max planck institut
informatik

Scalable Join Processing on Very Large RDF Graphs

Thomas Neumann Gerhard Weikum

Max-Planck-Institut Informatik

July 2, 2009

RDF and SPARQL

- RDF is conceptually a labeled graph
- edges form (subject,predicate,object) triples

```
<Id1> <hasName> "Jean-Marie Le Clézio".
<Id1> <hasNationality> "French".
<Id1> <hasWritten> <Id2>.
<Id2> <hasTitle> "Désert".
```

- SPARQL queries consist of sets of triple patterns
- query-by-example style, joins are implicit

```
SELECT ?y WHERE { ?x <hasName> ?y.
?x <hasNationality> "French".
?x <hasProfession> <novelist>.  }
```

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

RDF Storage and Retrieval

RDF and SPARQL think in terms of triples

- data is one large set of triples
- triple patterns in SPARQL filter specific triples
- no schema, everything is in one "relation"
- queries make extensive use of self-joins

RDF-3X uses extensive index structures for efficient processing

- dictionary compression to reduce size, results in integer triples
- 3 attributes $\Rightarrow$ 3!=6 permutations $\Rightarrow$ 6 compressed clustered B^+-trees

Each triple pattern is answered by **a single index range scan**.
Uses **merge-joins** as much as possible, then hash-joins.

SPARQL on Very Large RDF Graphs

Problem: individual triple patterns tend to be very **unselective**

```
?x <hasName> ?y
```

- has millions of possible bindings
- expensive to compute even in isolation

Even more extreme: patterns without contants

```
?s ?p ?o
```

- basically a full table scan of the whole RDF graph
- some graphs are really large (UniProt 57GB, nearly a billion triples)
- unacceptable execution times

For many queries only the **combination of patterns** is selective.

Our Approach

Efficient SPARQL processing on very large graphs:

- execution time should depend on result size, not data size
- unfortunately not feasible in general, but should be the goal
- exploit pattern combinations, avoid touching irrelevant parts of the graph
- we address this using **holistic sideways-information-passing** (SIP)

Related problem: predicting join selectivity

- required for join ordering, heavily affects execution times
- becomes increasingly difficult for large graphs
- we **precompute join cardinalities** for literal combinations

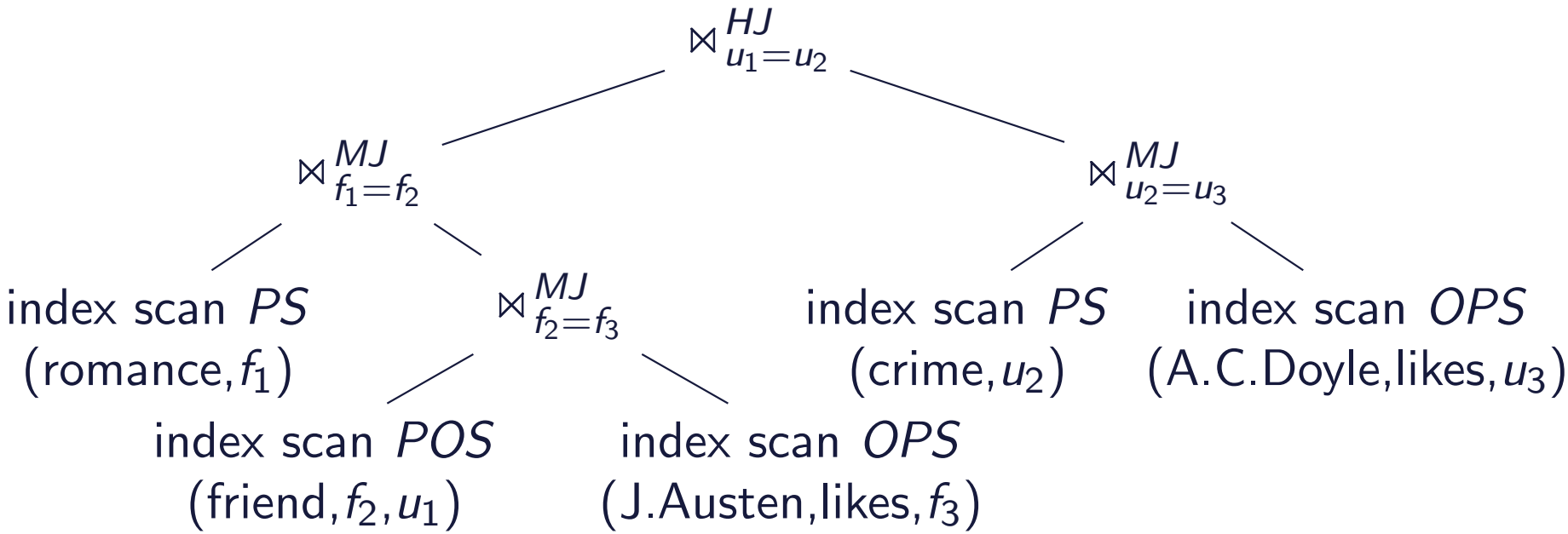
Implemented as improvements in an existing RDF engine (RDF-3X)

Overview

1. Motivation
2. **Improving Join Execution on Very Large RDF Graphs**
3. Improving Join Ordering on Very Large RDF Graphs
4. Evaluation
5. Conclusion

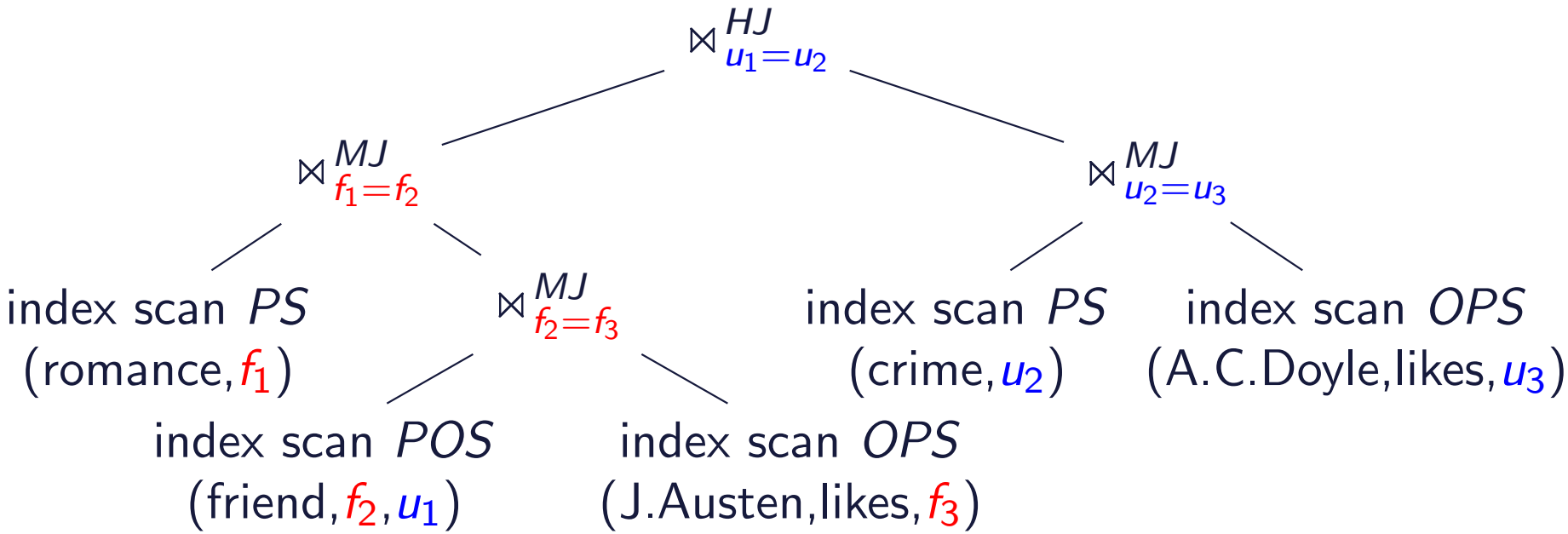
A typical execution plan

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



A typical execution plan

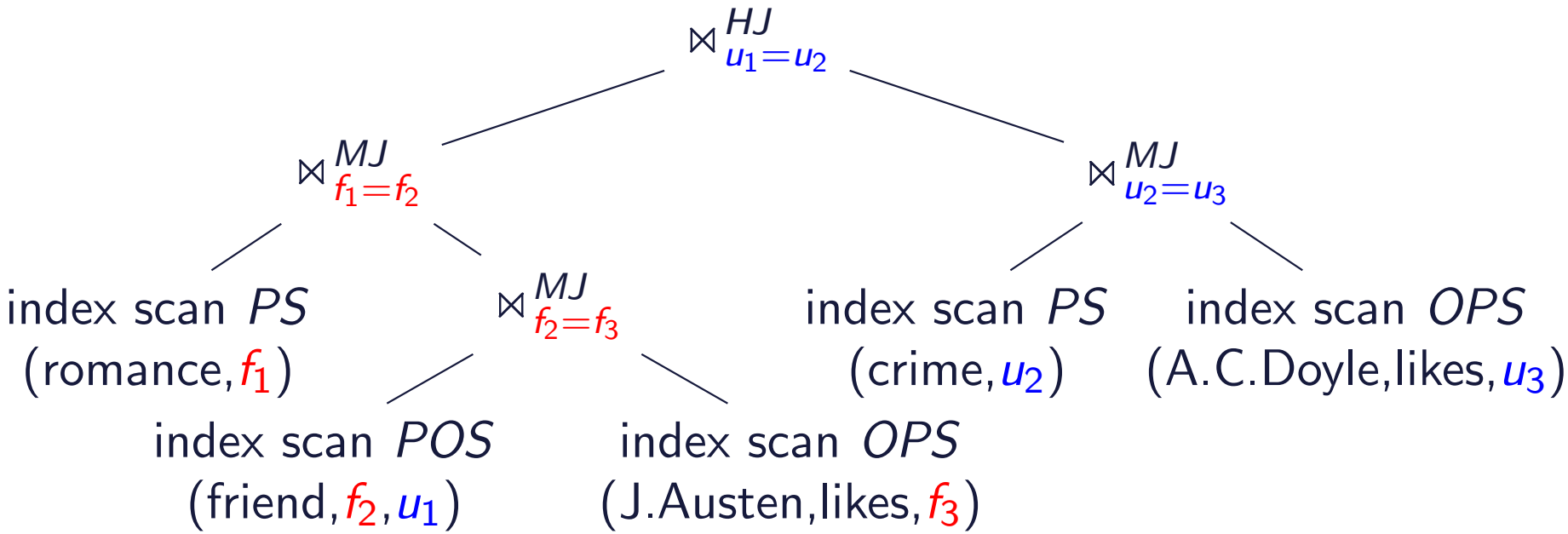
```
select ?u where { ?u <crime> [].  ?u <likes> "A.C.Doyle";
?u <friend> ?f.  ?f <romance> [].  ?f <likes> "J.Austen".  }
```



- attributes form equivalence classes

A typical execution plan

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

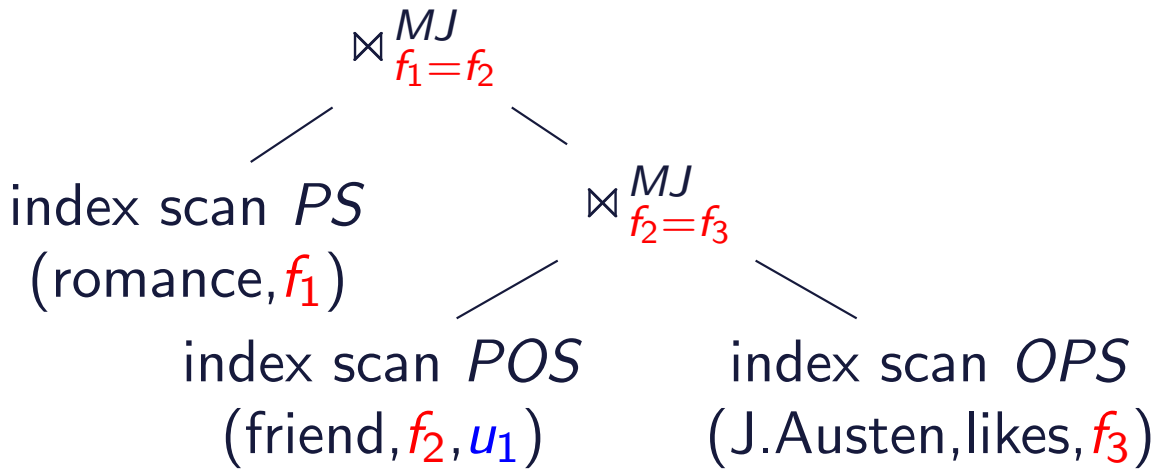


- attributes form equivalence classes
- can restrict scans by **sideways-information-passing** (SIP) between attributes. E.g., f_3 is affected by f_1 (indirectly)

SIP during Merge-Joins

Merge-Joins consume tuples in monotonic increasing order

- provides additional scan constraints
- $f_1 \geq f_2$
- $f_2 \geq max(f_1, f_3)$
- $f_3 \geq max(f_1, f_2)$



In general:
Compare with all equivalent attributes in join conditions of ancestor $\bowtie^{MJ}$.

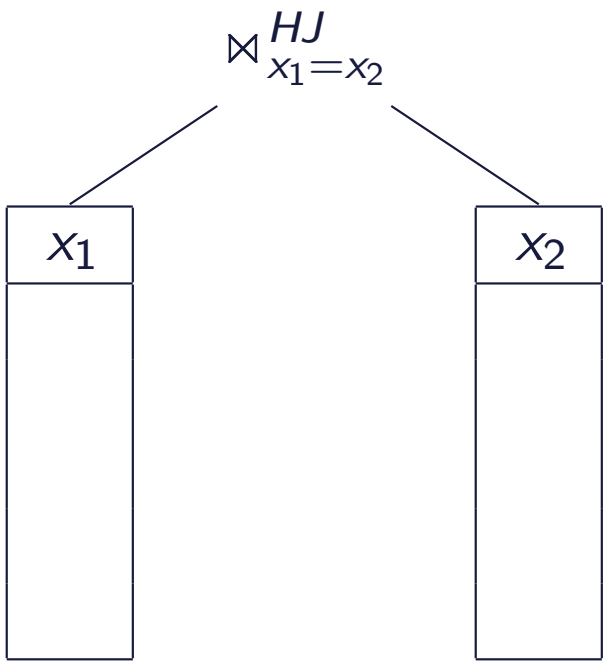
- the index scans can **skip** tuples that cannot make it into the result
- note we use a **holistic** SIP approach: there is no direction of data flow, everything prunes everything

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

0	∞	1111111

observed domain x_1

0	0	0000000

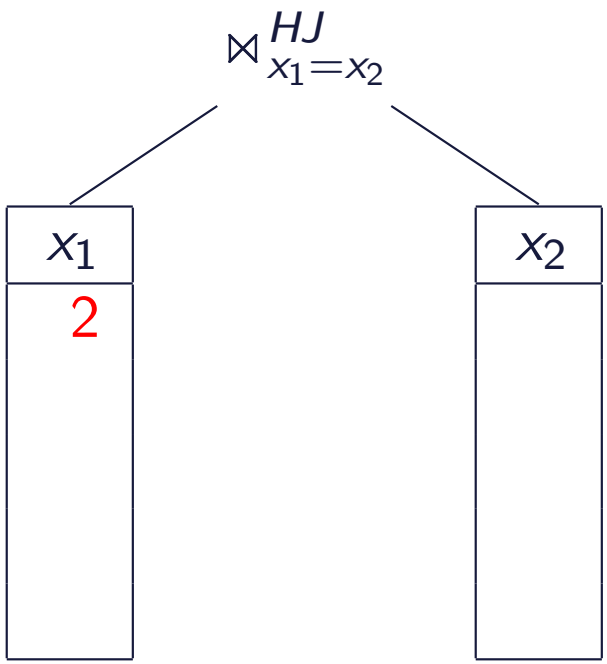
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

0	∞	1111111

observed domain x_1

2	2	0010000

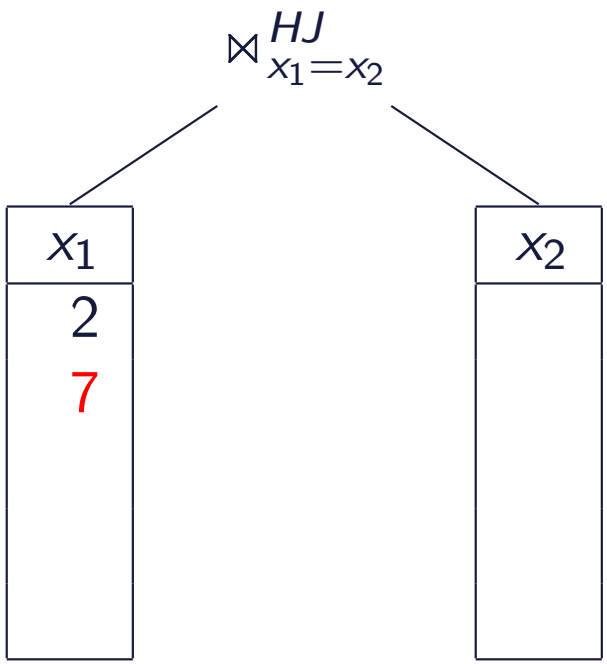
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

0	∞	1111111

observed domain x_1

2	7	1010000

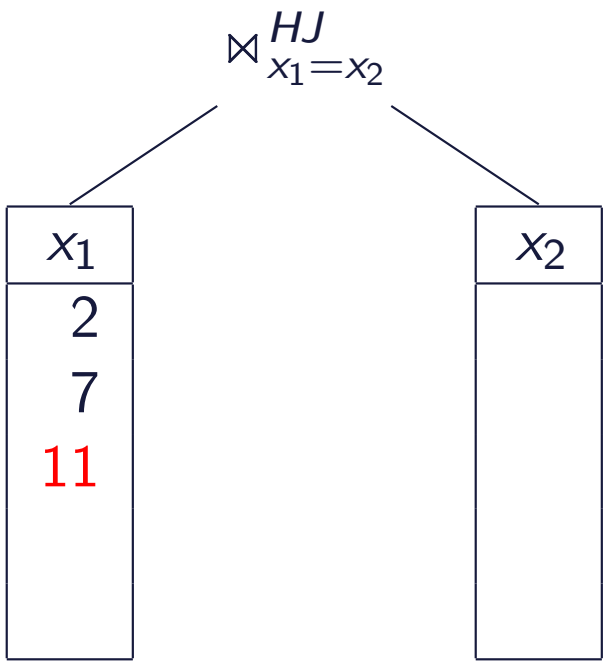
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

0	∞	1111111

observed domain x_1

2	11	1010100

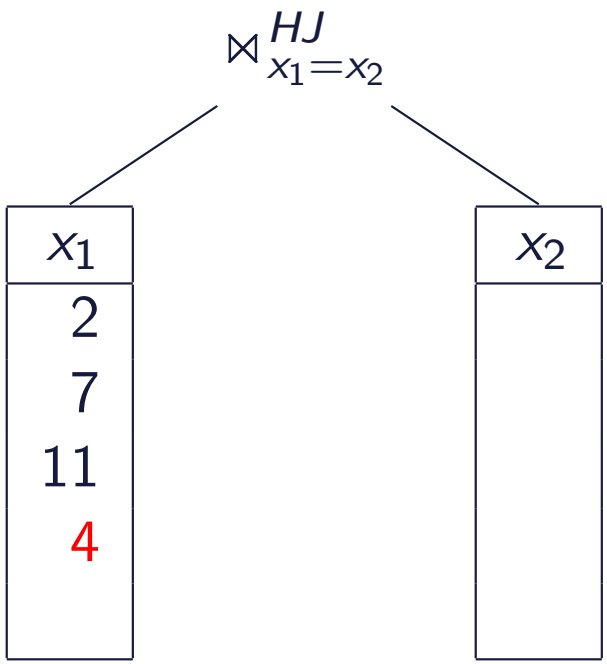
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

0	∞	1111111

observed domain x_1

2	11	1010100

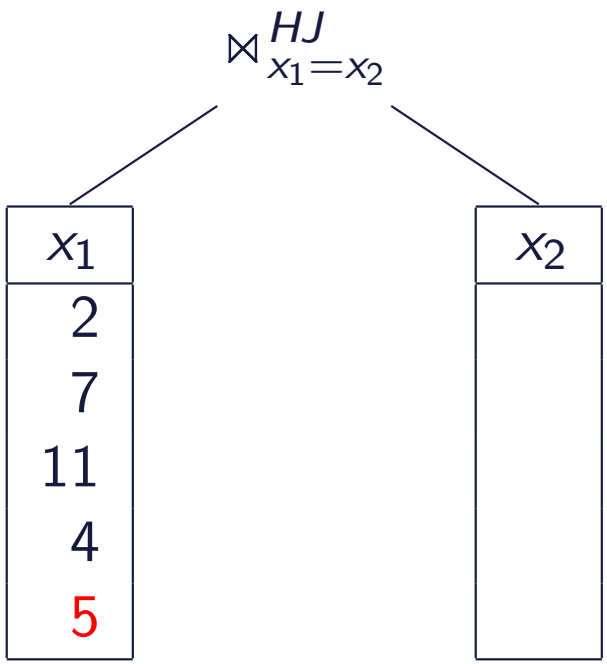
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

0	∞	1111111

observed domain x_1

2	11	1010110

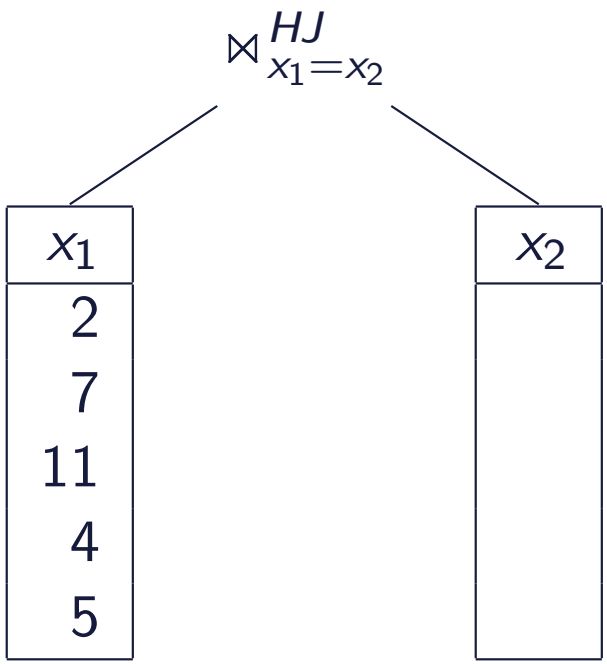
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

2	11	1010110

observed domain x_1

2	11	1010110

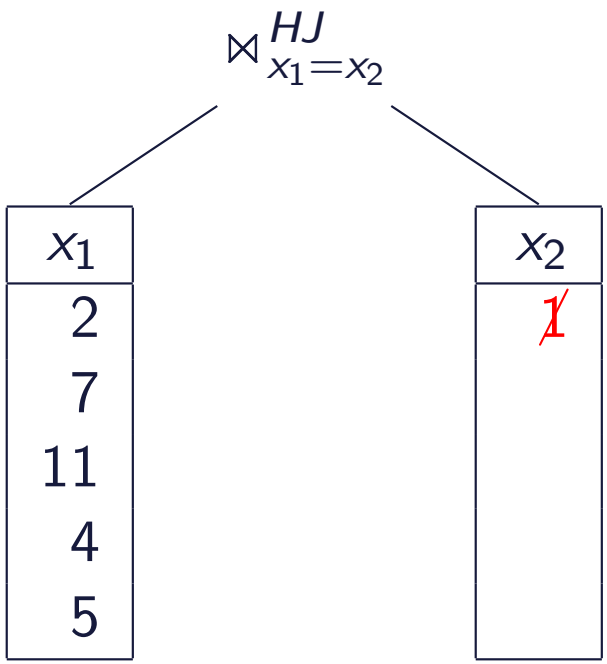
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

2	11	1010110

observed domain x_1

2	11	1010110

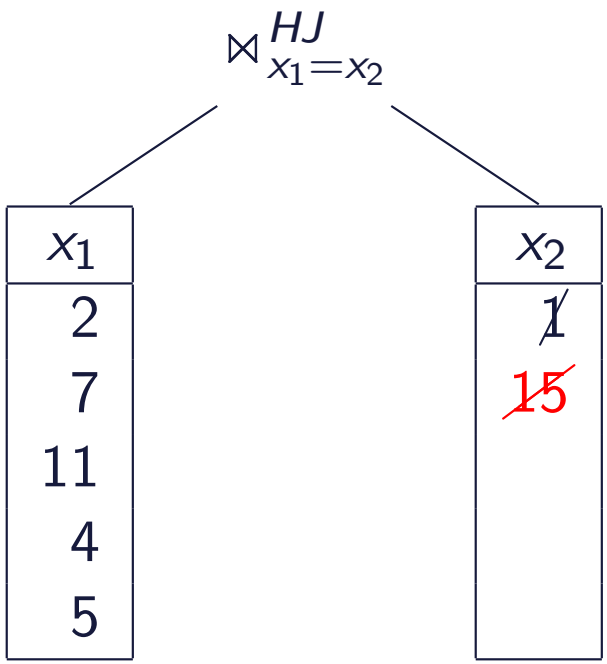
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

2	11	1010110

observed domain x_1

2	11	1010110

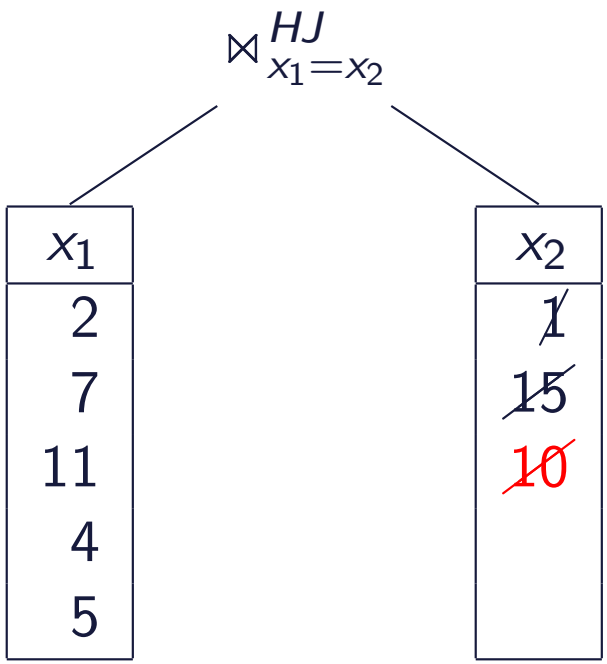
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

2	11	1010110

observed domain x_1

2	11	1010110

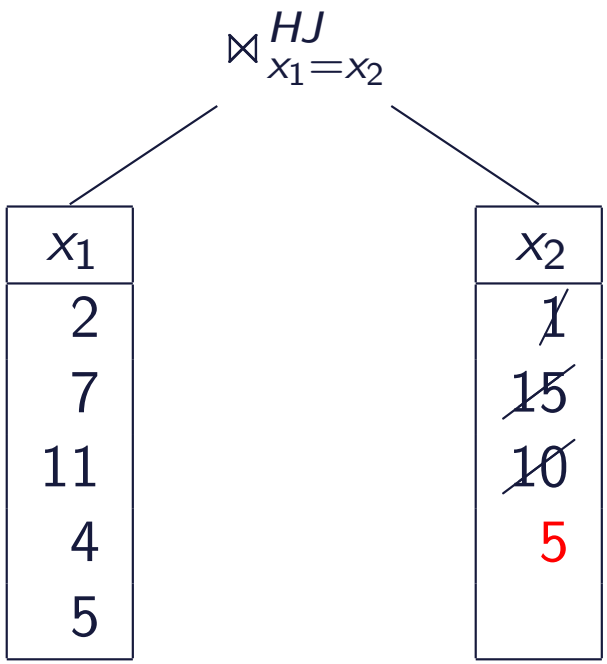
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

2	11	1010110

observed domain x_1

2	11	1010110

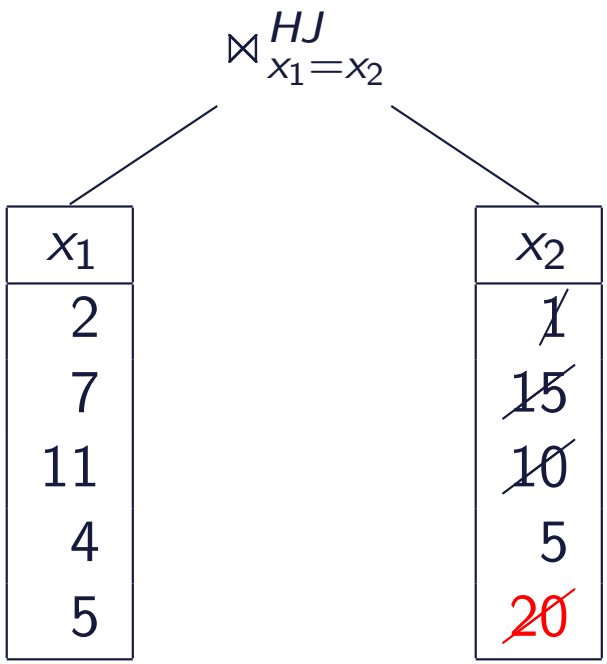
hash function in example: $v \mod 7$

SIP during Hash-Joins

Hash-Joins are pipeline-breakers, no direct attribute comparisons. Instead, build **domain filters** for observed attribute values:

min	max	Bloom filter (1024 bytes)

Each equivalence class has **potential domain** filter, each hash-join build constructs an **observed domain** filter and intersects with the potential domain when done.



potential domain x

2	11	1010110

observed domain x_1

2	11	1010110

hash function in example: $v \mod 7$

Integrating SIP into Scans

Index scans use the scan constraints and domain filters to skip tuples.

- could check for every tuple, but CPU costs are high
- instead, check on each page transition

On a new page search for the first potentially qualifying tuple

- if one is found, start scanning there
- if none on this page, skip using the B^+-tree

Frequently reduces the tuple reads to 10% or less.

Overview

1. Motivation
2. Improving Join Execution on Very Large RDF Graphs
3. **Improving Join Ordering on Very Large RDF Graphs**
4. Evaluation
5. Conclusion

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
- small intermediate results lead to better domain filters

Join Order is computed using Dynamic Programming, but relies upon statistical information:

- needs selectivity/cardinality estimates to order operations
- primarily two values: scan size and join selectivity

Misestimations can lead to very poor plans.

Selectivity Estimators for Large Graphs

Originally RDF-3X used a very typical estimation approach:

- histograms over value distributions for estimations
- one database page per histogram

but this does not scale

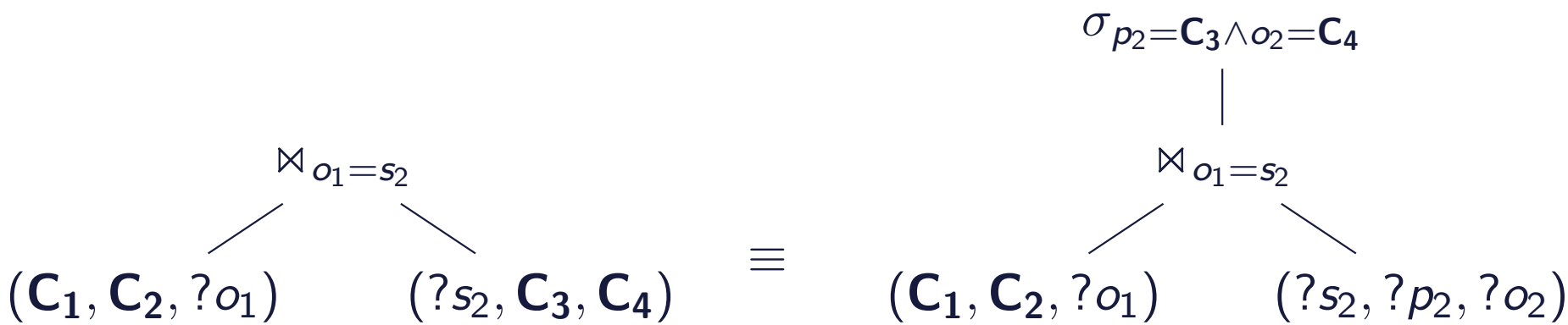
- as the graph grows, the prediction quality degrades
- the histogram must **grow with the data**
- this means accepting page faults for selectivity lookup

We use additional B^+-trees to organize estimation data.

Modelling Join Selectivities

- RDF stores all data items in one big "relation"
- very heterogeneous, join selectivities heavily depend on constants
- we cannot afford multi-dimensional histograms

But we can use this:



We model a join of two scans (with constants) as a join of a scan with all triples plus a selection.

Capturing the Join Selectivity

We compute the selectivity of a scan S joined with all triples R:

1. no constants in S
   precompute $|R \bowtie_{x=y} R|$ for $x, y \in \{s, p, o\}$
2. three constants in S
   use constant folding
3. one constant in S, e.g., in p
   precompute $|(?s_1, C_P, ?o_1) \bowtie_{x=y} R|$ for all values of C_P and $x \in \{s, o\}, y \in \{s, p, o\}$
4. two constants in S, e.g., in s and p
   precompute $|(C_S, C_P, ?o_1) \bowtie_{o_1=y} R|$ for all values of C_S, C_P and $y \in \{s, p, o\}$

Gives accurate cardinality of scan+join. Additional selection estimated assuming independence.

Main problem: space consumption

Storing the Join Cardinalities

We have 6/3 cardinalities per constant/constant pair. Store them in B^+-trees.

We use leaf compression to reduce the data size:

c_1	$c_2 - c_1$	$c_3 - c_2$	...	$c_1 \bowtie_{S=S}$	$c_2 \bowtie_{S=S}$	...	$c_1 \bowtie_{S=O}$	...

- frequently results in very regular patterns
- put LZ77 compression on top to reduce the data
- overall space increase $\approx 10\%$

Selectivity lookup might produce a page fault. But query processing is expensive anyway.

Overview

1. Motivation
2. Improving Join Execution on Very Large RDF Graphs
3. Improving Join Ordering on Very Large RDF Graphs
4. **Evaluation**
5. Conclusion

Evaluation

Compared four different systems:

- original RDF-3X [VLDB08]
- RDF-3X plus SIP and estimations
- PostgreSQL as classical triple store
- MonetDB as vertical partitioning approach [Abadi et al., VLDB07]

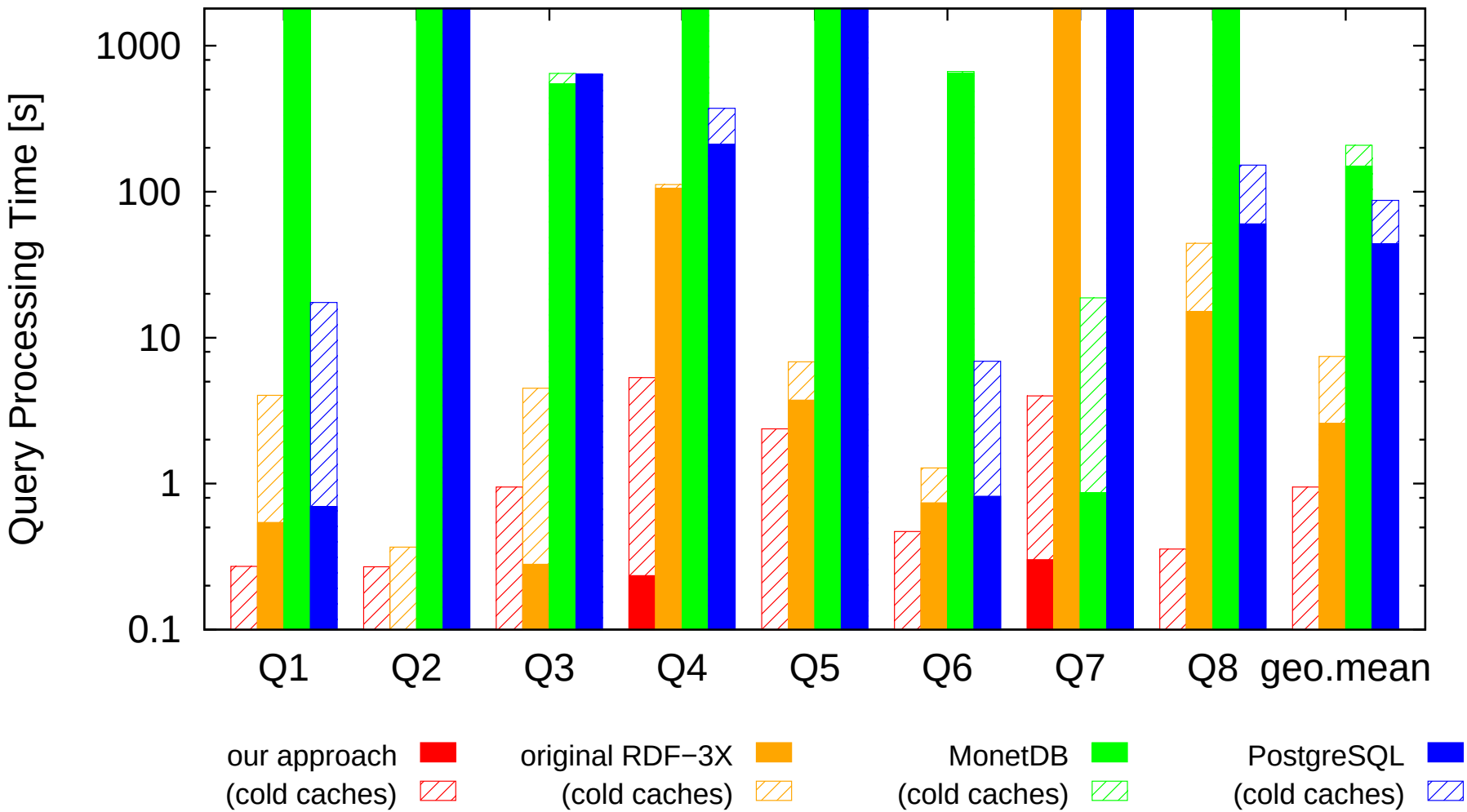
running on a laptop with 2GB main memory.

Two very large data sets:

- Billion Triples Challenge (88GB)
- UniProt protein information (57GB) (see the paper)

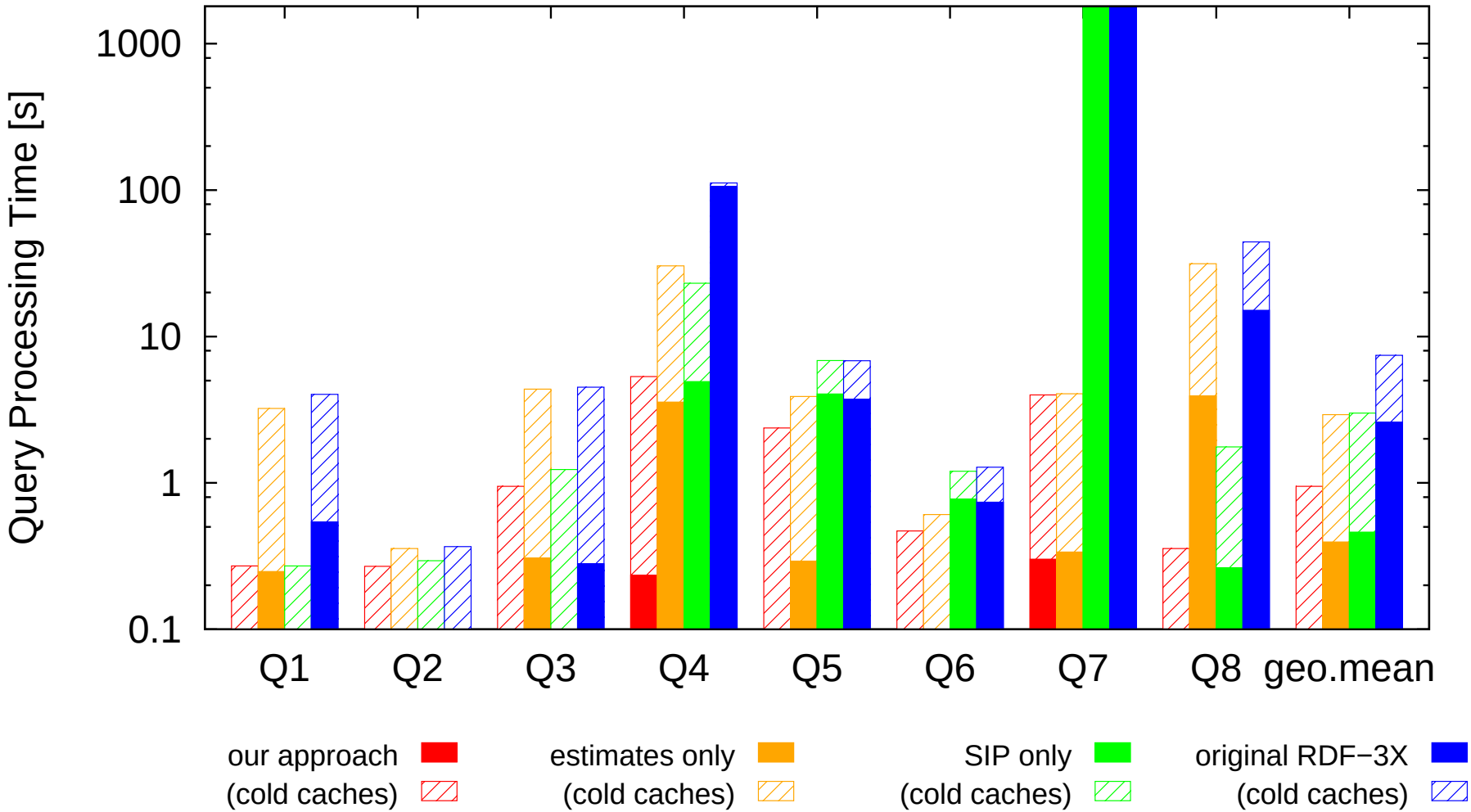
We measured both cold-cache and warm-cache execution times.

Evaluation - Billion Triples Challenge



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

Evaluation - Effect of Individual Techniques



Billion Triples Challenge data set, results for UniProt are similar.

Conclusion

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 improved 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 is open source `http://www.mpi-inf.mpg.de/~neumann/rdf3x`

We participated in the ACM SIGMOD 2009 Repeatability & Workability Evaluation (cf., `http://homepages.cwi.nl/~manegold/SIGMOD-2009-RWE/`).
The reviewers were able to repeat all the experiments presented in our paper, yielding results that match the ones published in our paper, except from insignificant and to be expected variation due to randomness and/or hardware/software differences.
The detailed reports will shortly be made publicly available by ACM SIGMOD.