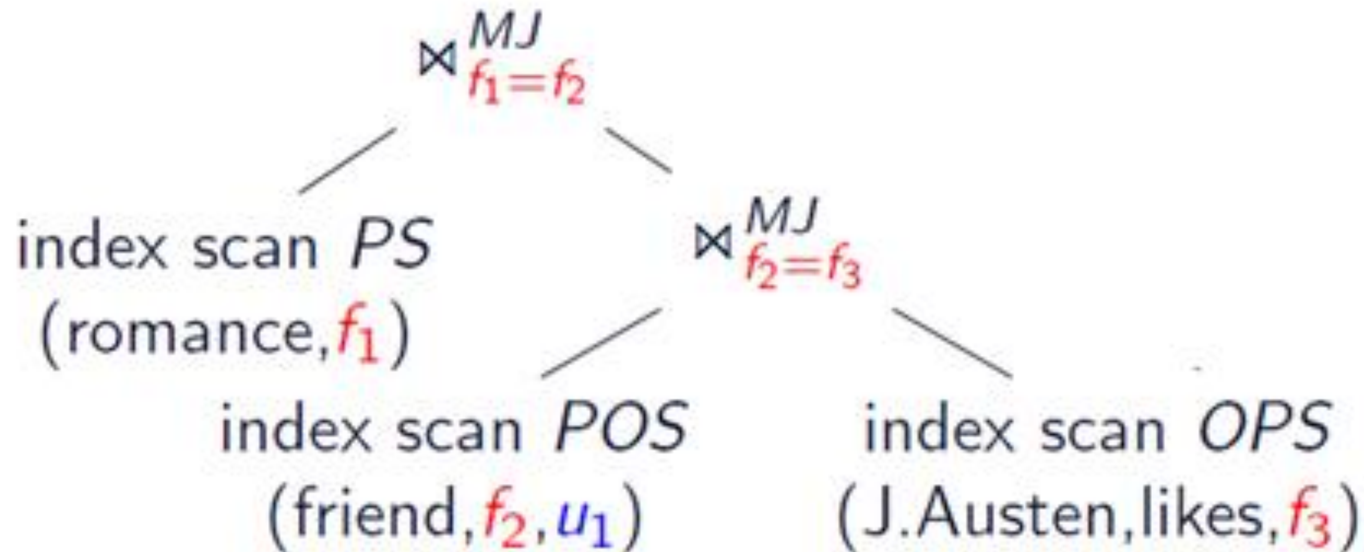


Scans can be restricted by SIP

SIP on Merge Joins

Merge Joins read their input in ascending order

- If a solution A read from one of the operands
- Pass A to the other operand
- The other optimizes retrieving a solution equal to or larger than A



provides additional scan constraints

f1 >= f2

f2 >= max(f1, f3)

f3 >= max(f1, f2)

- When data are stored using B+ trees, the larger value L can be used as key for directly finding the wanted leaf
- The index scans skip tuples that cannot make it into the result

SIP on Hash Joins

Use of Bloom filters to reduce index data to a fixed size

Bloom filter : data structure that tell whether an element is present in a set

Initially:
Potential domain (0, ∞, 1*) as every value is valid
Observed domain (∞, 0, 0*) as no values have been observed yet

potential domain x

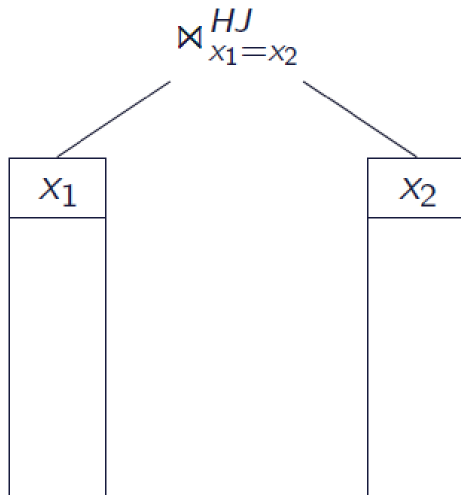
0	∞	1111111

observed domain x_1

0	0	0000000

min	max	Bloom filter (1024 bytes)

SIP on Hash Joins



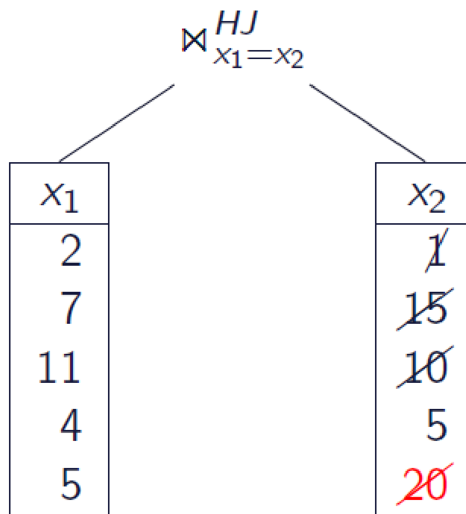
potential domain x

0	∞	1111111

observed domain x_1

0	0	0000000

hash function in example: $v \mod 7$



potential domain x

2	11	1010110

observed domain x_1

2	11	1010110

hash function in example: $v \mod 7$

Run-Time Acceleration of Index Scans

Index scans use scan constraints and domain filters to skip entries in the tree

But how often is it good to proceed a scan?
Scan for every triple ⟶ enormous CPU costs
So, scan when reading a new page

First entry on the pages list = candidate binding

- If not found in the page , skip it
- If found, scan there

Observation: Can reduce the CPU costs to nearly zero
Time saved by not reading whole pages ⟶ significant

Selectivity Estimation
Improving Join Ordering

Even with SIP, join ordering is very important

- bad join orders can lead to huge intermediate results
- SIP reduces this, but cannot eliminate the problem

RDF-3X uses a typical estimation → histograms

But this does not scale:

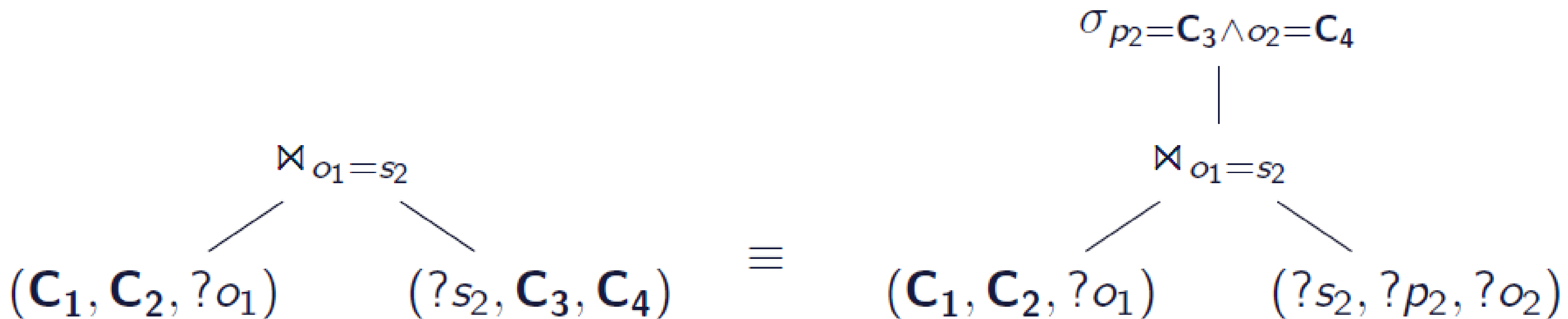
- Histogram grows with the data
- Prediction quality degrades

Use of Additional B+-trees to organize estimation data

Selectivity Estimation
Joins Between Triples

We transform:
join between two triple patterns in form easier to estimate

Join of two triple patterns interpreted as:
self join of one triple pattern with all its triples in the database and a final selection
JoinEdge Selectivity = FirstTripleNodeSelectivity * SecondTripleNodeSelectivity



- Compute selectivity by finding the cardinality of the second triple pattern
- Join selectivity: selectivity of one triple pattern joined with all other triples
- Compute join selectivities for all possible choices of one or two constants in triple pattern
- Materialize result in additional indexes

Evaluation

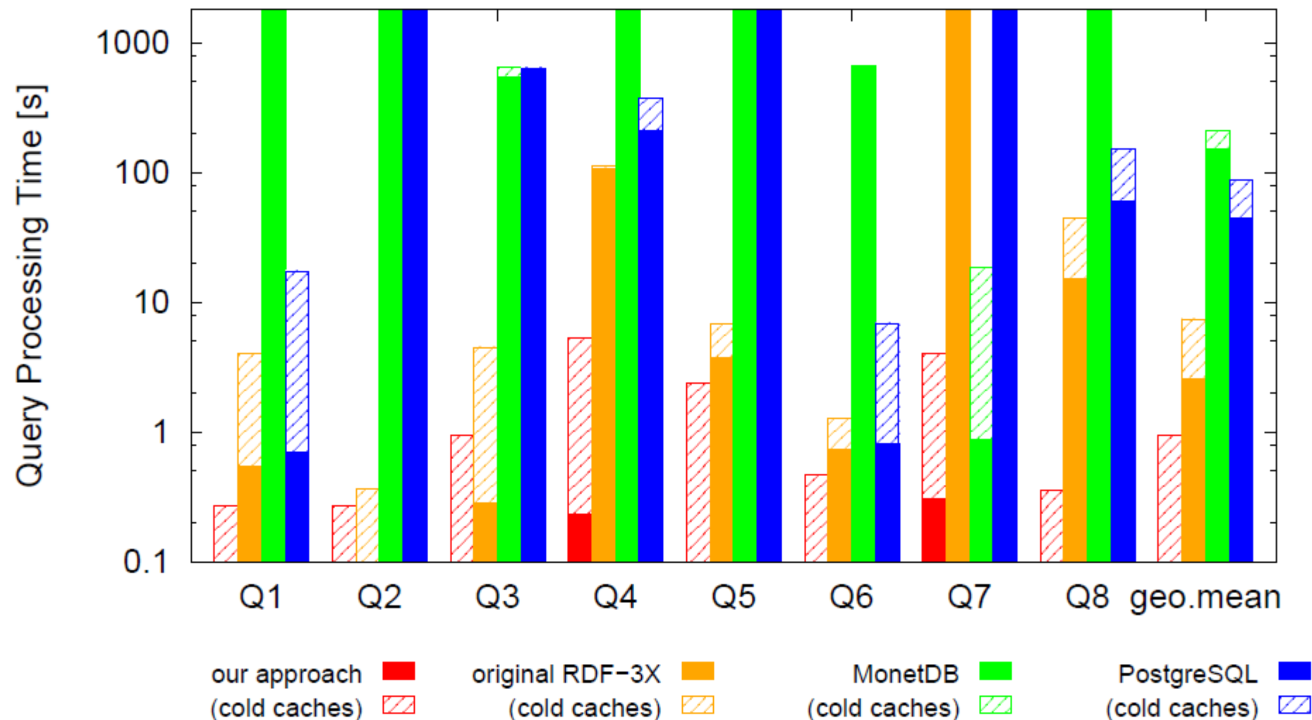
Following systems were compared:

- our approach
- RDF-3X
- MonetDB
- PostgreSQL

Datasets that we used:

- Billion Triples Challenge (88GB)
- UniProt (57GB)

Evaluation – Billion Triples Challenge



Q1: select ?lat ?long where {
?a [] "Eiel Tower". ?a geo:ontology#inCountry geo:countries/#FR.
?a pos:lat ?lat. ?a pos:long ?long. }

- Results for Uniplot dataset were similar

References

- T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. PVLDB, 1(1):647{659, 2008.
- J. Broekstra et al. Sesame: An architecture for storing and querying rdf data and schema information. In Spinning the Semantic Web, 2003.
- L. Baolin and H. Bo. Path queries based rdf index. In SKG, 2005.
- D. E. Simmen, E. J. Shekita, and T. Malkemus.Fundamental techniques for order optimization. In SIGMOD, 1996.
- Hai Huang, Chengfei Liu. Estimating Selectivity for Joined RDF Triple Patterns. Swinburne University of Technology
- Abraham Bernstein, Markus Stocker, and Christoph Kiefer, SPARQL Query Optimization Using Selectivity Estimation, 2006