CS 632 : Course Seminar Presentation

On the paper

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

Thomas Neumann and Gerhard Weikum,PVLDB 2008

Presented by: Jiji Angel
Guided by: Prof S Sudarshan

Seminar Outline

- Introduction
  - RDF
  - SPARQL
- Implementation details of RDF-3X
  - Storage & Indexing
  - Query Processing & Optimization
  - Selectivity Estimates
- Experimental Setup & Evaluation Results
- Conclusion

Introduction

- RDF – Resource Description Framework
- Originally was used to model data for semantic web
- Primarily used for knowledge representation and data interchange
- Usages:
  - Ontology representation for semantic web
  - Knowledge base representation; Examples: Freebase, DBpedia, YAGO
  - Import/export data format
  - Non-proprietary data exchange format

RDF Triples

- In RDF every data item is represented using a triple

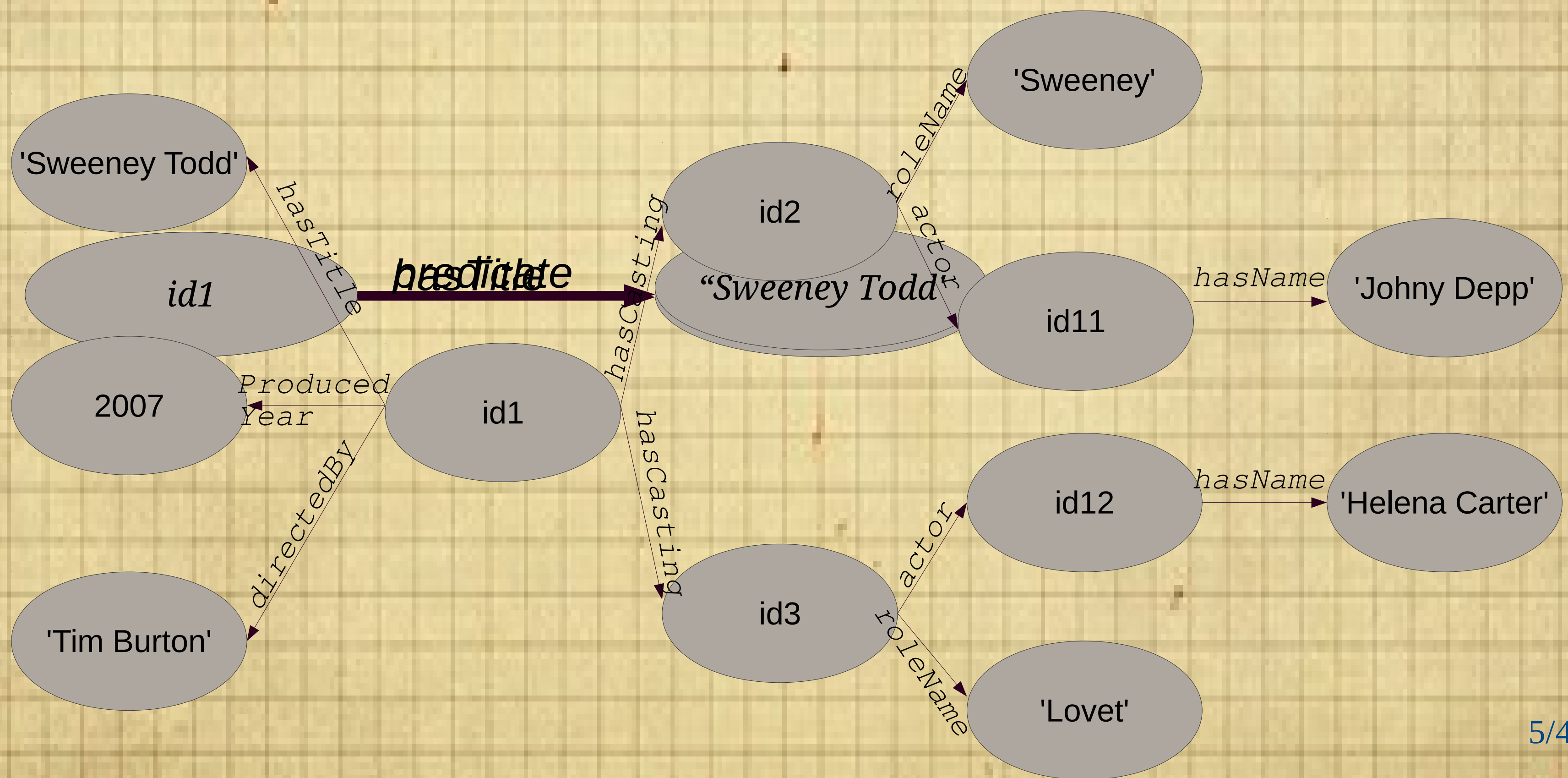
  (*subject*, *predicate*, *object*) aka (*subject*, *property*, *value*)

For example, information about the movie "Sweeney Todd" may be '*triplified*' as:

```
(id1, hasTitle, 'Sweeney Todd'),
(id1, producedYear, '2007'),
(id1, directedBy, 'Tim Burton'),
(id1, hasCasting, id2),
(id1, hasCasting, id3),
(id2, roleName, 'Sweeney'),
(id3, roleName, 'Lovett'),
(id2, actor, id11),
(id3, actor, id12),
(id11, hasName, 'Johny Depp'),
(id12, hasName, 'Helena Carter')
```

RDF – Graph based data model

- Each set of triples is called an RDF graph.
- Each triple is represented as a node-arc-node link; nodes denote subject or object; links denote the predicate

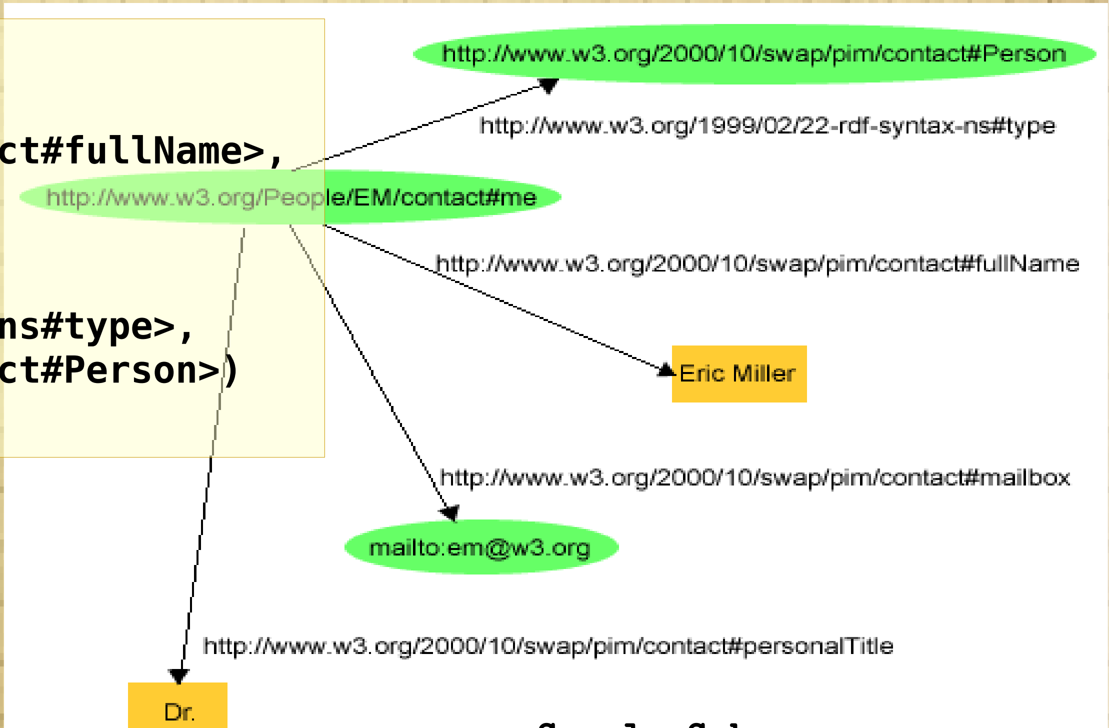


RDF

- Extends the linking structure of the Web by using URIs(Uniform Resource Identifiers) for relationship
- Subjects and predicates are identified by URI values
- Schema language is RDFS (RDF Schema)

```
Triples are:
(<http://www.w3.org/People/EM/contact#me>,
   <http://www.w3.org/2000/10/swap/pim/contact#fullName>,
   "Eric Miller"),

(<http://www.w3.org/People/EM/contact#me>,
   <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>,
   <http://www.w3.org/2000/10/swap/pim/contact#Person>)
```



Sample Schema

Freebase – a knowledge base

- Open knowledge base; Collaboratively edited
- Creative Commons Attribution License
- Repository size: 47+million topics, 2+billion facts, *(as of 15/03/2015)*
- Initially seeded by pulling data from sources such as Wikipedia, MusicBrainz etc.
- Uses RDF graph model for data storage
- Freebase triplestore is named as graphd
- Developed by Metaweb and later aquired by Google

Introduction - SPARQL

- SPARQL – SPARQL Protocol and RDF Query Language
  - Official standard query language for querying RDF repositories
- SPARQL queries are basically pattern matching queries on triples from the RDF data graph

SPARQL Query Examples

- Select all the movie titles *(assume that predicate <hasTitle> implies movie titles)*

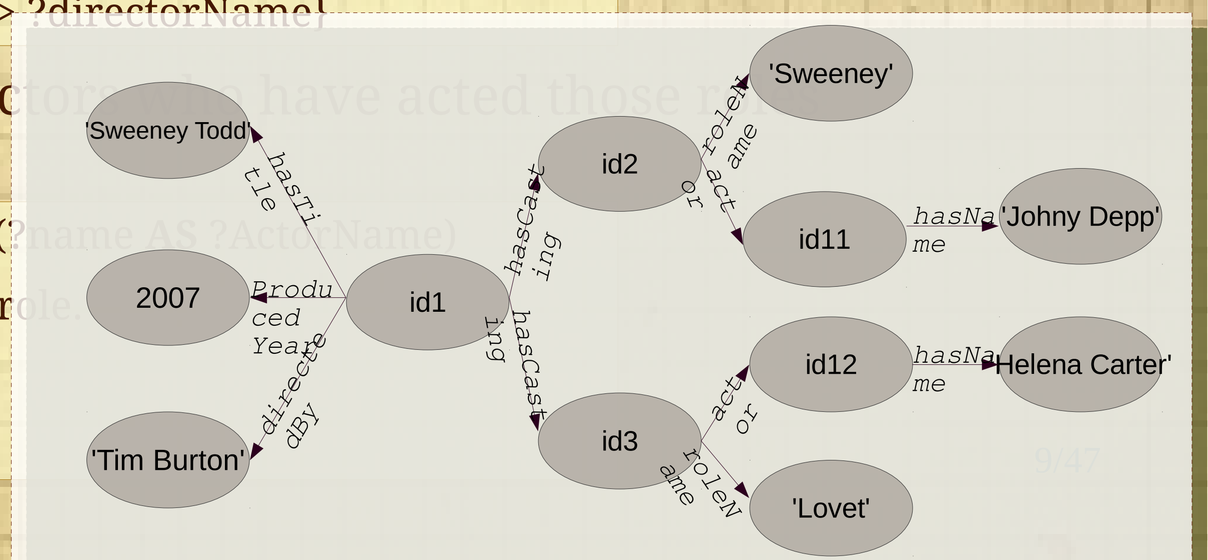
```
SELECT ?title
WHERE{ ?x <hasTitle> ?title }
```

- Select the director of the movie '*Sweeney Todd*'

```
SELECT ?directorName
WHERE{ ?movieId <hasTitle> “Sweeney Todd”.
       ?movieId <directedBy> ?directorName}
```

- Select all the roles and the actors who have acted those roles

```
SELECT (?role AS ?RoleName) (?name AS ?ActorName)
WHERE{ ?roleId <roleName> ?role.
?roleId <actor> ?actorId.
?actorId <hasName> ?name }
```



SPARQL Query Examples

- Select all the movie titles along with the year in which it was produced. *Make sure to get the movie titles even if the production year details are not available.*

SELECT (?title **AS** ?MovieTitle) (?pYear **AS** ?ProductionYear)

WHERE{ ?movieId <hasTitle> ?title

OPTIONAL {?movieId <producedYear> ?pYear}}

- Select all the movies released after ... movies titled 'Sweeney Todd'

SELECT (?title **AS** ?MoviesAfterS...

WHERE{ ?movieId <hasTitle> ?title.

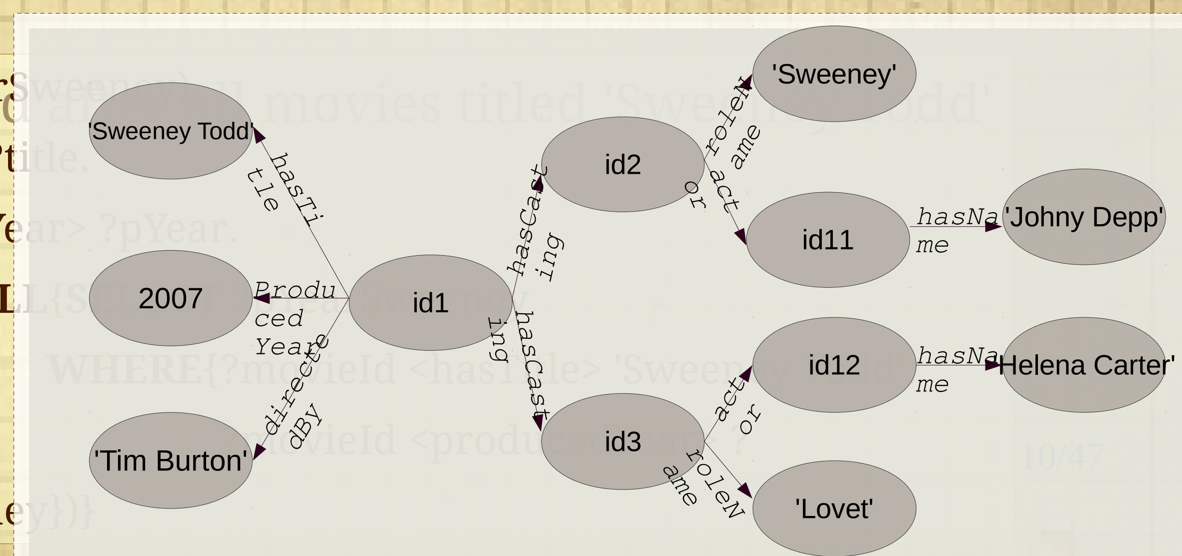
?movieId <producedYear> ?pYear.

FILTER (?pYear <= **AL**L{S...

WHERE{?movieId <hasTitle> 'Sweeney...

?movieId <produ...

pYearSweeney}})}



SPARQL Query Examples

- To retrieve the titles of all the movies with *Johny Depp* by the SPARQL query:

SELECT ?title

WHERE {

?m <hasTitle> ?title. ?m <hasCasting> ?c. ?c <actor> ?a. ?a

<hasName> “Johny Depp” }

- To retrieve movie titles and the list them if the number of actors is more than 10, SPAQRL query can be written as:

SELECT (?title **AS** ?movieTitle) (**COUNT**(?actors) **AS** ?numberOfActors))

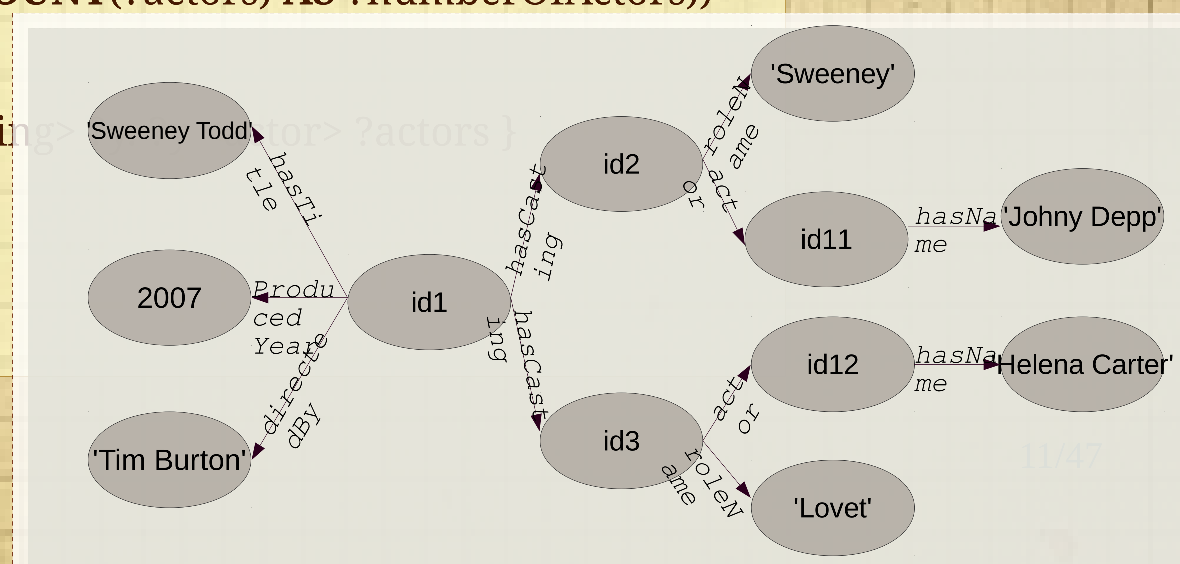
WHERE {

?x <hasTitle> ?title. ?x <hasCasting [illegible] tor> ?actors }

GROUP BY ?x

HAVING (**COUNT**(?actors) > 10)

ORDER BY ?numberOfActors



Introduction – SPARQL Syntax

```
PREFIX foo: <...>
PREFIX rdf: <...>
SELECT [DISCTINCT | REDUCED] ?variable1  ?variable2 ...
WHERE {
   pattern1.  pattern2. ...}
ORDER BY
LIMIT
OFFSET
```

- Each pattern consists of S, P, O and each of these may be either a variable or a literal
- A dot(.) corresponds to join/conjunction; UNION keyword is used for disjunctions
- ORDER BY keyword :orders the result
- DISTINCT keyword : removes duplicates from the result
- REDUCED keyword : may but need not remove duplicates

Result Query Modifiers

Motivation and Problem

- Managing large-scale RDF data involves technical challenges:
  - Physical database design is difficult because of the absence of a global schema
  - RDF data is fine-grained and meant for on-the-fly applications; this calls for appropriate choice of query processing and optimization algorithms
  - Statistics gathering for join-order and execution-plan optimization is not very obvious
  - RDF stores data as graph rather than tree structure used by XML

Contribution & Outline

- RDF-3X (RDF Triple eXpress)
  - RDF-3X engine is an implementation of SPARQL that achieves excellent performance through RISC-style architecture, puristic data structures and operations
  - Key Features:
    - Physical design is workload independent. With exhaustive compressed indexes, it eliminates need for physical-design tuning
    - Query processor rely mostly on merge joins over sorted index lists
    - Query optimizer focuses on join order in generating the execution plan; dynamic programming for plan enumeration
      - Cost model is based on RDF-specific statistics synopsis

Storage and Indexing

RDF Data Storage

- There are three approaches followed by various implementations:
  - Giant Triple Table method
  - Property Table method
  - Cluster Property Table method

Giant Triple Table method

- All triples are stored in a single, giant triple table with generic attributes subject, predicate, object
- RDF-3X follows this approach

Triple Table

Subject	Predicate	Object
id1	hasTitle	"Sweeney Todd"
id1	producedYear	2007
id1	directedBy	"Tim Burton"
id1	hasCasting	id2
id1	hasCasting	id3
id2	roleName	"Sweeney Todd"
id3	roleName	"Lovet"
id2	actor	id11
id3	actor	id12
id11	hasName	"Johny Depp"
id12	hasName	"Helena Carter"

Property Table method

- Separate tables for each predicate

Triple Table

Subject	Predicate	Object
id1	hasTitle	“Sweeney Todd”
id1	producedYear	2007
id1	directedBy	“Tim Burton”
id1	hasCasting	id2
id1	hasCasting	id3
id2	roleName	“Sweeney Todd”
id3	roleName	“Lovet”
id2	actor	id11
id3	actor	id12
id11	hasName	“Johny Depp”
id12	hasName	“Helena Carter”

hasTitle

S	O
id1	“Sweeney Todd”

producedYear

S	O
id1	2007

directedBy

S	O
id1	“Tim Burton”

hasCasting

S	O
id1	id2
id1	id3

roleName

S	O
id2	“Sweeney Todd”
id3	“Lovet”

actor

S	O
id2	id11
id3	id12

hasName

S	O
id11	“Johny Depp”
id12	“Helena Carter”

Cluster-property Table method

- Correlated predicates are kept together in a single table

Triple Table

Subject	Predicate	Object
id1	hasTitle	“Sweeney Todd”
id1	producedYear	2007
id1	directedBy	“Tim Burton”
id1	hasCasting	id2
id1	hasCasting	id3
id2	roleName	“Sweeney Todd”
id3	roleName	“Lovet”
id2	actor	id11
id3	actor	id12
id11	hasName	“Johny Depp”
id12	hasName	“Helena Carter”

Property Table

Subject	hasCasting	roleName	actor	hasName
id1	id2	“Sweeney Todd”	id11	“Johny Depp”
id1	id3	“Lovet”	id12	“Helena Carter”

Left Over Triple Table

Subject	Predicate	Object
id1	hasTitle	“Sweeney Todd”
id1	producedYear	2007
id1	directedBy	“Tim Burton”

Triple Store and Mapping Dictionary

- RDF-3X uses giant triple table approach
  - Drawback – literals can be very large and may contain lot of redundancy
- Solution used by RDF-3X:
  - Use dictionary compression: Mapping Dictionary
    - Compresses the triple store
    - Fast query processing
  - Store all the triples in a clustered B^+-tree
    - Triples are sorted lexicographically
    - Eases SPARQL range queries

Mapping Dictionary

- Used to map literals to a corresponding id
  - This compresses the triple store
  - Simplifies query processing
- Incurs a minor cost of additional dictionary indices

Triple Table

Subject	Predicate	Object
00	01	02
00	03	04
00	05	06
00	07	08
00	07	09
.	.	.
.	.	.
.	.	.

Mapping Dictionary

ID	Value
00	id1
01	hasTitle
02	"Sweeney Todd"
03	producedYear
04	2007
05	directedBy
06	"Tim Burton"
07	hasCasting
08	id2
09	id3

Compressed Indexes

- When literals are prefixes and variables are suffixes in the pattern, the query acts like a range query; suffices to have single index-range-scan
  - For example: (literal1, literal2, ?x)
- To guarantee that queries with all possible patterns are answered in a single index scan, RDF-3X maintain all six possible permutations of subject(S), predicate(P) and object(O), in six seperate indices
  - SPO, SOP, OSP, OPS, PSO, POS
  - Triples in the index are sorted lexicographically
  - Are directly stored in the leaf pages of the clustered B+-tree
  - This ordering causes neighboring triples to be very similar
  - Hence compression of triples is possible: instead of storing full triples RDF-3X stores only the changes between the triples

Sort Orders

- Which sort order to choose?
  - 6 possible orderings, store all of them (SPO, SOP, OSP, OPS, PSO, POS)
  - Will make merge joins very convenient
- Each SPARQL triple pattern can be answered by a single range scan
- Eg: If we need to know all actors of a film, the subject (*“Film object”*) and predicate (<*hasActor*>) remain the same. So, we use the index on sort order “SPO”
- On the other hand, if we need to find all movies in which an actor has acted, the object (*“Actor”*) and predicate (<*hasActor*>) remain the same. So, the index on sort order “OPS” would be more suitable

Compressed Triple Structure

- Comparion of triples is the difference in their id values
  - Triples are sorted lexicographically which allows SPARQL pattern matching into range scans
  - Can be compressed well (delta encoding)
  - Efficient scan, fast lookup if prefix is known
  - Structure of byte-level compressed triple is

Gap	Payload	Delta	Delta	Delta
1 Bit	7 Bits	0-4 Bytes	0-4 Bytes	0-4 Bytes
Header		**value1**	**value2**	**value3**

- Header byte denotes number of bytes used by the three values (5*5*5=125 size combinations)
- Gap bit is used when only value3 changes and delta is less than 128 (that fits in header)

Triple Compression Algorithm

<u>*compress*((*v1*, *v2*, *v3*), (*prev1*, *prev2*, *prev3*))</u>

//Writes (v1, v2, v3) relative to (prev1, prev2, prev3)

if *v1* = *prev1* **&&** *v2* = *prev2*

    if *v3* – *prev3* < 128

        write *v3* – *prev3*

    else *encode*(0, 0, *v3* – *prev3* – 128)

else if *v1* = *prev1*

    encode(0, *v2* – *prev2*, *v3*)

else

    encode(*v1* – *prev1*, *v2*, *v3*)

<u>*encode*($\delta 1$, $\delta 2$, $\delta 3$)</u>

//Writes the compressed tuple corresponding to the deltas

write
    128+*bytes*($\delta 1$)*25+*bytes*($\delta 2$)*5+*bytes*($\delta 3$)

write the non-zero tail bytes of $\delta 1$

write the non-zero tail bytes of $\delta 2$

write the non-zero tail bytes of $\delta 3$

Compressing Triple Example

- Example1: Suppose the first triple is (10,20,1123) and the next triple is (10,20,1173).

  v1 = prev1 and v2 = prev2

  Also, v3 - prev3 < 128

  Hence, the delta entry would be 1173-1123 = 50 in the header record

  Hence, the size of this tuple is only 1 byte; gap bit set to 0

Gap (1 bit)	Payload (7 bits)
0	50

Header Byte

- Example2: Suppose the first triple is (10,20,1000), second triple is (10,20,1500)

  v1 = prev1 and v2 = prev2; but (1500-1000) = 500 !< 128

  Function call: **encode (0,0,372)**

  Header will contain 128 + 0 + 0 + 2 = 130

  δ1 has 0 non-zero bytes, δ2 has 0 non-zero byte, δ3 has 2 non-zero bytes

  Hence, the overall size of the tuple will be 3 bytes

Gap (1 bit)	Payload (7 bits)
1	2

Header Byte

Aggregated Indices

- For many SPARQL queries indexing partial triples rather than full triples would be sufficient

  SELECT ?a ?c

  WHERE { ?a ?b ?c}

- Aggregated Indices:
  - Two-value indices: Each of the possible pairs out of a triple (SP, PS, SO, OS, PO, OP) and the number of occurences of each pair in in the full set of triples
  - One-value indices: Three one valued indices, (S/P/O, count) are stored

RDF-3X Indexing – Three Types Indices

- Six triple indexes: SPO, PSO, SOP, OSP, POS, OPS
- Six two valued aggregated indices and their count: SP, PS, PO, OP, SO, OS
- Three one valued aggregate indices and the respective counts
- Experimentally total size of all indexes is less than original data

Query Processing and Optimization

Translating SPARQL Queries

- *Step1*: Convert the SPARQL query into a query graph representation, interpreted as relational tuple calculus expression
- *Step2*: Conjunctions are parsed and expanded into a set of triple patterns
- *Step3*: Literals are mapped to ids through dictionary lookup
- *Step4*: Multiple query patterns are computed by joining individual triple patterns
- *Step5*: If distinct results are to be obtained, duplicates are removed from the result
- *Step6*: The result contains ids now; dictionary lookup is performed to get back the actual string equivalents

SPARQL Query Graph

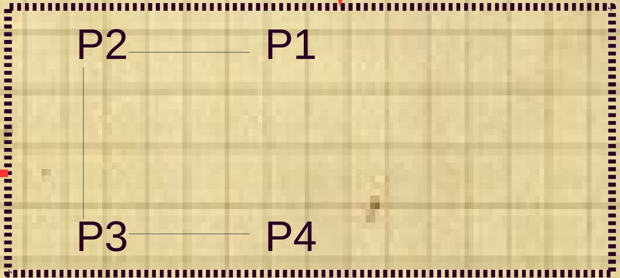
- Each triple pattern corresponds to one node in the query graph
- An edge between two nodes is a common query variable

```
SELECT ?title WHERE {
        ?m <hasTitle> ?title.
        ?m <hasCasting> ?c.
        ?a  <actor> ?c.
        ?a <hasName> "Johny Depp" }
```

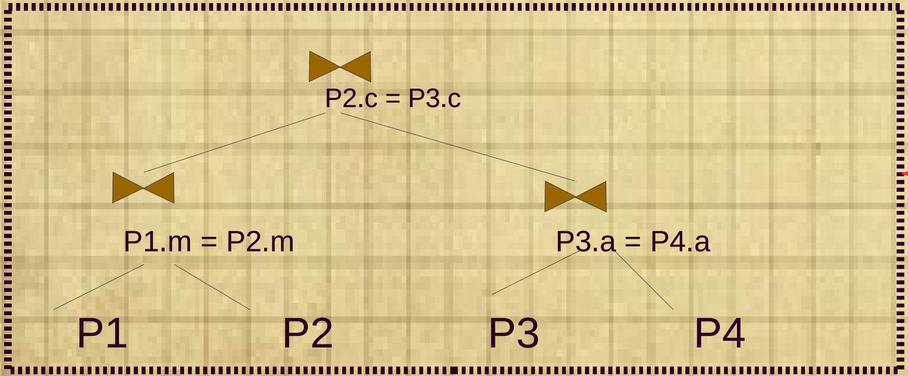
SPARQL query

```
P1 = ?m <hasTitle> ?title
P2 = ?m <hasCasting> ?c
P3 = ?a  <actor> ?c
P4 = ?a <hasName> "Johny Depp"
```

Triple Form



Query Graph



Possible Join Tree

Optimizing Join Ordering

- SPARQL query execution demands join queries which can be really complex:
  - SPARQL queries contain star-shaped subqueries and hence strategies to handle bushy join trees are required
  - Since large number of joins are common in SPARQL queries, fast plan enumeration and cost optimization are required
- RDF-3X uses desicion cost based dynamic programming approach for optimizing join orderings

DP Based Join Optimization

- RDF-3X uses bottom-up dynamic programming approach
  - Takes a connected query graph as input and outputs an optimal bushy join tree
  - Enumerates DP table with the initial set of triples efficiently and correctly
  - Unused(unbound) variables are projected away by using aggregated index
  - The plans that are costlier and equivalent to other plans are pruned
    - Sometimes plans are retained even if they are costlier based on order optimization
  - The larger optimal plan is generated by joining optimal solutions to smaller problems that are adjacent in the query graph

Selectivity Estimates

- Identification of lowest-cost execution plan hugely relies on the estimated cardinalities and selectivities
- A bit different from standard join ordering:
  - One big "relation", no schema
  - Selectivity estimates are hard
  - Standard single attribute synopses are not very useful:
    - Only three attributes and one big relation;
    - But (?a, ?b, ”Mumbai”) and (?a, ?b, ”1974-05-30”) produces vastly different values for ?a and ?b
- Two kinds of statistics are maintained by RDF-3X
  - Selectivity Histograms
  - Frequent Join Paths

Selectivity Histograms

- Query optimizer uses aggregated indexes for calculations based on triple cardinalities
- For estimating join selectivity, histogram buckets with additional information are maintained, as follows

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

Bucket structure

Range Start (10, 2, 30)	Range End (10, 5, 12000)	
Number of triples = 3000		
Number of distinct 2-prefixes = 3		
Number of distinct 1-prefixes = 1		
Join partners on subject		
4000	0	200
Join partners on predicate		
50	400000	200
Join partners on object		
6000	0	9000

Example Bucket implementation

Selectivity Histograms

- Generic but assumes predicates are independent
- Aggregates indexes until they fit into one page
- Merge smallest buckets(equi-depth)
- For each bucket compute statistics
- 6 indexes, pick the best for each triple pattern
- Assumes uniformity and independence, but works quite well

Frequent Paths

- Correlated predicates appear in SPARQL queries in two ways:
  - Stars of triple patterns: a number of triple patterns with different predicates sharing the same subject

    SELECT r_1, r_n

    WHERE{ $(r_1\ p_1\ r_2)$. $(r_1\ p_2\ r_3)$. … $(r_1\ p_n\ r_n)$}
  - Chains of triple patterns: a number of triple patterns where object of the first pattern is subject of the second pattern

    SELECT r_1, r_{n+1}

    WHERE{ $(r_1\ p_1\ r_2)$. $(r_2\ p_2\ r_3)$. … $(r_n\ p_n\ r_{n+1})$}
- Most frequent paths(ie., the paths with the largest cardinalities) are computed, the result cardinalities are materialised along with the path description $p_1, p_2, \ldots p_n$

Frequent Path Mining Algorithm

```
FrequentPath(k)
// Computes the k most frequent paths
C_1 = {P_p | p is a predicate in the database}
sort C_1, keep the k most frequent
C = C_1, i = 1
do
    C_{i+1} = ϕ
    for each p' Є C_i, p predicate in the database
        if top k of C U C_{i+1} U {P_p'p} include all subpaths of p'p
            C_{i+1} = C_{i+1} U {P_p'p}
        if top k of C U C_{i+1} U {P_pp'} include all subpaths of pp'
            C_{i+1} = C_{i+1} U {P_pp'}
    C = C U C_{i+1}, sort C, keep k the most frequent
    C_{i+1} = C_i ∩ C, i = i + 1
while C_i ≠ ϕ
return C
```

Estimates for Composite Queries

- Combining histogram with frequent path statistics
- Long join chain decomposed to subchains of maximal length
  - For exampleconsider a query like:

$?x_1\ a_1\ v_1.\ ?x_1\ p_1\ ?x_2.\ ?x_2\ p_2\ ?x_3.\ ?x_3\ p_3\ ?x_4.$
$?x_4\ a_4\ v_4.\ ?x_4\ p_4\ ?x_5.\ ?x_5\ p_5\ ?x_6.\ ?x_6\ a_6\ v_6$

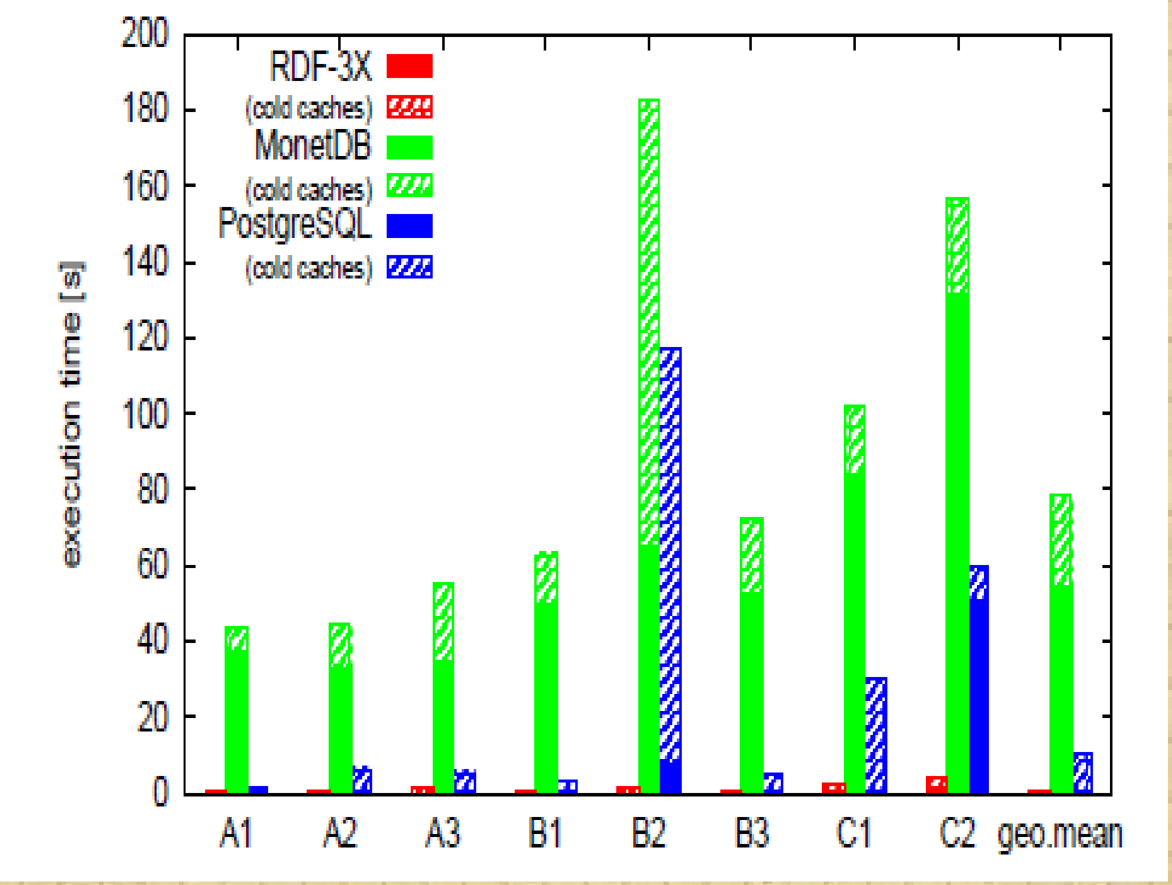
- For subchains p_1-p_2-p_3 and p_4-p_5, selectivity estimation is done using frequency path and for selections histograms are used
- In absence of any other statistics, assume the above two estimators as probabilistically independent - use product formula with per-chain and per-selection statistics as factors

Evaluation

- RDF-3X is compared with:
  - MonetDB (column store approach)
  - PostgreSQL (triple store approach)
- Three different data sets:
  - Yago, Wikipedia-based ontology: 1.8GB
  - LibraryThing : 3.1
  - Barton library data : 4.1GB

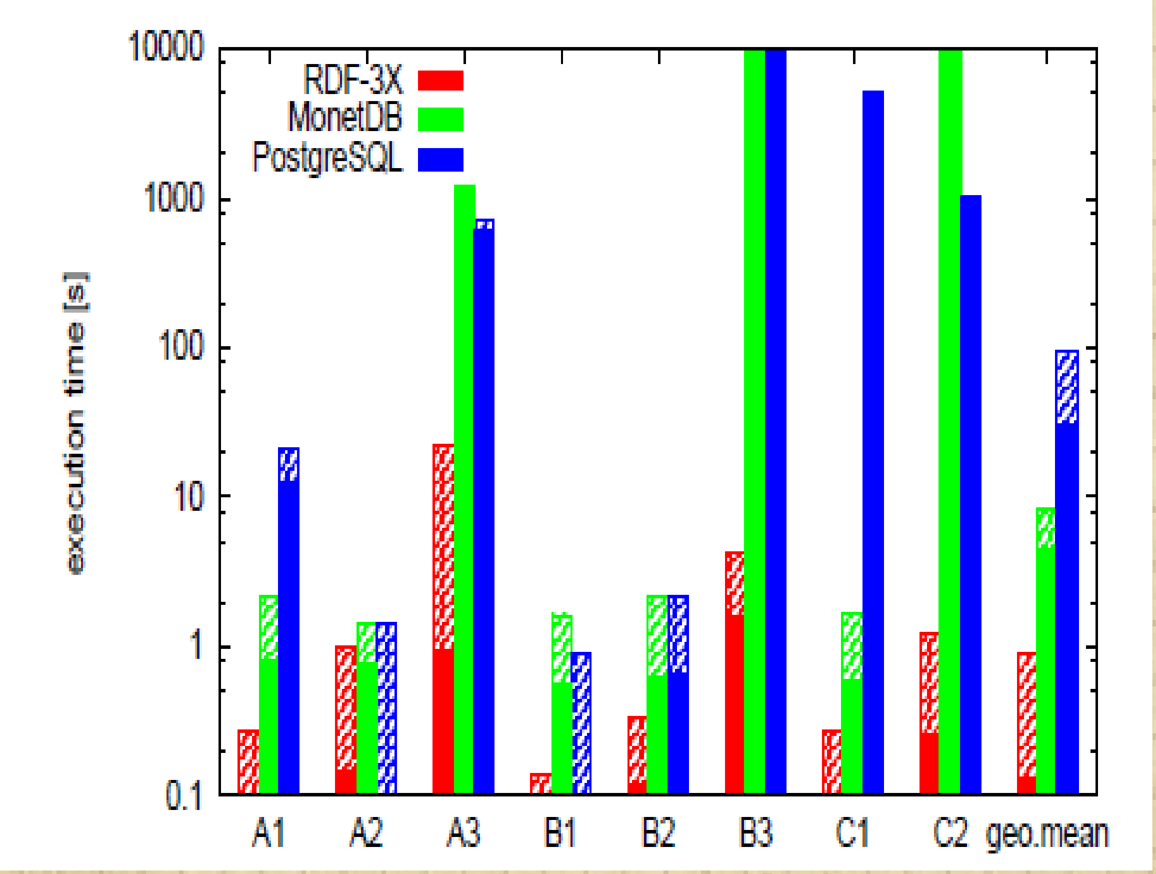
Evaluation

Evaluation - Yogo



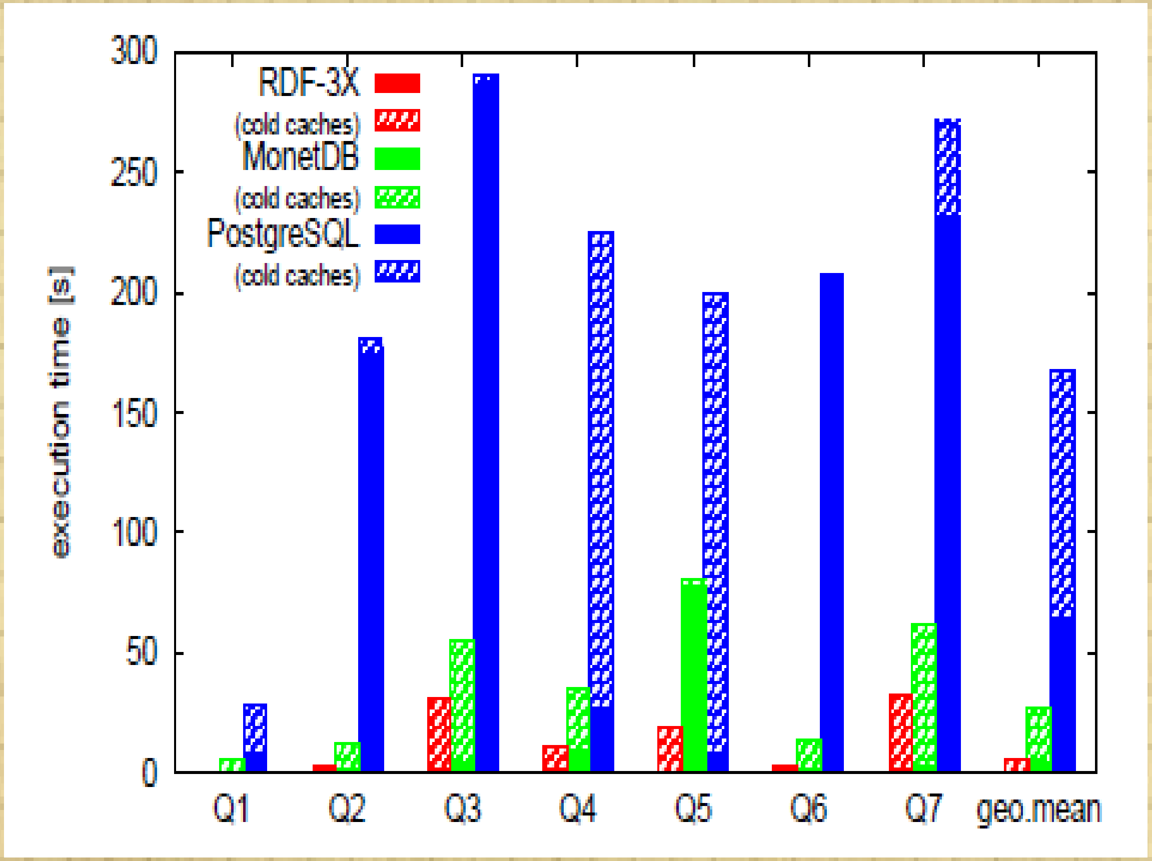
sample query(B2) : select ?n1 ?n2 where { ?p1 <isCalled> ?n1.
?p1 <bornInLocation> ?city. ?p1 <isMarriedTo> ?p2.
?p2 <isCalled> ?n2. ?p2 <bornInLocation> ?city }

Evaluation - LibraryThing



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

Evaluation – Barton Dataset



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

Conclusion

- RDF-3X is a fast and flexible RDF/SPARQL engine
  - Exhaustive but very space-efficient triple indexes
  - Avoids physical design tuning, generic storage
  - Fast runtime system, query optimization has a huge impact

Questions

Thank You