max planck institut
informatik

RDF-3X: a RISC-style Engine for RDF

Thomas Neumann Gerhard Weikum

August 27, 2008

Motivation

RDF is increasingly popular...

- Semantic Web
- Life-Sciences
- seems natural for Social-Networks

... but RDF indexing and query processing is non-trivial:

- no schema, very fine grained data items
- workloads hard to predict and characterize
- physical design difficult

Our solution: RDF-3X

- RISC-style execution engine
- exhaustive compressed indexes
- query optimization techniques

Overview

1. Short Introduction to RDF and SPARQL
2. Storage of RDF data
3. Retrieval for SPARQL queries
4. Evaluation
5. Conclusion

Short Introduction to RDF and SPARQL

RDF: Resource Description Framework

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

Examples:

- (id_1, <hasTitle>, "Sweeney Todd"),
- (id_1, <directedBy>, <Tim_Burton>),
- (id_1, <hasCasting>, id_2)
- (id_2, <Actor>, id_{11})
- (id_{11}, <hasName>, "Johnny Depp")

RDF data can be seen as a (potentially huge) set of triples.

Short Introduction to RDF and SPARQL (2)

SPARQL: SPARQL Protocol and RDF Query Language

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

- queries RDF data by matching patterns in the graph
- query-by-example style, joins are implicit
- set of triple patterns, shortcuts to avoid typing

Note: must produce all valid bindings (might create duplicates)

Storage of RDF data

Raw RDF input: triples

Facts		
Subject	Predicate	Object
object214	hasColor	blue
object214	belongsTo	object352
...	...	...

Literals can be very large, contains a lot of redundancy.
First step to reduce the data: dictionary compression

Facts		
Subject	Predicate	Object
0	1	2
0	3	4
...	...	...

Strings	
ID	Value
0	object214
1	hasColor
...	...

Storage of RDF data - RDF-3X

Our approach: Store everything in a clustered B^+-Tree

- triples sorted in lexicographical order
- can be **compressed** well (delta encoding)
- efficient scan, fast lookup if prefix is known

Which sort order to choose?

- index is compressed, we can afford redundancy
- 6 possible orderings, store all of them
- will make merge joins very convenient

Observation: Each SPARQL triple pattern can be answered by a single range scan.

Storage of RDF data - Aggregated Indices

Sometimes we do not need the full triple:

- is there a connection between $object_4$ and $object_{13}$?
- how many *author* annotations does $object_{14}$ have?

Therefore maintain **aggregated indexes** with $(value_1, value_2, count)$

- *count* is required for SPARQL duplicate semantic
- compressed, too
- much smaller than the full index

We can afford another 6 indexes. And three for $(value_1, count)$.

- smaller index $\Rightarrow$ faster scan
- improves query performance significantly

Retrieval of RDF data

- each SPARQL triple pattern becomes an index scan
- patterns with common variables induce joins
- indexes for all orderings, which makes merge joins very attractive

basic strategy: merge joins if possible, hash joins afterwards

- decision cost based, dynamic programming strategy
- order optimization required to infer orderings

A bit different from standard join ordering, though:

- one big "relation", no schema
- selectivity estimates are hard

Retrieval of RDF data - Selectivity Estimates

Standard single attribute synopses are not very useful:

- only three attributes
- one big "relation"
- but `(?a,?b,"Auckland")` and `(?a,?b,"1900-01-01")` produce vastly different values for `?a` and `?b`

Instead: Another six indexes

- aggregate indexes until they fit into one page
- merge smallest buckets ($\approx$ equi-depth)
- for each bucket (i.e., triple range) compute statistics
- 6 indexes, pick the best for each triple pattern

Retrieval of RDF data - Selectivity Estimates (2)

Example: bucket with (subject,predicate,object) statistics

range	(10,2,30) - (10,5,12000)		
	1	2	3
# prefixes of length	1	3	3000
	subject	predicate	object
# subject joins with	4000	0	200
# predicate joins with	50	400000	200
# object joins with	6000	0	9000

Estimations:

- `(10,4,?a)` $\Rightarrow$ 1000 triples
- $\{$`(10,4,?a),(?a,?b,?c)`$\}$ $\Rightarrow$ 2000 triples

Assumes uniformity, independence, etc., but works quite well

Retrieval of RDF data - Selectivity Estimates (3)

Still issues with (common) large correlated join patterns:

- navigation: {(?a,[],?b),(?b,[],?c),(?c,[],?d)} (chain)
- selection: {(?a,[],?b),(?a,[],?c),(?a,[],?d)} (star)

Capture common correlations:

- mine the most frequent paths (chains and stars) and count
- exact prediction for these paths, otherwise upper bound

Not as easily applicable as histograms, but very accurate

Evaluation

We compare RDF-3X with different competitors:

- MonetDB (column store approach, similar to Abadi et al., VLDB07)
- PostgreSQL (triple store approach, similar to Sesame)
- other approaches performed much worse (see the paper)

Three different data sets:

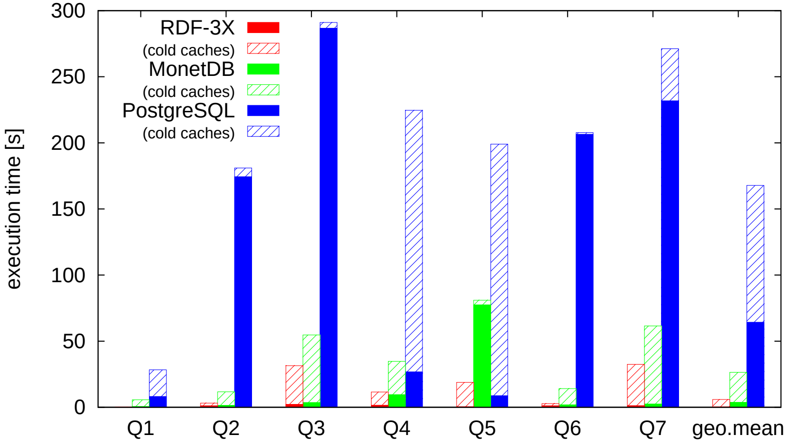
- Barton (same as the VLDB07 paper), library data
- Yago, Wikipedia-based ontology
- LibraryThing (partial crawl), users tag books

Same setup for all competitors:

- all competitors same preprocessing, same dictionary
- equivalent queries (SPARQL for RDF-3X, SQL for others)

Evaluation - Barton Data Set [VLDB07]

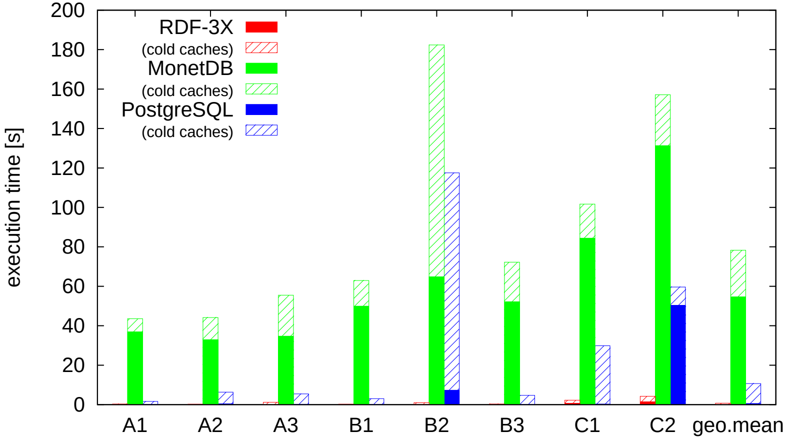
51M triples, 4.1GB original data, 2.8 GB in RDF-3X



sample query (Q5): select ?a ?c where { ?a <origin> <marcorg/DLC>.
?a <records> ?b. ?b <type >?c. filter (?c != <Text>) }

Evaluation - Yago

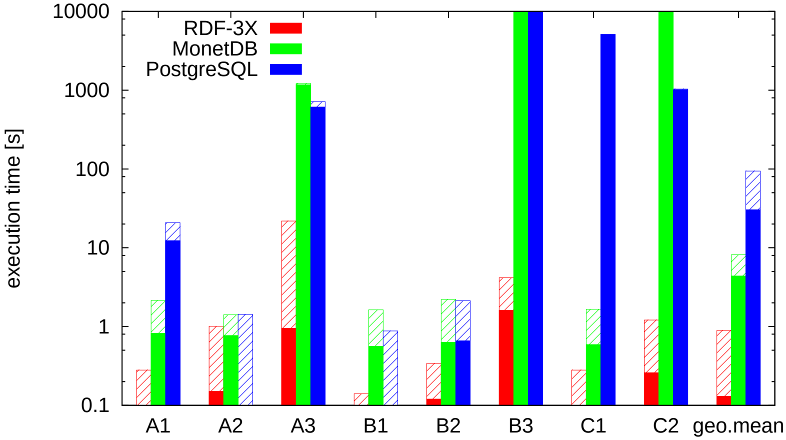
40M triples, 3.1GB original data, 2.7 GB in RDF-3X



sample query (B2): select ?n1 ?n2 where {
?p1 <isCalled> ?n1. ?p1 <bornInLocation> ?city. ?p1 <isMarriedTo> ?p2.
?p2 <isCalled> ?n2. ?p2 <bornInLocation> ?city. }

Evaluation - LibraryThing

36M triples, 1.8GB original data, 1.6 GB in RDF-3X



sample query (B3): select distinct ?u where { ?u [] ?b1. ?u [] ?b2. ?u [] ?b3. [] <english> ?b1. [] <german> ?b2. [] <french> ?b3. }

Conclusion

RDF-3X a fast and flexible RDF/SPARQL engine:

- exhaustive but very **space-efficient triple indexes**
- avoids physical design tuning, generic storage
- few assumptions about data and queries
- **fast runtime system**, exploits indexes for merge joins
- **query optimization** has a huge impact
- **accurate selectivity estimations** essential

RDF-3X is freely available, try it out:
`http://www.mpi-inf.mpg.de/~neumann/rdf3x`