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

Medha Atre[†], Vineet Chaoji[‡], Mohammed J. Zaki[†], and James A. Hendler[†]
[†]Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy NY, USA
{atrem, zaki, hendler}@cs.rpi.edu
[‡]Yahoo! Labs, Bangalore, India
chaojv@yahoo-inc.com

ABSTRACT

The Semantic Web community, until now, has used traditional database systems for the storage and querying of RDF data. The SPARQL query language also closely follows SQL syntax. As a natural consequence, most of the SPARQL query processing techniques are based on database query processing and optimization techniques. For SPARQL join query optimization, previous works like RDF-3X and Hexastore have proposed to use 6-way indexes on the RDF data. Although these indexes speed up merge-joins by orders of magnitude, for complex join queries generating large intermediate join results, the scalability of the query processor still remains a challenge.

In this paper, we introduce (i) BitMat – a compressed bit-matrix structure for storing huge RDF graphs, and (ii) a novel, light-weight SPARQL join query processing method that employs an initial pruning technique, followed by a variable-binding-matching algorithm on BitMats to produce the final results. Our query processing method does not build intermediate join tables and works directly on the compressed data. We have demonstrated our method against RDF graphs of upto 1.33 billion triples – the largest among results published until now (single-node, non-parallel systems), and have compared our method with the state-of-the-art RDF stores – RDF-3X and MonetDB. Our results show that the competing methods are most effective with highly selective queries. On the other hand, BitMat can deliver 2-3 orders of magnitude better performance on complex, low-selectivity queries over massive data.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query Processing

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

Resource Description Framework (RDF)[1], a W3C standard for representing *any* information, and SPARQL[2], a query language for RDF, are gaining importance as semantic data is increasingly becoming available in the RDF format. RDF data consists of *triples* where each triple presents a relationship between its *subject* (S) and *object* (O), the name of the relationship is given by the *predicate* (P), and the triple is represented as (S P O). Such an RDF data can be represented as a labeled directed graph and it can be serialized and stored in a relational database simply as a 3-column table where each tuple in that table represents a triple in the original RDF graph.

RDF is extensively being used for representing data from the fields of bioinformatics, life sciences, social networks, and Wikipedia as well. Since disk-space is getting cheaper, storing this huge RDF data does not pose as big a problem as executing queries on them. Querying these huge graphs needs scanning the stored data and indexes created over it, reading that data inside memory, executing query algorithms on it, and building the final results of the query. Hence, a desired query processing algorithm is one which (i) keeps the underlying size of the data small (using compression techniques), (ii) can work on the compressed data without uncompressing it, and (iii) doesn't build large intermediate results. A lot of work has already gone in using compression techniques in storing the data in a column-store database as well as trying to work on the compressed data without uncompressing it by *lazy materialization* [2, 4]. In this paper, we go one step further and propose a *compressed bitcube* of RDF data and a novel SPARQL join query processing approach which always works on the compressed data by producing the final results in a *streaming fashion* without building intermediate join tables.

A SPARQL join query, which can also be viewed as a *Basic Graph Pattern Matching* (BGP) query, or a *conjunctive triple pattern query*, resembles closely to an SQL join query (in fact any SPARQL join query can be systematically translated into an SQL join query [9]). A typical SPARQL join query looks like the one shown in Figure 1. This query shows a join between three triple patterns.

Such join queries can be broadly classified into three categories. The first type is – queries having highly *selective triple patterns*[3]. E.g., consider the query *(?s :residesIn USA)(?s :hasSSN "123-45-6789")*. Since SSN is a unique attribute of a person, the second triple pattern has only one triple associated with it thereby being highly selective. The second type is – queries having triple patterns with low-selectivity but which generate few results, i.e., highly selective join results. E.g., consider a multi-national orga-

[1]http://www.w3.org/TR/rdf-syntax-grammar/
[2]http://www.w3.org/TR/rdf-sparql-query/

[3]Selectivity of a triple pattern is low if there are more number of triples associated with it and vice versa.

WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

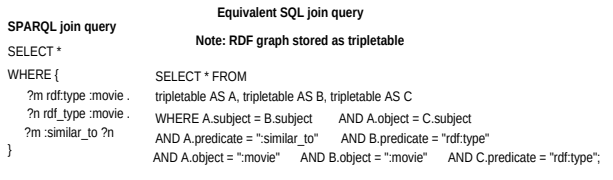

Figure 1: An example of SPARQL join query

nization *BigOrg*, having employees on the order of few millions all over the world. Now consider a densely populated country like India having population close to 1.2 billion and consider a query like *(?s :residesIn India)(?s :worksFor BigOrg)*. Although the number of triples associated with the two triple patterns is quite high, their join will produce much fewer number of results as there are only a few employees of *BigOrg* in India. The third type of queries are the ones having low-selectivity triple patterns and low-selectivity join results, i.e., the ones generating a lot of results. For instance, a modification of the first query given above, to find SSNs of all the people *(?s :residesIn USA)(?s :hasSSN ?y)*.

Most of the systems which generate various indexes on the data do well on the first type of queries – highly selective triple patterns. Especially having 6-way indexes helps in picking the right set of triples at the beginning, avoiding scanning a large amount of data. For the second type of queries – low-selectivity triple patterns, but highly selective join results – systems using *join selectivity estimation* or pre-computed join tables/indexes get benefited to a certain extent. Although as shown in our experiments, join selectivity estimation does not always help in improving the query performance in case of complex joins involving low-selectivity intermediate join results. For the third type of queries – low-selectivity triple patterns generating a large number of results – even the state-of-the-art systems run into problems (as shown by our evaluation).

Although disk-space is growing at a much faster speed, the available main-memory still remains very small compared to it. This creates the main bottleneck while executing queries of the second and third type mentioned above. Hence our goal is to build a scalable query algorithm which operates on the compressed data. Second our goal is to not generate intermediate join tables, thereby keeping the memory footprint of the system small (hence light-weight), and cope well with the second and third type of queries mentioned above. Our key contributions in this work are:

1. A compressed data structure for RDF data – called BitMat – to increase the size of the data that can fit in memory.

2. A novel algorithm that performs efficient pruning of the triples during the first phase of SPARQL join query execution and in the second phase, performs variable binding matching across the triple patterns to obtain the final results (both phases use compressed BitMats without any join table construction).

3. Procedures and algorithms implemented to work on the compressed data directly.

4. Experiments on very large RDF graphs (∼845 million and ∼1.33 billion) using a set of queries published on the web by the owners of the datasets, showing a comparison with the state-of-the-art RDF storage systems – RDF-3X and MonetDB. Our results indicate that competing methods are much better on high-selectivity queries, whereas our method outperforms them by 2-3 orders of magnitude on complex queries with low-selectivity intermediate join results.

Work presented in this paper is a considerable extension of our preliminary work outlined previously in [5].

2. RELATED WORK

RDF data can be serialized and stored in a database and a SPARQL join query can be executed as an SQL join, hence recently a lot of database join query optimization techniques have been applied to improve the performance of SPARQL join queries. Notably, in the past couple of years, C-Store [3], RDF-3X [16], MonetDB [21], and Hexastore [24] systems have proposed ways of optimizing SPARQL join queries.

Out of these systems, C-Store and MonetDB exploit the fact that typically RDF data has much less number of properties (predicates), thereby *vertically partitioning* the data for each unique predicate and sorting each partition (predicate table) on subject, object order (creating a *subject-object* index on each property table). Hexastore and RDF-3X make use of the fact that an RDF triple is a fixed 3-dimensional entity and hence they create all 6-way indexes (SPO, SOP, PSO, POS, OPS, OSP). Although Hexastore does share common indexes within these 6 indexes, e.g., SPO and PSO share the "O" index, without any compression, it suffers from 5-fold increase in the space required to store these indexes. RDF-3X goes one step further and compresses these indexes as described in their paper [15]. RDF-3X also implements several other join optimization techniques like RDF specific *Sideways-Information-Passing*, selectivity estimation, merge-joins, and using bloom-filters for hash joins.

Along with these systems, there are other systems being developed for RDF data storage and querying, such as, Jena-TDB [1] and Virtuoso [10]. Jena-TDB faces scalability issues while executing queries on very large datasets. Along with these, BRAHMS [11] and GRIN [22] focus more on path-like queries on RDF data, typically which cannot be expressed using existing SPARQL syntax.

Most systems built to store and query RDF data typically use a left-deep join tree which requires materialization of the intermediate join results in case of a complex join query involving several join variables. Merge-joins cannot always be used while performing later joins, especially when the join column of an intermediate result is not sorted. In contrast to these, in our system instead of using sophisticated join optimization techniques, we have followed a simple rule of keeping the data compressed without materializing the intermediate join results. This helps to keep a large amount of required data in memory. We execute the join by following a novel algorithm, which propagates the constraints on the join-variable bindings among different join variables in the query. Our technique is reminiscent of the concept of *semi-joins* [7, 6] as discussed further in Section 4.1. We consider our query processing engine *light-weight* – light-weight on runtime memory consumption as well as optimization techniques. We have shown results by analyzing where our system outperforms the state-of-the-art systems like RDF-3X and MonetDB.

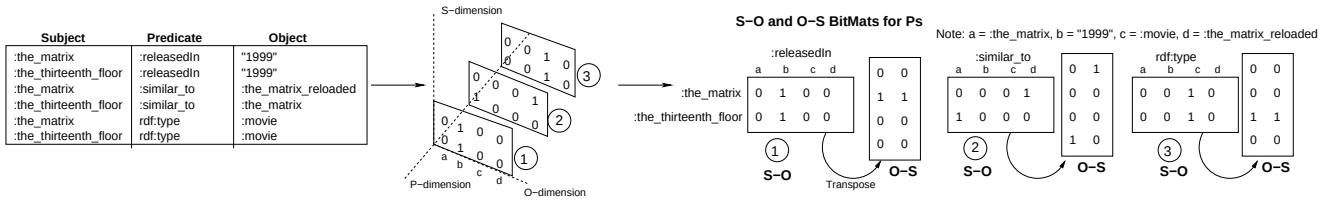

Figure 2: Example of S-O, O-S BitMat construction procedure for each P

3. BITMAT CONSTRUCTION

Figure 2 shows sample RDF data and gives a pictorial representation of constructing a BitMat. Let V_s, V_p and V_o denote the sets of distinct subjects, predicates, and objects, respectively, in the RDF data. This RDF data can be represented by a 3D bit-cube, where each dimension of the bitcube represents subjects (S), predicates (P), and objects (O). The volume of this bitcube is $V_s \times V_p \times V_o$. Each cell in the bitcube represents a unique RDF triple that can be formed by the combination of S, P, O positions which are the coordinates of that bit. A bit set to 1 denotes presence of that triple in the given RDF data.

This 3D bit-cube is sliced along a dimension to get 2D matrices. Figure 2 shows slicing along the P-dimension, which gives S-O bit-matrices (BitMats). Inverting an S-O BitMat gives an O-S BitMat. We store these S-O and O-S BitMats for each P value. In all we get $|V_p|$ such S-O and O-S matrices. Additionally, we slice the bitcube along S and O dimensions which gives P-O and P-S BitMats respectively. Note that we do not store inverted O-P and S-P BitMats, since based on our experience, usage of those BitMats is rare; and even if needed their construction from the corresponding P-O or P-S BitMat is easier due to the relatively few number of predicates in typical RDF data. In all we have $|V_s|$ P-O BitMats and $|V_o|$ P-S BitMats. To summarize, for each P value we have a S-O and an O-S BitMat, for each S value – a P-O BitMat, and for each O value – a P-S BitMat (in all $2|V_p| + |V_s| + |V_o|$ such BitMats).

There are total $|V_s| \times |V_p| \times |V_o|$ possible triples with the given V_s, V_p, and V_o sets. But it is observed that typically RDF data contains much fewer number of triples, hence the S-O, O-S, P-S, P-O BitMats are very sparse. We make use of this fact by applying gap compression on each bit-row of these four types of BitMats. In gap compression scheme, a bit-row of "0011000" will be represented as "[0] 2 2 3". That is, starting with the first bit value, we record alternating run lengths of 0s and 1s.

We also store the number of triples in each compressed BitMat (this statistics is useful while executing our query algorithm as described later). Along with this, we store two bitarrays – row and column bitarray – which give a condensed representation of all the non-empty row and column values in the given BitMat. For example, in Figure 2, for the S-O BitMat of ":similar_to" predicate (marked by BitMat "2" in Figure 2), we store row bitarray "1 1" and a column bitarray "1 0 0 1", and for the O-S BitMat we store row bitarray "1 0 0 1", and column bitarray "1 1" respectively. These bitarrays are useful while performing "star join" queries (as elaborated in the Evaluation section). We store the compressed S-O, O-S, P-S, P-O BitMats in one file on the disk and maintain a meta-file which gives the offset of each BitMat inside the BitMat file. Due to this, addition or deletion of the triples might require moving a large amount of data, but if bulk updates are expected on the RDF data, all the BitMats can be rebuilt at once since the BitMat construction time even for very large data is very small (as shown at the end of the Evaluation section).

The above construction reveals that each unique S, P, and O is mapped to a unique position along each dimension and this position can be represented as an integer ID. We decide this mapping with the following procedure: Let V_{so} represent the $V_s \cap V_o$ set. Each element in V_{so}, along with the elements in V_s, V_p and V_o, is assigned an integer ID as follows:

- *Common subjects and objects*: Set V_{so} is mapped to a sequence of integers: 1 to $|V_{so}|$.
- *Subjects*: Set $V_s - V_{so}$ is mapped to a sequence of integers: $|V_{so}| + 1$ to $|V_s|$.
- *Predicates*: Set V_p is mapped to a sequence of integers: 1 to $|V_p|$.
- *Objects*: Set $V_o - V_{so}$ is mapped to a sequence of integers: $|V_{so}| + 1$ to $|V_o|$.

The common subject-object identifier assignment facilitates the bitwise operations in join queries wherein an S position in one triple pattern is joined over an O position in another triple pattern (e.g. *?n* in the query in Figure 1). For the present considerations, we do not handle joins across S-P and P-O dimensions. Such queries are rare in the context of *assertional* RDF data. None of the benchmark queries published for the large RDF datasets have queries having joins over S-P or P-O dimensions. Hence overlapping S, P, O IDs except for the common S and Os do not pose a problem while processing a query.

With respect to the construction described above, the RDFCube [14] system is conceptually closest to BitMat. RDFCube also builds a 3D cube of S, P, and O dimensions. However, RDFCube's design approximates the mapping of a triple to a cell by treating each cell as a hash bucket containing multiple triples. They primarily used this as a distributed structure in a peer-to-peer setup (RDFPeers [8]) to reduce the network traffic for processing join queries in a conventional manner. In contrast, BitMat's compressed structure maintains unique mapping of a triple to a single bit, and also employs a different query processing algorithm. Further, RDFCube has demonstrated their results on a bitcube of only up to 100,000 triples, whereas we have used more than 1.33 billion triples in this paper.

3.1 BitMat Operations

In this section we define two basic operations *fold* and *unfold* which are used by our join query algorithm. Fold

and unfold operate on the S-O, O-S, P-O, P-S compressed BitMats constructed while storing the original RDF data.

(1) **Fold:** *fold* operation represented as *'fold(BitMat, RetainDimension) returns bitArray'* folds the input BitMat by retaining the *RetainDimension*. For example, if in an S-O BitMat of a given predicate P, *RetainDimension* is set to *'columns'*, then BitMat is folded along the subject "rows" resulting into a single bitarray, i.e., all the subject bit-rows are ORed together to give an "object" bit-array. Intuitively, a bit set to 1 in this array indicates the presence of *at least* one triple with the "object" corresponding to that position in the given S-O BitMat. Without loss of generalization, this procedure can be applied to any of the S-O, O-S, P-O, P-S BitMats with "rows" or "columns" as *RetainDimension*.

(2) **Unfold:** Specified as *'unfold(BitMat, MaskBitArray, RetainDimension)'*, unfolds the *MaskBitArray* on the *BitMat*. Intuitively, in the *unfold* operation, for every bit set to *0* in the *MaskBitArray* all the bits corresponding to that position of the *RetainDimension* in the *BitMat* are cleared. For example, *unfold(BitMat, '011000', 'columns')* would result in a bitwise AND of "[0] 1 2 3" (gap compressed representation of '011000') and each row of the BitMat.

Note that *fold* and *unfold* operations are implemented to operate directly on a compressed BitMat. For example, a bitwise AND of compressed arrays – *arr1* as [0] 2 3 4 and *arr2* as [1] 3 4 2 – can be performed by sequentially looking at their "gap values". E.g., AND the first gap of *arr1* – 2 0s and *arr2* – 3 1s, which gives the first gap of 2 0s in the result. Since the two gaps were of uneven length, there is a leftover 1 from the first gap of *arr2*. Now AND the second gap of *arr1* – 3 1s, and leftover first gap of 1 1s from *arr2*, which gives second gap in the result – 1 1s, so on and so forth. Bitwise OR on the compressed bitarrays can be done with AND using simple Boolean logic (a OR b) = NOT(NOT(a) AND NOT(b)). A bitwise NOT operation on a compressed bitarray is simply – NOT([0] 2 3 4) = [1] 2 3 4.

4. JOIN PROCESSING ALGORITHM

Before describing our join processing algorithm, we would like to note some facts about the join process.

PROPERTY 1. *Each triple pattern in a given join query has a set of RDF triples associated with it which satisfy that triple pattern. These triples generate bindings for the variables in that triple pattern. If the triples associated with another triple pattern containing the same variable cannot generate a particular binding, then that binding should be dropped. In that case, all the triples having that binding value should be dropped from the triple patterns which contain that variable.*

PROPERTY 2. *If two join variables in a given query appear in the same triple pattern, then any change in the bindings of one join variable can change the bindings of the other join variable as well.*

PROPERTY 3. *A join between two or more triple patterns over a join variable indicates an intersection between bindings of that join variable generated by the triples associated with the respective triple patterns.*

To elaborate the use of these properties, consider the query given in Figure 1. *?m* and *?n* appear in the same triple pattern *(?m :similar_to ?n)*. A position marked with "?" in the triple pattern is variable. If we perform a join of *(?m :similar_to ?n) (?m rdf:type :movie)* first, we get two bindings for the variable *?m* viz. *:the_matrix* and *:the_thirteenth-_floor* and two for *?n :the_matrix_reloaded* and *:the_matrix* corresponding to *?m*'s bindings. When we do the join between *(?n rdf:type :movie)(?m :similar_to ?n)*, we consider bindings generated for *?m* and *?n* after the first join. After the join on *?n*, binding *:the_matrix_reloaded* for *?n* gets dropped, hence the triple *(:the_matrix :similar_to :the_matrix_reloaded)* gets dropped from the triples associated with *(?m :similar_to ?n)* which in turn drops the the binding *:the_matrix* for *?m*.

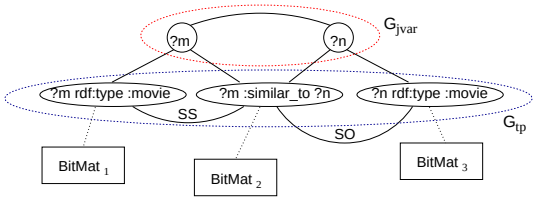

Figure 3: Graph $\mathcal{G}$ for the query in Figure 1

Properties 1, 2 and 3 together establish the basis of our pruning algorithm. We propagate the constraints on the bindings of each join variable in a given triple pattern to all other triple patterns and do aggressive pruning of the RDF triples associated with them.

4.1 Step 1 – Pruning the RDF Triples

First we construct a *constraint graph*[4] $\mathcal{G}$ out of a given join query. The constraint graph is built as follows:

1. Each triple pattern in the join query is denoted by a *tp-node* in $\mathcal{G}$. Hence forth we use the terms "tp-node" and "triple pattern" interchangeably. A *jvar-node* in $\mathcal{G}$ corresponds to a join variable in the query. Hence forth we use terms "jvar-node" and "join variable" interchangeably.

2. An undirected, unlabeled edge between a jvar-node and a tp-node exists in $\mathcal{G}$ if that join variable appears in the triple pattern represented by the tp-node. This edge represents the dependency between triples associated with the tp-node and the join variable bindings (ref. Property 1).

3. An edge exists between two jvar-nodes if the two join variables appear in the same triple pattern. This undirected, unlabeled edge represents the dependency between their bindings (ref. Property 2).

4. An edge between two tp-nodes exists if they share a join variable between them. This is an undirected, *labeled* edge with potentially multiple labels. Multiple labels can appear if the two triple patterns share more than one join variables. The labels denote the type of join between the two triple patterns – *SS* denotes subject-subject join, *SO* denotes subject-object join etc.

For a query having no *Cartesian joins*[5], constraint graph

[4]This graph is reminiscent of similar terminology used in the constraint satisfaction literature.

[5]A Cartesian join is where there is no shared variable in a set of triple patterns, and hence the result of the query is a full Cartesian product of all triples associated with each triple pattern.

$\mathcal{G}$ is always connected. Figure 3 shows the constraint graph for the join query given in Figure 1.

Before starting the pruning algorithm, we *initialize* each tp-node by loading the triples which match that triple pattern. In Section 3 we elaborated the construction of four types of BitMats viz. S-O and O-S for each P value, P-S for each O value and P-O for each S value. Assuming that a given query does not have any triple pattern with all variable positions (e.g. ?x ?y ?z) we initialize the BitMats associated with each triple pattern using the four types of stored BitMats (case of all-variable triple pattern is discussed at the end of this section). E.g., if the triple pattern in the query is of type (?s :p2 :o321) then we load only one row corresponding to ":p2" from the P-S BitMat created for ":o321". If the triple pattern is of type (?s :p6 ?o) then we load either the S-O or O-S BitMat created for ":p6". If ?s is a join variable and ?o is not, we load S-O BitMat and vice versa. If both, ?s and ?o, are join variables, then the decision depends on whether ?s will be processed before ?o. If a join over ?s is processed before ?o, we load S-O BitMat and vice versa.

If we have a triple pattern of type (:s2 ?p :o6), then first we decide whether P-S BitMat for ":o6" has less number of triples or P-O BitMat for ":s2" has less number of triples. If P-S BitMat has less number of triples, then we load the P-S BitMat by keeping only the bit corresponding to ":s2" in each row and mask out all other bits. Note that all these operations are done directly on the compressed BitMats. Thus at the end of initialization, each tp-node has a BitMat associated with it which contains only the triples matching that triple pattern. For example, $BitMat_1$ associated with *(?m rdf:type :movie)* has just a single row corresponding to "rdf:type" loaded from the P-S BitMat created for the object value ":movie".

Now we start the pruning algorithm. First, we consider an induced subgraph $\mathcal{G}_{jvar}$ of $\mathcal{G}$ containing only jvar-nodes. By the construction of graph $\mathcal{G}$, $\mathcal{G}_{jvar}$ is also always connected (see Figure 3). $\mathcal{G}_{jvar}$ can be cyclic or acyclic. Next, we embed a tree on $\mathcal{G}_{jvar}$ discarding any cyclic edges. To propagate the constraints on join variable bindings (Property 2), we walk over this tree from root to the leaves and backwards in breadth-first-search manner. At every jvar-node, we take intersection of bindings generated by its adjacent tp-nodes and after the intersection, drop the triples from tp-node BitMats as a result of the dropped bindings.

It can be seen that by the construction of graph $\mathcal{G}$ and following the tree over $\mathcal{G}_{jvar}$, constraints on the join variable bindings get propagated to other jvar-nodes through the tp-node BitMats (when the triples get dropped), and this propagation follows an alternating path between jvar-nodes and tp-nodes. This procedure is elaborated in Algorithm(1). A *topological sort* of an undirected tree is nothing but enumerating all the nodes from root to leaves in a breadth-first-search fashion.

For each node in the topological sorted list of join variables, we call *prune_for_jvar* (Lines 2 – 4 in Algorithm(1)). A topological sort ensures that a child jvar node always gets processed after all of its ancestors. The bitwise AND between folded bitarrays in *prune_for_jvar(J)* computes the *intersection* of all the bindings generated by the tp-nodes which contain J (Lines 2 – 5 in Algorithm(2)). According to Property 1, for any binding dropped in the intersection, the respective triples are removed from the BitMats associated with the tp-nodes which contain J using the *unfold* operation (Lines 6 – 9 in Algorithm(2)). *getDimension* returns the position of J in the BitMat of the triple pattern. For instance, *getDimension(?n, (?m :similar_to ?n))* can return *column* or *row* depending on whether it is an S-O or O-S BitMat.

Algorithm 1 Pruning Step

1: **queue** q = topological_sort($V(\mathcal{G}_{jvar})$)
2: **for each** J in q **do**
3: prune_for_jvar(J)
4: **end for**
5: **queue** q_rev = q.reverse() - leaves($\mathcal{G}_{jvar}$)
6: **for each** K in q_rev **do**
7: prune_for_jvar(K)
8: **end for**

Algorithm 2 prune_for_jvar(jvar-node J)

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

One such pass over all the jvar-nodes ensures that the constraints are propagated to the adjacent jvar-nodes from root to leaves of the tree. For a complete propagation of constraints, we traverse jvar-nodes second time by following the reverse order of the first pass (Lines 5 – 8 in Algorithm(1)). The leaves of the tree embedded on $\mathcal{G}_{jvar}$ appear last in queue q. Since they are processed last in the first traversal over the tree, in the second traversal, we directly start with the parent nodes of these leaves (Line 5 in Algorithm(1)). Notably, since we take *intersection* of the bindings in each pass, the number of triples in the tp-node's BitMat decrease monotonically as the constraints are propagated.

At the end of Algorithm(1), each tp-node contains a much reduced set of triples adhering to the constraints on join variable bindings. Typically, when $\mathcal{G}_{jvar}$ is acyclic, this set of triples is also *minimal*, i.e., each triple in the BitMat of a triple pattern is necessary to generate one or more final results. If $\mathcal{G}_{jvar}$ is cyclic, then the set of triples is not guarantied to be minimal. But in any case, the unwanted set of triples get dropped in the following phase of final result set generation.

Our pruning method closely resembles the idea of *semi-joins* [7, 6]. Semi-joins also build a query graph (QG) where the nodes of the graph are relations (tables) and an edge between the two nodes indicate a join between the two relations. A semi-join QG for a SPARQL join query can be reduced to $\mathcal{G}_{jvar}$ in BitMat's constraint graph by following simple transformation: each edge in the QG is a node with the join-variable name in $\mathcal{G}_{jvar}$, two nodes in $\mathcal{G}_{jvar}$ have an edge between them if the corresponding edges in the QG are incident on the same node in the QG. If a QG is *proper cyclic* [6], $\mathcal{G}_{jvar}$ is cyclic and if QG is a *tree query* then $\mathcal{G}_{jvar}$ is acyclic. Bernstein et al. have proved in [7, 6] that for the *tree queries*, semi-joins can *fully reduce* the database for a given query, i.e., at the end of a semi-join the database has minimal tuples, whereas *cyclic queries* cannot be guarantied to have full reducers. A formal proof of minimal triple set generation in case of an acyclic $\mathcal{G}_{jvar}$ in our method is rem-

iniscent of this proof. It is not included in this paper due to space constraints.

In our current implementation, we load the BitMat associated with each triple pattern at the beginning of query processing and then *never* seek a disk access in the entire lifespan of the query. This necessitates that for a query having n triple patterns it needs $\sum_{i=0}^{n} size(BitMat_i)$ amount of memory at the beginning. This poses limitations for queries having triple patterns with all variable positions (?x ?y ?z), as it is not feasible to load a BitMat for the all-variable tp-node containing the entire dataset in memory. Also, due to this condition, it can happen that for certain queries with highly selective triple patterns, the memory requirements of the conventional query processors are lesser than BitMat as they do not need to load entire indexes in memory to perform joins (e.g. RDF-3X). Notably, BitMat's memory requirement remains linear in terms of the triples associated with the triple patterns in the query, whereas for conventional query processors it can degrade polynomially for low-selectivity multi-join queries.

Simple optimizations

While performing the pruning step as elaborated before, we use some simple statistical optimization techniques.

Tree root selection: After *initialization*, in a join query with n triple patterns, we sort all the triple patterns first in the order of increasing number of triples associated with them. If the first triple pattern in this list has only one join variable, we pick this join variable as the *root* of the tree embedded on the graph $\mathcal{G}_{jvar}$ as described before. If it has more than one join variables, then we scan through the sorted list of triple patterns and find another triple pattern such that it shares a join variable with the first triple pattern (since constraint graph $\mathcal{G}$ is always connected for the queries without Cartesian joins, we are sure to find such a triple pattern). We then assign this shared join variable as the root of the tree embedded on $\mathcal{G}_{jvar}$. This method is similar to choosing tables with least number of triples to be joined first in the SQL joins.

Early stopping condition: While performing the pruning at each jvar-node, at any point if the $MaskBitArr_J$ contains all 0 bits, that is a direct evidence of the query generating empty set of results. If such a condition occurs we exit the query processing at that point telling that the query has 0 results. This avoids unnecessary further processing of other join variables and *fold/unfold* operations over BitMats.

4.2 Step 2 – Generating Final Results

After the pruning phase, we are left with a much reduced set of triples associated with each triple pattern. Intuitively, each BitMat of a triple pattern can be viewed as a compressed table in a relational database. Hence, one way of producing the results could have been to simply materialize these BitMats into tables and perform standard joins over them. But our goal is to avoid building intermediate join results by building a left-deep join tree; which precludes a 2-way sequence of joins as done in a typical SQL query processor. In our method, we build and output an entire resulting row of variable bindings, which is similar to *multi-way* joins.

Notably for this process, we use at most k size additional memory buffer, where k is the number of variables in the query (and hence the additional buffer size is negligible). We keep a map of bindings for all k variables at a time, output one result when all k variables are mapped, and proceed to generate the next result (hence we call these "streaming results").

Let us assume that a query has n triple patterns and N is the maximum number of triples in any of the n BitMats associated with the triple patterns. For simplicity, we denote $BitMat_i$ as the BitMat associated with the i^{th} tp-node ($tpnode_i$).

A simple brute force approach can be as follows: Choose say $BitMat_1$, pick a triple from it. This triple will generate bindings for the variables in $tpnode_1$. Store these bindings in the map. Next pick the first triple from $BitMat_2$ and generate bindings for the variables in $tpnode_2$. If $tpnode_2$ and $tpnode_1$ share one or more join variables, check the map if the variable bindings generated by both of them are the same, if not, pick a second triple from $BitMat_2$. Repeat this procedure until you get the variable bindings consistent with the ones stored in the map. Then consider $BitMat_3$ and repeat the same procedure as described above. Repeat this procedure until the last $BitMat_n$ in the query. If a triple in $BitMat_n$ generates valid variable bindings, such that all k variables are mapped to the bindings, output one result. Now start with $BitMat_1$ again and choose the second triple, store the bindings for the variables in $tpnode_1$, and repeat the same process. In general, while generating variable bindings from any $BitMat_i$, check all the variable bindings stored in the map. We can quickly see that the worst case complexity of such a brute force approach is $O(N^n)$.

Since BitMat is a fully inverted index structure, in practice we devise following method which speeds up the above procedure by several orders of magnitude: In general a BitMat having lesser number of triples generates lesser number of unique bindings for the variables in its tp-node. This means that in the final results of the query, these bindings will get repeated more often in different result rows than other bindings (just like the product of two columns where first column has lesser number of rows than the other – values from the first column get repeated more often in the product). Making use of this fact, we choose a BitMat as $BitMat_1$, which has the least number of triples, to be processed first (similar to the way of choosing the table having least number of triples to join first), generate bindings for the variables in $tpnode_1$, and store them in the map. Next instead of picking $BitMat_2$ randomly, we pick a $tpnode_2$ which shares a join variable with $tpnode_1$. Depending on the variable bindings stored in the map, we directly locate the triples which can satisfy these bindings inside $BitMat_2$. Recall that BitMat being a completely inverted index structure, it is easy to locate specific triples. If no such triple exists in $BitMat_2$, we discard the variable bindings in the map, go back to $BitMat_1$, and pick the second triple from it to generate new bindings (this can happen only in case of a cyclic $\mathcal{G}_{jvar}$). If $BitMat_2$ generates variable bindings consistent with $BitMat_1$, pick $tpnode_3$ which shares join variables either with $tpnode_1$ or $tpnode_2$. Considering the constraint graph given in Figure 3, let $\mathcal{G}_{tp}$ be an induced subgraph of $\mathcal{G}$ having only tp-nodes and edges between them. We make use of $\mathcal{G}_{tp}$ to make the choice of the next tp-node at every step. Hence it can be seen that after one walk over all the tp-nodes of $\mathcal{G}_{tp}$, if the map has all k variables mapped to bindings, we output one result. The procedure is repeated again until all the triples in $BitMat_1$ are exhausted.

Presently the final phase (Step 2) always projects out bindings of all variables in the query unless it is a "star join". However, in the future, depending on the nature of the constraint graph $\mathcal{G}$ for a given query and the variable bindings asked by the SELECT clause, it might not be required to traverse $\mathcal{G}_{jvar}$ twice (ref. Section 4.1) thereby further improving the overall query processing time.

Note that all the procedures described previously work on a compressed BitMat. Since the pruning phase only reduces the number of triples in the BitMat monotonically, the memory requirement of the query processor goes on reducing as the pruning progresses and the final phase of result generation doesn't build join tables.

To conclude the description of our procedure, we would like to point out certain key differences of our query processing algorithm from the typical bitmap index joins. BitMat's structure is similar to the idea of compressed bitmap indexes which are widely used to improve joins in OLAP data warehousing techniques [17, 12, 20]. But an SQL join between multiple tables over different columns cannot always make use of the bitmap indexes for the later joins. This is due to the fact that after the first level of join, a relational query processor has to materialize the results of the previous join to carry out the next join and this materialized table does not always have indexes formed on it (unless join-indexes are precomputed based on heuristics). As opposed to that, BitMat's pruning and final result generation steps *always* use compressed BitMats, without materializing the intermediate join results.

5. EVALUATION

BitMat structure and query algorithm is developed in C and is compiled using g++ with -O3 optimization flag. For the experiments we used a Dell Optiplex 755 PC having 3.0 GHz Intel E6850 Core 2 Duo Processor, 4 GB of memory, running 64 bit 2.6.28-15 Linux Kernel (Ubuntu 9.04 distribution), with 7 GB of swap space on a 7200 rpm disk with 1 TB capacity.

5.1 Choice of competitive RDF stores

We had a wide choice to select the systems for competitive evaluation due to the availability of numerous RDF triplestores. We experimented with Hexastore[6], Jena-TDB, RDF-3X, and MonetDB. Out of these we chose RDF-3X (v0.3.3) and MonetDB (v5.14.2) – latest versions – for our evaluation, as they could load a large amount of RDF data, gave better performance than others, and are open-source systems used by the research community for performance intensive RDF query execution.

Like BitMat, RDF-3X maps strings/URIs in RDF data to integer IDs and mainly operates on these IDs, building the entire result of a query in the integer ID format. It converts the IDs to strings using their dictionary mapping just before outputting the results in a user readable format. We observed that RDF-3X was taking significant amount of time to convert IDs into strings; in certain cases it took even more time for this conversion than the time taken for the core query execution. Current BitMat system doesn't support a formal SPARQL query parser interface and the interface to output the results in the string format is still under preliminary development. Hence for a fair comparison, we disabled the ID to string mapping in RDF-3X, which improved their query times a lot. All the RDF-3X query times reported in this paper are without their ID to string mapping.

For a fair comparison, we loaded MonetDB[7] by inserting the integer IDs generated out of BitMat dictionary mapping (ref. Section 3). Hence essentially all the MonetDB queries were performed on S, P, Os as integer IDs. We created separate predicate tables in MonetDB by inserting the respective triples by ordering on S-O values [21] and used these predicate tables in the query whenever there is a bound predicate in the triple pattern instead of the giant triple-table containing all the triples.

5.2 Choice of datasets and queries

We chose UniProt dataset with 845,074,885 triples, 147,524,984 subjects, 95 predicates, and 128,321,926 objects [23], which is a protein dataset. We also generated a dataset using LUBM [13] – a synthetic data generator provided by Lehigh University – with over 10,000 universities which gave 1,335,081,176 unique triples with 217,206,845 subjects, 18 predicates, and 161,413,042 objects. LUBM is widely used by the Semantic Web community for benchmarking triplestores.

For UniProt dataset we used 6 out of 8 queries published by RDF-3X in [16] (Q7-Q10, Q12, Q13 in our list, leaving out 2 queries which have 'all-variable' triple patterns). To increase the diversity, we also included 5 more queries (out of 9) published by the UniProt dataset owners [19] (Q1, Q4-Q6, Q11 in our list). We removed the FILTER condition in the original Q1 as currently it is not supported by our query processor and had to modify a 'bound position' in Q5, Q11 as that value did not exist in the dataset. We modified two of the RDF-3X queries by removing some bound positions to reduce the selectivity of triple patterns (Q2, Q3 in our list). For the LUBM dataset, OpenRDF has published a list of queries [18]. But many of these queries are simple 2-triple pattern queries or they are quite similar to each other. Hence we chose 7 representative queries out of this list. All the queries are listed in Appendix A.

5.3 Discussion

For the evaluation, we measured the following parameters: (i) query execution times (cold and warm cache). This is an end-to-end time counted from the time of query submission to the time including outputting the final results. For cold cache we dropped the file systems caches using `/bin/sync` and `echo 3 > /proc/sys/vm/drop_caches`, (ii) initial number of triples – the sum of triples matching each triple pattern in the query, and (iii) the number of results. The evaluation is given in Tables 1, 2, and 3. Query times were averaged over 10 consecutive runs. *Geometric mean** is the geometric mean of the query times excluding the ones on which RDF-3X failed to complete processing.

Note that our current BitMat query processing system does not use any sophisticated cache management (like MonetDB) and also does not `mmap` datafiles into the memory (like RDF-3X). Due to this, as opposed to RDF-3X and MonetDB, in most of the queries the difference between our cold and warm cache times was not very high.

After the evaluation, we could classify the queries into 3 categories – queries where BitMat clearly excelled over

[6]We obtained compiled binaries of Hexastore from the authors.

[7]MonetDB was compiled using "--enable-optimization" flag.

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

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		

Table 3: Evaluation – LUBM 1.33 billion triples (time in seconds, best times are boldfaced)

	Q1	Q2	Q3	Q4	Q5	Q6	Geom. Mean	Geom. Mean* (without Q1)
Cold cache								
BitMat	**51.21**	**2.71**	**6.56**	2.45	0.503	3.81	**4.0285**	**2.4227**
MonetDB	109.35	27.17	455.23	34.12	18.89	14.6	48.3195	41.0377
RDF-3X	Aborted	34.868	2328.753	**0.588**	**0.425**	**1.129**	N/A	7.4474
Warm cache								
BitMat	**48.57**	**2.11**	**1.94**	0.686	0.27	2.85	**2.1719**	1.1666
MonetDB	96.65	6.56	398.46	3.209	0.566	0.542	7.9301	4.8094
RDF-3X	Aborted	29.033	2028.6855	**0.0024**	**0.0029**	**0.1814**	N/A	**0.5947**
#Results	2528	10,799,863	0	10	10	125		
#Initial triples	165,397,764	224,805,759	219,416,877	438,912,513	3,000,966	9,100,649		

both RDF-3X and MonetDB (in both cold and warm cache times), queries where BitMat did better than one of the systems or the reported query times were comparable to the other systems, and queries where BitMat's performance was worse than both RDF-3X and MonetDB.

Queries of the first type are – Q1, Q2 of UniProt and Q1, Q2, Q3 of LUBM. Notably, UniProt Q1, Q2 had a high number of initial triples and the join results were quite unselective. As was our initial conjecture, BitMat did much better on such queries than RDF-3X and MonetDB. On the other hand, LUBM Q1 and Q3 were more complex queries having a high number of initial triples associated with the triple patterns, but the final number of results were quite small (2528 and 0 respectively). For these queries, the initial selectivity of the triple patterns and selectivity of the intermediate join results were quite low, but together they gave highly selective results (these queries have *cyclic* dependency among join variables – ref. Section 4.1). For these queries BitMat was upto 3 orders of magnitude faster than RDF-3X and MonetDB due to its way of producing join results without materializing the intermediate join tables. RDF-3X aborted while executing UniProt Q1 and LUBM Q1 as the system ran out of its physical memory and swap space. We executed the same queries on RDF-3X on a higher configuration server having 16 GB physical memory. RDF-3X processed UniProt Q1 in 858.464 sec; was observed to consume ∼11 GB resident memory. For LUBM Q1, RDF-3X took 1613.178 sec and the peak memory consumption was ∼11 GB. For both queries BitMat took 448.169 and 50.70 sec, and consumed ∼2.6 GB and ∼3 GB respectively on the same server.

A "star join" query was the one where many triple patterns joined on one variable, the query had only one join variable, and that variable got projected in the final results. LUBM Q2, Q4, Q5 were star-join queries. For star-joins BitMat worked much better, because our query processor doesn't need to load the BitMats of all the triple patterns in memory. It just loads the pre-computed row or column bitarrays of each BitMat associated with the triple pattern (ref. Section 3). The final result generation phase consists of just listing out the 1-bit positions from the bitwise AND of the loaded bitarrays (similar to the bitmap index joins).

Notably, although Q2 has only two triple patterns in it, each has very low selectivity and the query generates a lot of results compared to query Q4 and Q5.

RDF-3X did very well on the UniProt queries Q7-Q10, Q12, Q13. These queries have a lot of triple patterns, with many having bound predicate and object positions which make them highly selective. Also many of these triple patterns join on one variable where RDF-3X's method of *Sideways Information Passing* worked much better. The number of results produced by these queries were highly selective too (less than 30 results for 5/6 queries).

In the case of UniProt Q4, Q5, Q7-Q10 BitMat did better than MonetDB (cold cache), but RDF-3X still outperformed BitMat (Q7-Q10 are the queries published by RDF-3X). For Q6 the margin of difference between cold cache times of RDF-3X and BitMat was quite small. But in case of Q5, the difference was quite high. Further dissection of BitMat query processing times revealed that *initialization* and *pruning* phases were very fast, but more than 90% of the time was spent in the last phase of the result construction. The reason behind this is – our current data structures and result enumeration algorithm are not tuned to exploit the "locality" in memory while generating the final results. This query has only one join variable but all the variables in the query get projected in the results. On the other hand, for UniProt Q11-Q13, more than 90% of the query processing time was spent in the *initialization* to load the BitMats associated with each triple pattern (ref. Section 4.1). In the future, this effect can be alleviated by implementing a "lazy loading" of the BitMats associated with the tp-nodes – instead of loading all the BitMats at the beginning, one can wait until the very first join and then load only the required portion of the BitMat in the *unfold* operation.

To summarize the results – it was evident that for complex join queries with low-selectivity intermediate results, BitMat outperformed both RDF-3X and MonetDB by a significant margin. Although for queries with highly selective triple patterns generating fewer results, RDF-3X and MonetDB performed better. This re-emphasizes our initial goal of targeting low-selectivity queries with our novel query processing algorithm.

In view of these results, we would like to mention one specific LUBM query which turned out be an *outlier* (LUBM Q7). RDF-3X aborted due to the system running out of memory on the Dell PC. BitMat took several hours to process this query, although the processor clock showed that the query spent only ∼200 seconds actually executing on the processor. MonetDB processed it in 449.048 sec. For further investigations, we evaluated this query on the server having 16 GB of memory and we found the following:

- BitMat finished processing this query in 139.94 sec. The peak resident memory consumption was reported to be 6.3 GB, and on an average the process consumed 5 GB of the resident memory. The BitMat associated with tp-node *'?x ub:takesCourse ?z'* was very large, ∼3.4 GB, having 288,017,530 triples (22% of the total triples) in it – largest among all the predicates. Thus, on the 4 GB Dell PC, BitMat process spent a lot of time in the kernel waiting for the pages to be allocated. MonetDB handled this situation well due to its better cache-memory management.

- MonetDB processed this query in 136.082 sec on the 16 GB memory server, but the peak resident memory consumption was 9 GB, and on an average the process consumed 8.6 GB of resident memory.

- RDF-3X processed the same query in 66.139 sec on the same server, but its peak resident memory consumption was 14 GB and on an average it consumed 13 GB of resident memory for most of the lifespan of the query.

This query has 442,351,492 initial triples associated with it – largest among the listed LUBM queries – and generates 439,994 results. We believe that with a "lazy loading" strategy along with "proactive cache management" BitMat would be able to handle these type of queries in a better manner in future.

The on disk size of all $2|V_p| + |V_s| + |V_o|$ BitMats (ref. Section 3) was 48 GB and 67 GB for UniProt and LUBM respectively and corresponding LZ77 compressed dictionary mappings were 3.2 GB and 1.8 GB. These BitMat sizes include the size of the meta-file too. But note that for any given query with n triple patterns, the runtime memory requirement is just $\sum_{i=1}^{n} size(BitMat_i)$; which is typically a much smaller fraction of the total datafile size. For RDF-3X and MonetDB the on-disk size of datafiles were 42 GB and 16 GB for UniProt, and 70 GB and 25 GB for LUBM. The sizes of raw RDF Ntriple files of UniProt and LUBM were 205 GB and 451 GB respectively.

We used an external Perl script to parse the raw triples and build string to ID dictionary mapping. It took ∼12 hours to parse and build dictionary mappings of LUBM data and ∼9 hours for UniProt data. After parsing the data, the S-O, O-S, P-S, P-O BitMats for UniProt and LUBM were built in 41 and 56 minutes respectively. This process is much faster than parsing due to our method of building the compressed bit-row of a BitMat directly without building an uncompressed array first using the ID based triples sorted on their S, P, O positions (details of this process are omitted due to space constraints).

6. CONCLUSION AND FUTURE WORK

In this paper we demonstrated a novel method of processing RDF join queries, following a simple principle of keeping the data compressed as much as possible, without building intermediate join tables, and producing the final results in a streaming fashion. Our evaluation using the state-of-the-art RDF stores like RDF-3X and MonetDB showed that while RDF-3X and MonetDB gave better performance on highly selective queries, BitMat gave much superior performance on low-selectivity queries, where sophisticated query optimization techniques did not fetch a lot of benefits. BitMat could deliver over 3 orders of magnitude better performance for some of the queries (e.g., LUBM Q3, warm cache).

Notably, working on the compressed data fetches benefits when the size of the underlying data is higher and when the selectivity of triple patterns in the query and intermediate join results is lower. This is due to our query processing algorithm which keeps the runtime memory footprint small. On the other hand, for queries with highly selective triple patterns, processing the joins in a conventional manner can be more beneficial.

BitMat system is a prototype implementation of our query processing algorithm. Since BitMat's basic data structure resembles compressed bitmap indexes, in the future it is

possible to develop a hybrid system having BitMat's query processing algorithm and the conventional query processor. The system can choose the method of processing the query based on heuristics and selectivity of the triple patterns in the query. Along with these avenues, in the future, we plan to improve the system by further optimizing our algorithm.

Acknowledgments

We would like to thank Dr. Jagannathan Srinivasan for his invaluable inputs in the initial phase of the work and Thomas Neumann for his timely help in working with RDF-3X sources.

7. REFERENCES

[1] Jena TDB. http://jena.sourceforge.net/TDB/.
[2] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating Compression and Execution in Column Oriented Database Systems. In *SIGMOD*, 2006.
[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management using Vertical Partitioning. In *PVLDB*, 2007.
[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.
[5] M. Atre and J. A. Hendler. BitMat: A Main-memory Bit-Matrix of RDF Triples. In *SSWS workshop at ISWC*, 2009.
[6] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1), 1981.
[7] P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal of Computing*, 10(4), 1981.
[8] M. Cai and M. Frank. RDFPeers: A Scalable Distributed RDF Repository based on a Structured Peer-to-Peer Network. In *WWW*, 2004.
[9] R. Cyganiak. A Relational Algebra for SPARQL. *Technical Report, HP Laboratories Bristol*, (HPL-2005-170), 2005.
[10] O. Erling. Virtuoso, 2006. http://virtuoso.openlinksw.com/-wiki/main/Main/VOSBitmapIndexing.
[11] M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *ISWC*, 2005.
[12] T. Johnson. Performance Measurements of Compressed Bitmap Indices. In *PVLDB*, 1999.
[13] LUBM. http://swat.cse.lehigh.edu/projects/lubm/.
[14] A. Matono, S. M. Pahlevi, and I. Kojima. RDFCube: A P2P-based Three-dimensional Index for Structural Joins on Distributed Triple Stores. In *DBISP2P at VLDB*, 2006.
[15] T. Neumann and G. Weikum. RDF3X: a RISC style Engine for RDF. In *PVLDB*, 2008.
[16] T. Neumann and G. Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.
[17] P. O'Neil and G. Graefe. Multi-Table Joins Through Bitmapped Join Indices. In *SIGMOD Record*, volume 24, September 1995.
[18] OpenRDF LUBM Queries. http://repo.aduna-software.org/viewvc/org.openrdf/?pathrev=6875.
[19] UniProt RDF Queries. http://dev.isb-sib.ch/projects/expasy4j-webng/query.html#examples.
[20] S. Sarawagi. Indexing OLAP data. *Data Engineering Bulletin*, 20(1), 1997.
[21] L. Sidirourgos, R. Goncalves, M. Kersten, et al. Column-store Support for RDF Data Management: not all swans are white. In *PVLDB*, 2008.
[22] O. Udrea, A. Pugliese, and V. Subrahmanian. GRIN: A Graph Based RDF Index. In *AAAI*, 2007.
[23] UniProt RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.
[24] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *PVLDB*, 2008.

APPENDIX

A. QUERIES

A.1 UniProt queries

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX uni: <http://purl.uniprot.org/core/> PREFIX uni2: <http://purl.uniprot.org/>
Q1: SELECT ?modified ?author ?citation ?title ?protein WHERE { ?protein rdf:type uni:Protein . ?protein uni:/modified ?modified . ?protein uni:citation ?citation . ?citation uni:author ?author . ?citation uni:title ?title .}
Q2: SELECT ?a ?vo ?b ?ab ?x ?z ?p WHERE { ?a uni:encodedBy ?vo . ?a schema:seeAlso ?x . ?b uni:sequence ?z . ?b uni:replaces ?p . ?b rdf:type uni:Protein . ?a uni:replaces ?ab . ?ab uni:replacedBy ?b . }
Q3: SELECT ?a ?x ?vo ?b ?ab ?z ?p WHERE { ?a schema:seeAlso ?x . ?a uni:encodedBy ?vo . ?b uni:sequence ?z . ?b uni:replaces ?p . ?b uni:modified "2008-07-22" . ?b rdf:type uni:Protein . ?a uni:replaces ?ab . ?ab uni:replacedBy ?b . }
Q4: SELECT ?name ?gene ?protein WHERE { ?protein rdf:type uni:Protein . ?protein uni:encodedBy ?gene . ?gene uni:name "hup" . ?protein uni:name ?name }
Q5: SELECT ?related ?p ?protein WHERE { ?protein rdf:type uni:Protein . ?protein ?p uni2:keywords/482 . ?protein rdfs:seeAlso ?related . }
Q6: SELECT ?p2 ?interaction ?p1 WHERE { ?p1 uni:enzyme uni2:enzyme/2.7.7.- . ?p1 rdf:type uni:Protein . ?interaction uni:participant ?p1 . ?interaction rdf:type uni:Interaction . ?interaction uni:participant ?p2 . ?p2 rdf:type uni:Protein . ?p2 uni:enzyme uni2:enzyme/3.1.3.16 .}
Q7-Q10: Same as Q1, Q3, Q6, Q8 in [16] respectively.
Q11: SELECT ?aa ?s ?protein WHERE { ?protein uni:organism uni2:taxonomy/287. ?protein rdf:type uni:Protein . ?protein uni:sequence ?s . ?s rdf:value ?aa . }
Q12-Q13: Same as Q5 and Q7 respectively in [16],
Note: We project all the variables in the queries Q7-Q10, Q12, Q13 above.

A.2 LUBM queries

PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
Q1: SELECT ?x ?y ?z WHERE { ?z ub:subOrganizationOf ?y . ?y rdf:type ub:University . ?z rdf:type ub:Department . ?x ub:memberOf ?z . ?x rdf:type ub:GraduateStudent . ?x ub:undergraduateDegreeFrom ?y . }
Q2: SELECT ?x WHERE { ?x rdf:type ub:Course . ?x ub:name ?y . }
Q3: SELECT ?x ?y ?z WHERE { ?x rdf:type ub:Undergraduate-Student. ?y rdf:type ub:University . ?z rdf:type ub:Department . ?x ub:memberOf ?z . ?z ub:subOrganizationOf ?y . ?x ub:under-graduateDegreeFrom ?y . }
Q4: SELECT ?x WHERE { ?x ub:worksFor <http://www.-Department0.University0.edu> . ?x rdf:type ub:FullProfessor . ?x ub:name ?y1 . ?x ub:emailAddress ?y2 . ?x ub:telephone ?y3 . }
Q5: SELECT ?x WHERE { ?x ub:subOrganizationOf <http-://www.Department0.University0.edu> . ?x rdf:type ub:Research-Group}
Q6: SELECT ?x ?y WHERE { ?y ub:subOrganizationOf <http-://www.University0.edu> . ?y rdf:type ub:Department . ?x ub:worksFor ?y . ?x rdf:type ub:FullProfessor . }
Q7: SELECT ?x ?y ?z WHERE { ?y ub:teacherOf ?z . ?y rdf:type ub:FullProfessor . ?z rdf:type ub:Course . ?x ub:advisor ?y . ?x rdf:type ub:UndergraduateStudent . ?x ub:takesCourse ?z}