Matrix “Bit” loaded: A Scalable Lightweight Join Query Processor for RDF Data

Hot Topics in Information Retrieval

Amalia Radoiu

Outline

1. Introduction
2. BitMat method overview
3. BitMat construction
4. Join processing algorithm
5. Evaluation
6. Conclusions and further work

Introduction

- RDF (Resource Description Framework) for representing any information

Subject	Predicate	Object
:the_matrix	:releasedIn	"1999"
:the_thirteenth_floor	:releasedIn	"1999"
:the_matrix	:similar_to	:the_matrix_reloaded
:the_thirteenth_floor	:similar_to	:the_matrix
:the_matrix	rdf:type	:movie
:the_thirteenth_floor	rdf:type	:movie

Introduction - SPARQL

SPARQL join query	**Equivalent SQL join query**

SELECT *
 	
 WHERE {
 B - ?m rdf:type :movie
 C - ?n rdf:type :movie
 A - ?m :similar_to ?n
 }
	SELECT * FROM
	 tripletable AS A, tripletable AS B,
	 tripletable AS C
	 WHERE A.subject = B. subject
	 AND A.object = C. subject
	 AND A.predicate = ":similar_to"
	 AND B.predicate = "rdf:type"
	 AND C.object = ":movie"
	 AND B.object = ":movie"
	 AND C. predicate = "rdf:type"

Introduction

- Database systems used for querying and storage of RDF data:

  RDF-3X

  - generic solution for storing and indexing RDF triples
  - query processor that leverages fast merge joins
  - a query optimizer for choosing optimal join orders using a cost model based on statistical synopses for entire join path

Introduction

- MonetDB
  - storage model based on vertical fragmentation
  - a modern CPU-tuned query execution architecture
  - automatic and self-tuning indexes
- Jena-TDB
- Hexastore

Introduction

Join queries classification:

1. Highly selective triple patterns

(?s :residesInUSA)(?s :hasSSN "123-56-6789")

2. Low-selectivity triple patterns, but highly selective join results

(?s :residesIn India)(?s :worksFor BigOrg)

3. Low-selectivity triple patterns and low-selectivity join results

(?s :residesIn USA)(?s :hasSSN ?y)

Introduction

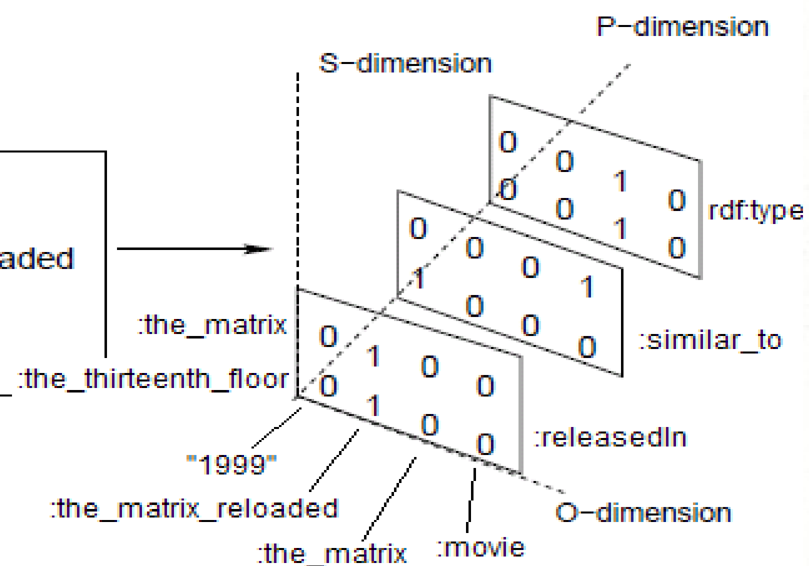
BitMat method overview

- compressed bit-matrix structure for storing huge RDF graphs
- novel, lightweight, SPARQL query processing method
- no intermediate join tables
- works directly on compressed data

BitMat Construction

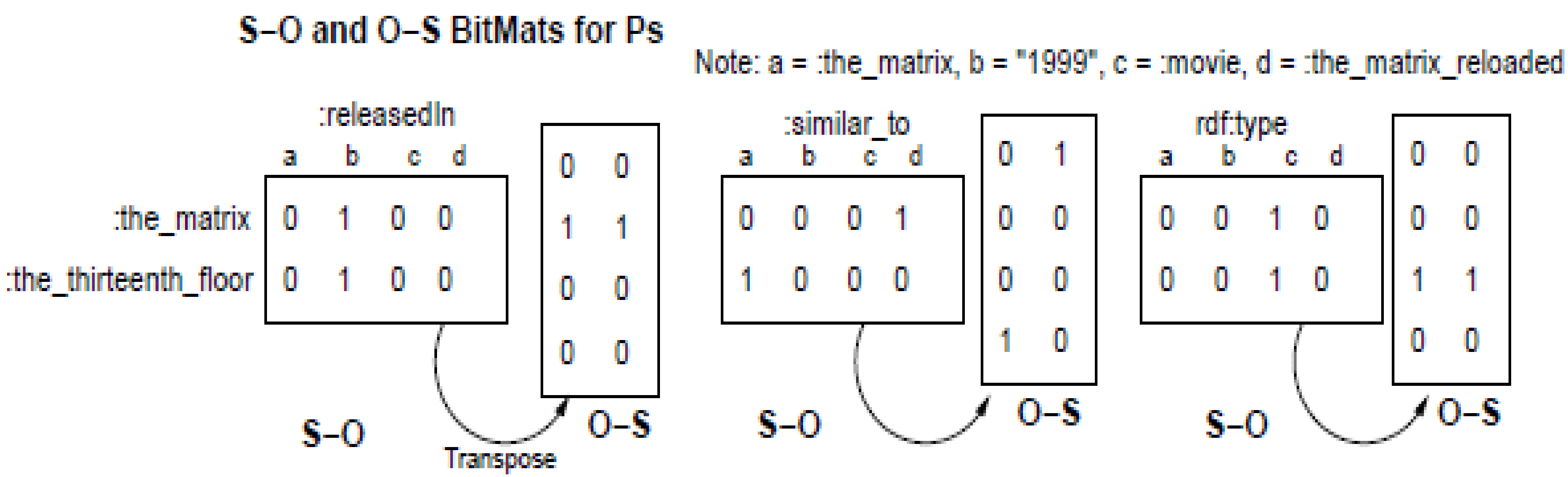
- V_S, V_P and V_O denote sets of distinct subjects, predicates and objects in the RDF data
- 3D bit cube with volume V_S x V_P x V_O

Subject	Predicate	Object
:the_matrix	:releasedIn	"1999"
:the_thirteenth_floor	:releasedIn	"1999"
:the_matrix	:similar_to	:the_matrix_reloaded
:the_thirteenth_floor	:similar_to	:the_matrix
:the_matrix	rdf:type	:movie
:the_thirteenth_floor	rdf:type	:movie



BitMatConstruction

- 3D bit-cube sliced along P-dimension to get 2D matrices
- Inverting an S-O BitMat gives an O-S BitMat
- S-O and O-S BitMats stored for each P value



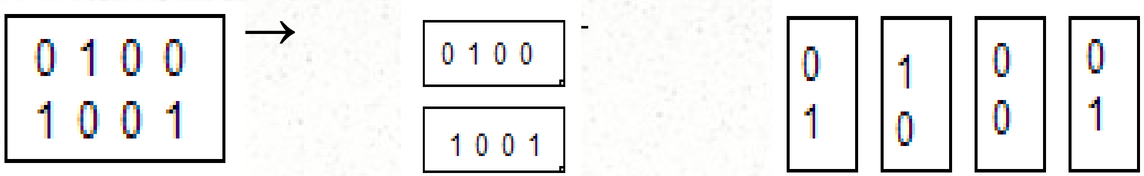
BitMat Construction

- $|V_S|$ x $|V_P|$ x $|V_O|$ possible triples
- RDF data contains much fewer number of triples

→ gap compression scheme

Example: 0011000 → [0] 2 2 3

- Store the number of triples in each compressed BitMat
- Store two bitarrays – row and column bitarray



- Store the compressed S-O, O-S, P-S, P-O BitMats

BitMat Operations

Fold

- *'fold(BitMat, retainDimension) returns bitArray'*
- *RetainDimension* can take the values “rows” or “columns”
- Folds the input BitMat by retaining the *retainDimension*
- Example:

0 0 0 1 OR
1 0 0 0
----
1 0 0 1

BitMat Operations

Unfold

- *'unfold(BitMat, MaskBitArray, RetainDimension)'*
- Unfolds the *MaskBitArray* on the BitMat
- Example: *unfold(BitMat, '011000', 'columns')*

```
0 1 0 1 0 0  AND          [0] 1 1 1 1 2 AND
0 1 1 0 0 0               [0] 1 2 3
---------------------------   -----------------------------
0 1 0 0 0 0               [0] 1 1 4
```

Join Processing Algorithm - Properties

Property 1

?x	p1	?n	p2	?y
x1		n1 n1		y1
x2		~~n4~~ ~~n5~~		y2
x3		n7 n7		y3

Property 2

?x	p	?y
x1		y1
x2		y2
x3		y3
x4		y4

?x	p	?y
x1		y1
x2		y2
~~x3~~		y3
x4		y4

→

?x	p	?y
x1		y1
x2		y2
~~x3~~		~~y3~~
x4		y4

Join Processing Algorithm - Properties

Property 3

?x	p1	?n
x1	p1	n1
x2	p1	n4
x3	p1	n7

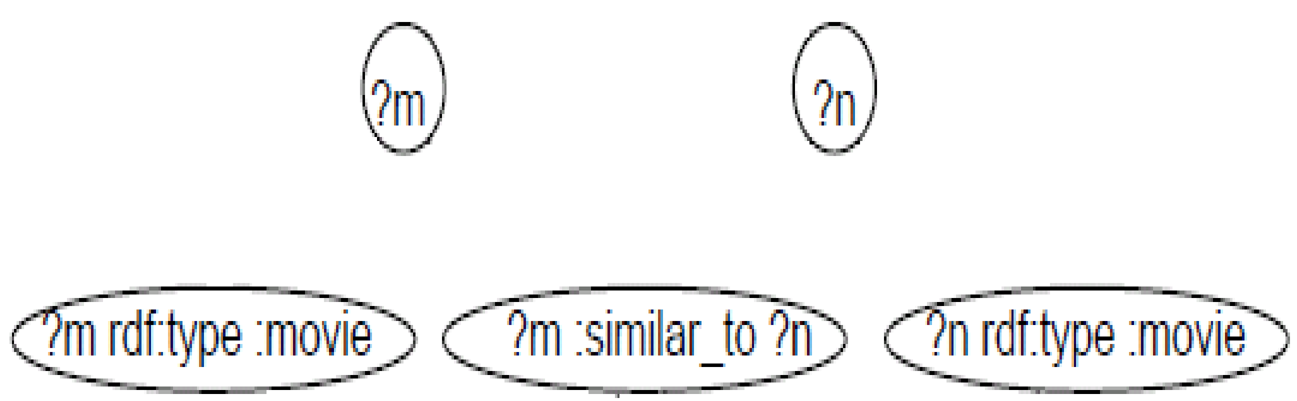
?n	p2	?y
n1	p2	y1
n5	p2	y2
n7	p2	y3

?x	p1	?n	p2	?y
x1		n1 n1		y1
~~x2~~		~~n4~~ ~~n5~~		~~y2~~
x3		n7 n7		y3

Constructing the Constraint Graph G

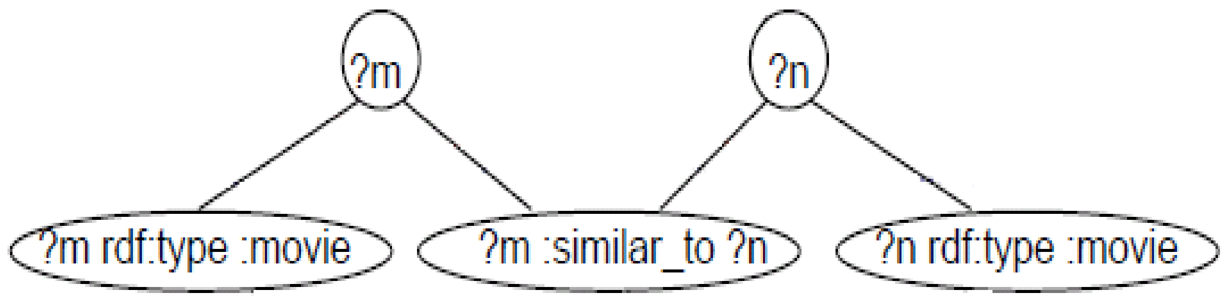
triple pattern = tp-node

join variable = jvar-node



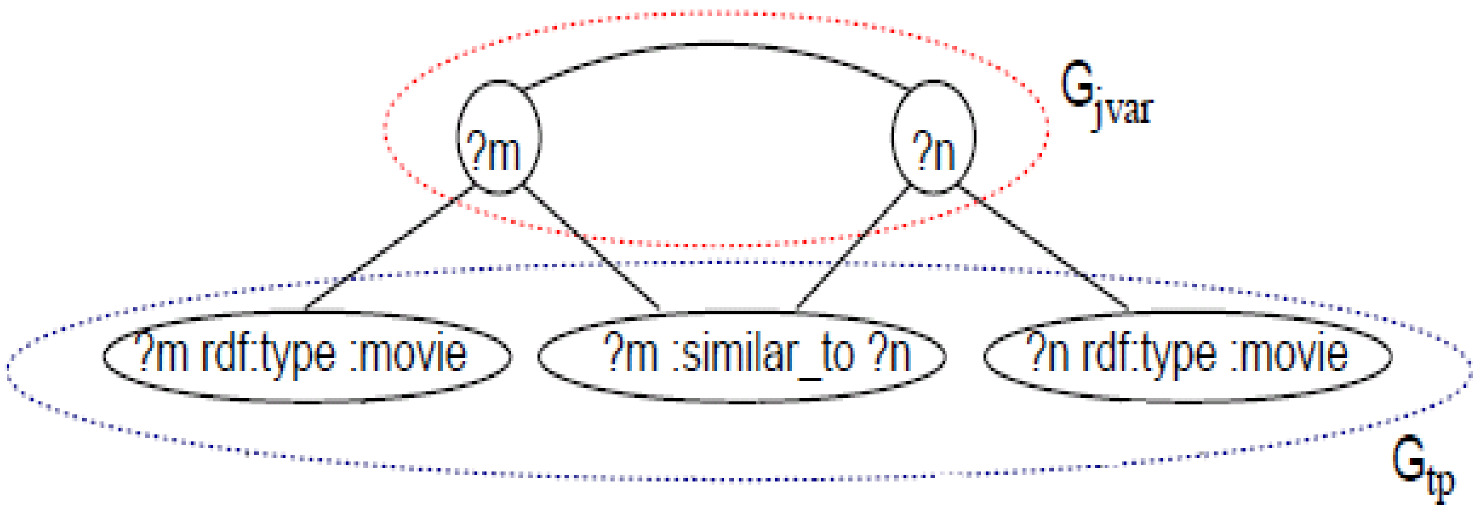
Constructing the Constraint Graph G

- undirected, unlabeled edge between a jvar-node and a tp-node



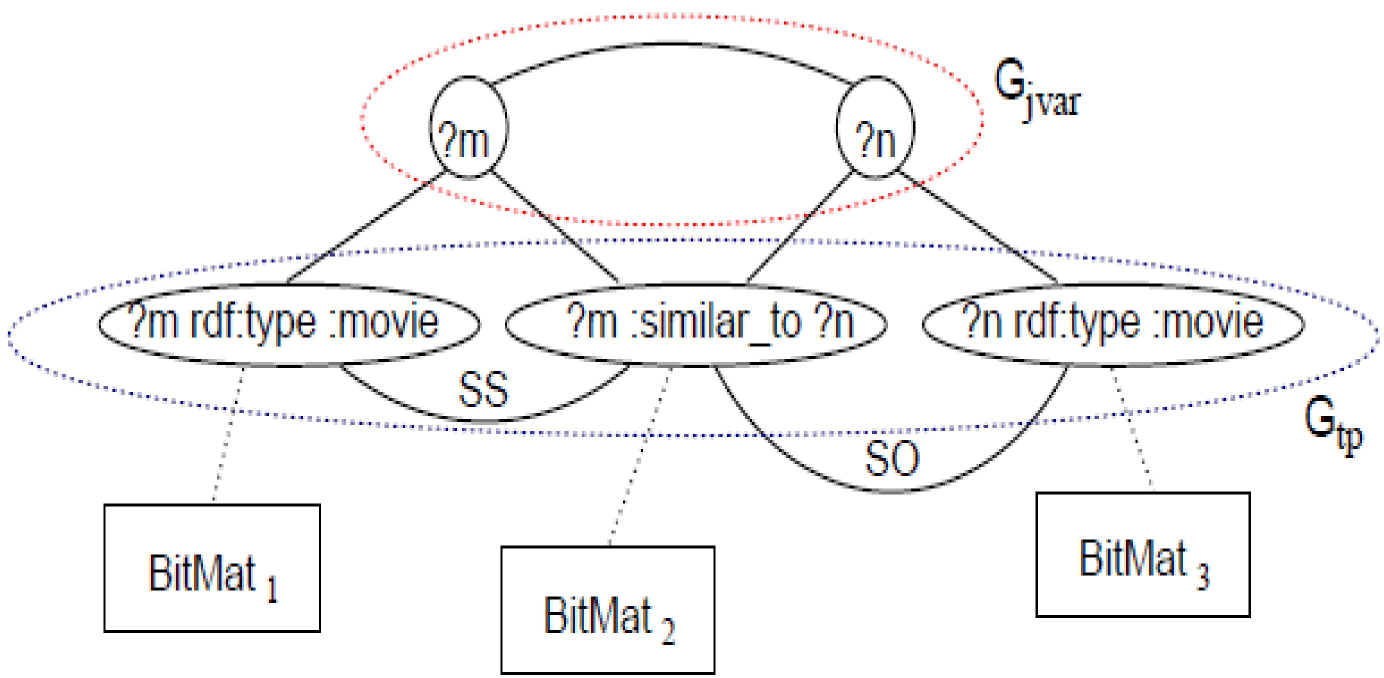
Constructing the Constraint Graph G

- Edge between two jvar-nodes



Constructing the Constraint Graph G

- Edge between two tp-nodes



Join Processing Algorithm – Step 1

Preparation for the pruning algorithm

- Initialize each tp-node by loading the triples which match that triple pattern
- Construct the 4 BitMats S-O, O-S, P-S, P-O

The pruning algorithm

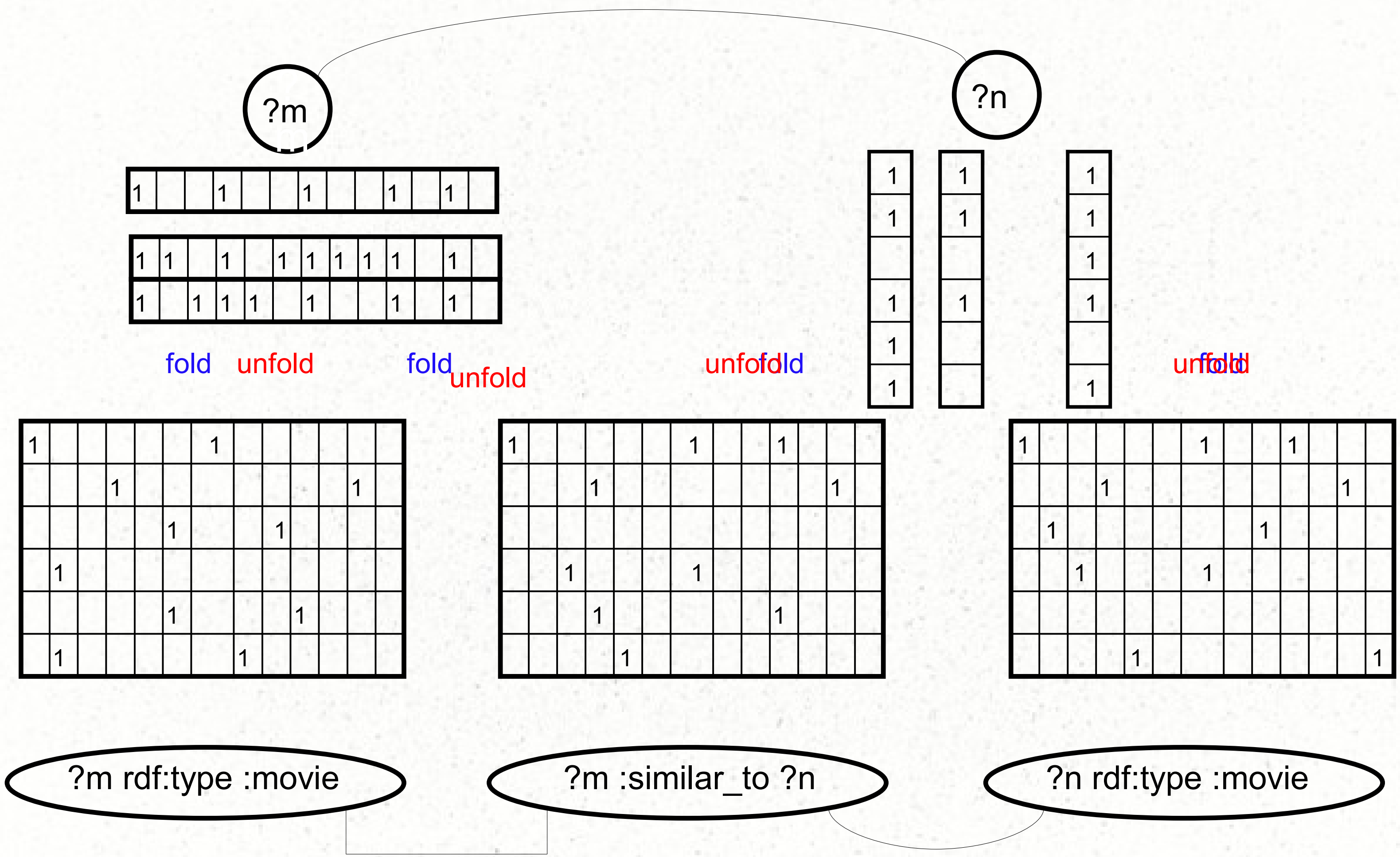
- G_{jvar} contains only jvar-nodes
- Embed a tree on G_{jvar} discarding cyclic edges

Join Processing Algorithm – Step 1

The pruning algorithm (continued)

- Walk the tree from root to the leaves and backwards in a "breadth first search manner"
- At every jvar-node take the intersection of bindings generated by its adjacent tp-nodes
- After the intersection, drop the triples from tp-node BitMats as a result of dropped bindings

Pruning phase



Join Processing Algorithm – Step 1

Algorithm 1 Pruning Step

queue q = topological_sort($V(\mathcal{G}_{jvar})$)
for each J in q **do**
    prune_for_jvar(J)
end for
queue q_rev = q.reverse() - leaves($\mathcal{G}_{jvar}$)
for each K in q_rev **do**
    prune_for_jvar(K)
end for

Algorithm 2 prune_for_jvar(jvar-node J)

$MaskBitArr_J$ = a bit-array containing all 1 bits.
for each tp-node $\mathcal{T}$ adjacent to J **do**
    dim = getDimension(J, $\mathcal{T}$)
    $MaskBitArr_J = MaskBitArr_J$ AND fold($BitMat_{\mathcal{T}}$, dim)
end for
for each tp-node $\mathcal{T}$ adjacent to J **do**
    dim = getDimension(J, $\mathcal{T}$)
    unfold($BitMat_{\mathcal{T}}$, $MaskBitArr_J$, dim)
end for

Join Processing Algorithm – Step 2

- Start with the triple pattern with least number of triples left in its BitMat
- Generate bindings for variables in that triple pattern
- select another triple pattern which shares a join variable with any of the previously selected triple patterns
- Check if it can generate the same bindings for the shared join variable and generate bindings for its other variables
- Continue this and at the end of one round when all triple patterns are processed and all variables have consistent bindings, output the result

Evaluation

Choice of competitive RDF stores

- Considered stores: Hexastore, Jena-TDB, RDF-3X and MonetDB
- Chose RDF-3X and MonetDB because

  → can load a large amount of RDF data

  → gave better performance than others

  → are open-source systems, which can be used by the research community

Evaluation

During the evaluation the following parameters were measured:

1. query execution times (cold and warm cache)
2. initial number of triples
3. the number of results

Cache = A cache in general is a fast temporary store that speeds up access to a (larger) slower store.

Cold Cache = data that isn't in the CPU cache

Warm Cache = data that is in cache

Evaluation

- A UniProt dataset was used in 845,074,885 triples, 147,524,984 subjects, 95 predicates, and 128,321,926 objects

- Another dataset which was generated using LUMB was used, which gave 1,335,081,176 unique triples with 217,206,845 subjects, 18 predicates and 161,413,042 objects

Evaluation

Table 1: Evaluation – UniProt 845 million triples (time in seconds, best times are boldfaced)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Cold cache								
BitMat	**451.365**	**269.526**	173.324	9.396	78.35	1.34	9.33	13.06
MonetDB	548.21	303.2134	**124.3563**	9.63	97.28	11.28	9.91	15.93
RDF-3X	Aborted	525.105	244.58	**1.38**	**4.636**	**0.902**	**0.892**	**1.353**
Warm cache								
BitMat	**440.868**	**263.071**	168.6735	8.305	77.442	0.448	8.36	10.87
MonetDB	495.64	267.532	**113.818**	0.584	96.02	0.822	0.861	0.362
RDF-3X	Aborted	487.1815	226.050	**0.077**	**1.008**	**0.0064**	**0.003**	**0.0299**
#Results	160,198,689	90,981,843	50,192,929	0	179,316	0	0	19
#Initial triples	92,965,468	73,618,481	78,840,372	16,626,073	60,260,006	15,408,126	16,625,901	53,677,336

Evaluation

Table 2: Evaluation – UniProt 845 million triples (time in seconds, best times are boldfaced)

	Q9	Q10	Q11	Q12	Q13	Geom. Mean	Geom. Mean* (without Q1)
Cold cache							
BitMat	11.43	10.49	15.56	26.98	17.37	**25.775**	20.304
MonetDB	21.37	21.39	12.33	**2.468**	12.884	27.891	21.761
RDF-3X	**1.718**	**1.549**	**3.268**	2.804	**1.765**	N/A	**4.268**
Warm cache							
BitMat	9.78	8.69	14.13	25.19	15.77	21.754	16.929
MonetDB	0.611	0.563	0.71	0.744	1.02	**3.845**	2.565
RDF-3X	**0.047**	**0.0469**	**0.547**	**0.295**	**0.0486**	N/A	**0.255**
#Results	2	28	8893	2495	9		
#Initial triples	19,312,584	20,594,986	20,951,969	38,141,013	38,064,279		

Evaluation

Queries where BitMat performed better: Q1, Q2

Queries where results were comparable to other systems: Q3, Q6

Queries where BitMat performed worse than other systems: Q4, Q5, Q7, Q8, Q9, Q10, Q11, Q12, Q13

Conclusions and Further Work

- A novel method of processing RDF join queries was introduced, which:

  → works with compressed data

  → doesn't build intermediate join tables

  → produces the final results in a streaming fashion

Conclusions and Further Work

- RDF-3X and MonetDB gave better results in highly selective queries
- BitMat had a better performance on low-selectivity queries

→ develop a hybrid system having BitMat's query processing algorithm and the conventional query processor

→ the optimal method would be chosen based on heuristics and selectivity of the triple patterns in the query

Questions

- Would it actually be feasible to create a hybrid system? Would the structures be compatible in order to be able to implement such a system?
- There have been made improvements to the RDF-3X system. What would the performance comparison look like with the new version for RDF-3X?
- To what extent is BitMat usable for queries of type (?x ?y ?z), as it is not possible to load a BitMat for all-variable tp-node containing the entire data set in memory?