BitMat: A Main Memory Bit-matrix of RDF Triples

Medha Atre and James A. Hendler

Tetherless World Constellation,
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy NY USA
{atrem, hendler}@cs.rpi.edu

Abstract. BitMat is a main memory based bit-matrix structure for representing a large set of RDF triples, designed primarily to allow processing of conjunctive triple pattern (join) queries. The key aspects are as follows: i) its RDF triple-set representation is compact compared to conventional disk-based and existing main-memory RDF stores, ii) basic join query processing employs logical bitwise AND/OR operations on parts of a BitMat, and iii) for multi-joins, intermediate results are maintained in the form of a BitMat containing candidate triples without complete materialization, thereby ensuring that the intermediate result size remains bounded across a large number of join operations, provided there are no Cartesian joins. We present the key concepts of the BitMat structure, its use in processing join queries, describe our experimental results with RDF datasets of different sizes (from 0.2 to 47 million), and discuss the use case scenarios.

1 Introduction

RDF [4] and SPARQL [16] are gaining importance as semantic data is increasingly becoming available in the RDF format. The growing scale of RDF data necessitates novel ways of storing and querying this data in a compact form. To handle this large scale RDF data, numerous systems are being developed [13, 23, 1]. Many of these systems are implemented as a straight-forward extension of relational database systems and SQL querying techniques. These systems can be broadly classified as persistent disk-based and main-memory-based systems. The work described in this paper belongs to the latter category proposing a main-memory RDF triple store.

Most of the other RDF store systems depend on building efficient auxiliary indexes on the RDF data and using them either for specific type of queries or to improve the overall query performance. In contrast, in our approach, BitMat which is an compressed inverted index structure itself makes up the primary storage for RDF triples. Our proposed query processing algorithm voids the need to uncompress this data at any point during query processing.

Generic relational data can have varied dimensions (i.e. number of columns), and hence the SQL query processing algorithms have to encompass this nature of relational data. As opposed to that, an RDF triple is a fixed 3-dimensional (S, P, O) entity and the dimensionality of SPARQL conjunctive triple pattern queries is also fixed (which depend on the number of conjunctive patterns in the query). Hence while building the BitMat structure and query processing algorithms, we made use of this fact.

In essence, BitMat is a 3-dimensional bit-cube, in which each cell is a bit representing a unique triple denoting the *presence* or *absence* of that triple by the bit value 1 or 0. This bit-cube is flattened in a 2-dimensional bit matrix for implementation purpose. Figure 1 shows an example of a set of RDF triples and the corresponding BitMat representation.

Subject	**Predicate**	**Object**
:the_matrix	:released_in	"1999"
:the_thirteenth_floor	:released_in	"1999"
:the_thirteenth_floor	:similar_plot_as	:the_matrix
:the_matrix	:is_a	:movie
:the_thirteenth_floor	:is_a	:movie

Distinct subjects: [:the_matrix, :the_thirteenth_floor]
Distinct predicates: [:released_in, :similar_plot_as, :is_a]
Distinct objects: [:the_matrix, "1999", :movie]

	:released_in	:similar_plot_as	:is_a
:the_matrix	0 1 0	0 0 0	0 0 1
:the_thirteenth_floor	0 1 0	1 0 0	0 0 1

Note: Each bit sequence represents sequence of objects (:the_matrix, "1999", :movie)

Fig. 1. BitMat of sample RDF data

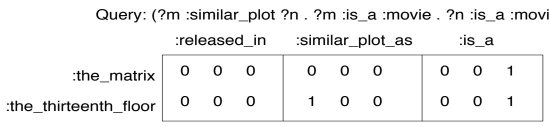
If the number of distinct subjects, predicates, and objects in a given RDF data are represented as sets V_s, V_p, V_o, then a typical RDF dataset covers a very small set of $V_s \times V_p \times V_o$ space. Hence BitMat inherently tends to be very sparse. We exploit this sparsity to achieve compactness of the BitMat by compressing each bit-row using D-gap compression scheme [7][1].

Since *conjunctive triple pattern* (join) queries are the fundamental building blocks of SPARQL queries, presently our query processing algorithm supports only those. These queries are processed using bitwise AND, OR operations on the compressed BitMat rows. Note that the bitwise AND, OR operations are directly supported on a compressed BitMat thereby allowing memory efficient execution of the queries. At the end of the query execution, the resulting filtered triples are returned as another BitMat (i.e. a query's answer is another result BitMat). This process is explained in Section 4.

Figure 2 shows the conjunctive triple pattern *for movies that have similar plot* and the corresponding result BitMat. Unlike the conventional RDF triple stores,

[1] E.g. In D-gap compression scheme a bit-vector of "0011000" will be represented as [0]-2,2,3. A bit-vector of "10001100" will be represented as [1]-1,3,2,2.

where size of the intermediate join results can grow very large, our BitMat based multi-join[2] algorithm ensures that the intermediate result size remains bounded, (at most to (n * size of the original BitMat), where n is the number of triple patterns in the query), across any number of join operations (provided there are no Cartesian joins).



Query: (?m :similar_plot ?n . ?m :is_a :movie . ?n :is_a :movie)

	:released_in			:similar_plot_as			:is_a		
:the_matrix	0	0	0	0	0	0	0	0	1
:the_thirteenth_floor	0	0	0	1	0	0	0	0	1

Fig. 2. Result BitMat of a sample query

A conventional SPARQL join query engine produces zero or more *matching subgraphs* (each resulting row with variable bindings identifies a matching subgraph). BitMat join processing returns a set of distinct triples in the result BitMat that together form all the matching subgraphs. Currently we are working on an algorithm to enumerate all the matching subgraphs – which is not presented here. But even without this last phase of result generation, current query processing algorithms can be used for:

– *'EXISTS'* or *'ASK'* queries as very large multi-joins can be performed in memory using BitMat (existence of one or more 1 bits in the resulting BitMat indicates that the query will produce at least one matching subgraph).
– As a precursor to an in-memory query processing engine (e.g. Jena-ARQ [2]) to identify the result triples from a large triple set.

BitMat is designed specifically to process conjunctive triple pattern queries and presently the query processing interface does not support full SPARQL syntax or other SPARQL constructs.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the related work. Section 3 describes the BitMat structure and its composition in greater details. Sections 4 and 5 describe the basic algorithm of single join procedure and an algorithm for multi-join based on the single-join algorithm respectively. Section 6 gives our evaluation of the system, and Section 7 concludes the paper with strengths and weaknesses of the present structure of the BitMat and query processing system.

2 Related Work

The structure of a BitMat is somewhat similar to the idea of bitmap indexes, which are used in relational database systems and more recently by the Virtuoso RDF store [8] to efficiently process queries over low cardinality data. The key

[2] Multi-join is a conjunctive triple pattern with two or more triple patterns having two or more join variables and a single join has two triple patterns with only one join variable.

difference is that BitMat's query processor always operates on the compressed BitMat without uncompressing it anytime. This is not possible with a traditional SQL based query processor employing bitmap indexes over multi-joins.

In addition to these, there are various systems being developed for processing RDF data. Some notable ones are – Hexastore [23], RDF-3X [13, 14], BRAHMS [10], GRIN [21], SwiftOWLIM [20], Jena-TDB [1] etc.

Out of these, Hexastore and RDF-3X exploit the nature of RDF data by creating 6-way indexes (SPO, SOP, PSO, POS, OSP, OPS) on it. RDF-3X goes one step further by compressing these indexes and organizing them as clustered B+-trees. BRAHMS and GRIN mainly use their system for path-based queries on RDF graph. But they have not published results on very large RDF graphs, putting the scalability of their system under question. BRAHMS have used LUBM data of only 6 million triples and GRIN has published results for only upto 17,000 triples[3].

The notable difference between these systems and BitMat is that – BitMat's conjunctive triple pattern query processing algorithm which controls the intermediate memory utilization in a large multi-join query.

The structure that comes closest to BitMat is RDFCube [12], which also builds a 3D cube of subject, predicate, and object 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. It is primarily used to reduce the network traffic for processing join queries over a distributed RDF store (RDF-Peers [5]) by narrowing down the candidate triples. In contrast, BitMat structure maintains unique mapping of a triple to a single bit element and further compresses the BitMat. Our goal here is to represent large RDF triple-sets with a compact in-memory representation and support a scalable multi-join query execution *completely in-memory*.

SPARQL query language, which is structurally quite similar to the SQL query language [6], is being studied specifically with respect to the join processing [9, 19] and query benchmarking [18]. In contrast to the conventional SPARQL query processing scheme, we employ a different approach for multi-joins as elaborated in Section 5.

3 BitMat Concepts

A bit-cube of RDF triples is a 3-dimensional structure with subject (S), predicate (P), and object (O) dimensions. Individual cell is a single bit, and 1 or 0 value of the bit represents presence or absence of the triple. This conceptual bit-cube can be represented as a concatenation of (S,O) or (O,S) matrices for all the distinct predicates thereby forming a *mat of bits*, BitMat. Concatenation done along the subject dimension is referred to as a *subject BitMat* and concatenation done along the *object dimension* is referred to as an object BitMat. Altogether, there are 6 ways of flattening a bit-cube into a BitMat (as is the case of six-way

[3] GRIN focuses on path-like queries which are not even expressible in current SPARQL query language.

indexes on the RDF data). For the current set of experiments, we have used subject BitMats. Exploring other structures of BitMat is a part of future work.

3.1 BitMat Structure

BitMat is constructed from a set of RDF triples as follows: Let V_s, V_p, and V_o represent the sets of distinct subject, predicate, and object values occurring in a RDF triple set. Let V_{so} represent the $V_s \cap V_o$ set. These four sets are mapped to the integer sequence based identifiers as shown below:

- *Common subjects and objects*: Set V_{so} is mapped to a sequence of integers: 1 to $|V_{so}|$ in that order.
- *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 $|V_o|$.

Basically, each ID-space is treated independently with the exception that URIs which appear as subjects as well as objects are assigned the same sequence identifiers. This is done to facilitate the subject-object (S-O) cross dimensional join as discussed in Section 4. Cross-dimension joins over subject-predicate (S-P) or predicate-object (P-O) dimensions are rare in the context of *assertional* RDF data. Since the large scale RDF data available on the web is predominantly assertional, presently we do not handle S-P or P-O cross dimensional joins.

The above mapping is a direct representation of the position of those triples in the BitMat structure. For the example given in Section 1, *:the_matrix* as a subject is mapped to *1*, *:the_matrix* as an object is also mapped to *1*, *:the_thirteenth-_floor* is mapped to *2*, *:similar_plot_as* is mapped to *2* etc. Hence triple (*:the_thirteenth_floor :similar_plot_as :the_matrix*) is represented as (*2 2 1*) indicating to set the first bit (O-position) in the second row (S-position) of the second (S,O) matrix (P-position) (see Figure 1). A complete BitMat is built this way by setting the bit corresponding to each encoded RDF triple. Although this is the conceptual structure of a BitMat, we build the compressed BitMat directly from the encoded triple set as explained further in Section 6.

3.2 BitMat Operations

The process of evaluating conjunctive triple pattern (join) queries is carried out with three primitive operations on a BitMat. They are as follows:

(1) **Filter:** Filter operation is represented as *'filter(BitMat, TriplePattern) returns BitMat'*. It takes an input BitMat and returns a new BitMat which contains only triples that satisfy the *TriplePattern*.

Effectively, *filter* operation on a BitMat involves clearing the bits of the triples that are *filtered out*. For example, a triple pattern with only bound subject value like *filter(BitMat, ':s1 ?p ?o')*, clears all the bits in all the rows other than the row corresponding to the bound subject value *:s1* etc.

(2) **Fold:** Fold function represented as *'fold(BitMat, RetainDimension) returns bitarray'* folds the input BitMat along the two dimensions other than the *RetainDimension*.

For example, if *RetainDimension* is set to *'object'*, then BitMat is folded along the subject and predicate dimensions resulting into a single bitarray. Intuitively, bits set to 1 in this bitarray indicate the presence of *at least* one triple with the object corresponding to that position in the given BitMat. Typically *fold* is called on the BitMat returned by *filter*. E.g. *fold(filter(BitMat, ':s1 ?p ?o'), 'object')*.

(3) **Unfold:** Specified as *'unfold(BitMat, MaskBitArray, RetainDimension) returns BitMat'* takes a BitMat, a bitarray, and *unfolds* the bitarray 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 a BitMat are cleared. Typically *MaskBitArray* is generated by a bitwise AND of the bitarrays returned by *fold* operations before. E.g. *unfold(BitMat, '101001', 'predicate')* would result in clearing all the bits in second, fourth, and fifth (S,O) matrices which correspond to predicates mapped to $\{2, 4, 5\}$.

Filter, *fold*, and *unfold* operations are implemented to operate on a compressed BitMat without uncompressing it.

4 Single Join Processing

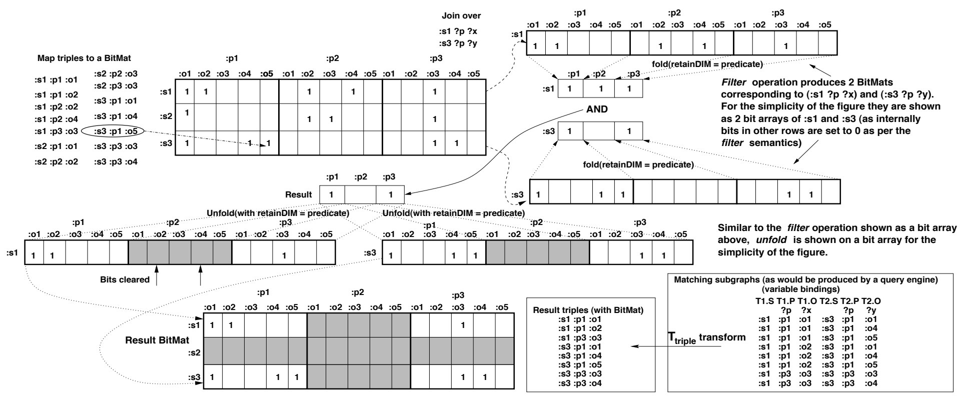
A conventional SPARQL join query processing engine produces zero or more *matching subgraphs* (each resulting row with the variable bindings identifies a matching subgraph) (see Figure 3). The query being evaluated is (:s1 ?p ?x . :s3 ?p ?y). Intuitively, every resulting matching subgraph is a proper subgraph of the original RDF graph G, which satisfies the SPARQL query graph pattern (provided there are no Cartesian joins).

BitMat based join algorithm is given in Algorithm 1 and is elaborated as follows.

Algorithm 1 BitMat_SingleJoin(BM, tp_1, tp_2) returns BitMat

```
1: Let $BM$ be the BitMat of the original triple-set
2: Let $tp_1$ and $tp_2$ be the two triple patterns in the join
3: /* filter and fold */
4: $BM_{tp1}$ = filter($BM$, $tp_1$)
5: $BM_{tp2}$ = filter($BM$, $tp_2$)
6: $BitArr_1$ = fold($BM_{tp1}$, $RetainDimension_{tp1}$)
7: $BitArr_2$ = fold($BM_{tp2}$, $RetainDimension_{tp2}$)
8: $BitArr_{res}$ = $BitArr_1$ AND $BitArr_2$
9: $BM_{tp1}$ = unfold($BM_{tp1}$, $BitArr_{res}$, $RetainDimension_{tp1}$)
10: $BM_{tp2}$ = unfold($BM_{tp2}$, $BitArr_{res}$, $RetainDimension_{tp2}$)
11: /* Produce the final result BitMat */
12: Let $BM_{res}$ be an empty BitMat
13: $BM_{res}$ = $BM_{tp1}$OR $BM_{tp2}$
```

On lines 4, 5 *filter* operation is used to get two BitMats containing only triples satisfying the first and second triple pattern respectively. This resembles the *selection* operator used in SQL style queries. *Fold* is used on these two BitMats to get $BitArr_1$ and $BitArr_2$. *Fold* is analogous to the *projection* operator



:s1 :p1 :o1
:s1 :p1 :o2
:s1 :p2 :o2
:s1 :p2 :o4
:s1 :p3 :o3
:s2 :p1 :o1
:s2 :p2 :o2
:s2 :p2 :o3
:s2 :p3 :o3
:s3 :p1 :o1
:s3 :p1 :o4
:s3 :p1 :o5
:s3 :p3 :o3
:s3 :p3 :o4

Filter operation produces 2 BitMats corresponding to (:s1 ?p ?x) and (:s3 ?p ?y). For the simplicity of the figure they are shown as 2 bit arrays of :s1 and :s3 (as internally bits in other rows are set to 0 as per the *filter* semantics)

Similar to the *filter* operation shown as a bit array above, *unfold* is shown on a bit array for the simplicity of the figure.

Result triples (with BitMat)
:s1 :p1 :o1
:s1 :p1 :o2
:s1 :p3 :o3
:s3 :p1 :o1
:s3 :p1 :o4
:s3 :p1 :o5
:s3 :p3 :o3
:s3 :p3 :o4

Matching subgraphs (as would be produced by a query engine) (variable bindings)

T1.S	T1.P	T1.O	T2.S	T2.P	T2.O
	?p	?x		?p	?y
:s1	:p1	:o1	:s3	:p1	:o1
:s1	:p1	:o1	:s3	:p1	:o4
:s1	:p1	:o1	:s3	:p1	:o5
:s1	:p1	:o2	:s3	:p1	:o1
:s1	:p1	:o2	:s3	:p1	:o4
:s1	:p1	:o2	:s3	:p1	:o5
:s1	:p3	:o3	:s3	:p3	:o3
:s1	:p3	:o3	:s3	:p3	:o4

Fig. 3. Single Join on a BitMat

of SQL queries. If RDF triples are presented in a 3-column table (S, P, O), then these bitarrays correspond to a single column in the table and bit positions set to 1 indicate presence of the S, P, or O values corresponding to those bit positions. Bitwise AND is performed on these bitarrays which is same as a relational join on the column represented by *RetainDimension*. The result of the bitwise AND is *unfolded* back on the filtered BitMats BM_{tp1} and BM_{tp2}. Finally the two BitMats obtained after *unfold* are combined using bitwise OR on the corresponding rows of them. This procedure is depicted in Figure 3.

It can be shown and proved step-by-step that *filter*, *fold*, and *unfold* operators can be mapped to equivalent SQL operations and the correctness of the algorithm can be proved. For the scope of this paper, we have omitted these details, but they can be referred in our technical report [3].

For simplicity of presentation of the algorithm, we have shown it only for a single join with two triple patterns, but the same algorithm can be extended to '*n*' triple patterns joining over a single join variable occurring in the same dimension by performing filter and fold on each triple pattern, ANDing all the bitarrays generated by the fold operation, unfolding the AND results on each of the filtered BitMats, and finally combining all these BitMats with bitwise OR on the corresponding rows to get the result BitMat. The procedure for subject-object cross dimensional join (as shown by an example in Section 3.1) is slightly different and is elaborated in Section 4.1.

4.1 Cross Dimensional Joins

Bitwise AND operation can be performed on two bitarrays only if the corresponding bit positions have the same URI or literal values mapped to them. This is the case for the same dimension joins.

Cross-dimensional joins need special handling. (*?s* :p1 ?x . ?y :p2 *?s*) is an example of subject-object (S-O) cross-dimensional join, for which we need to perform bitwise AND on the subject and object bitarrays. As elaborated in

Section 3.1 and Section 3.2, every bit position in the folded bitarray corresponds to a unique identifier assigned to a URI or literal in the respective subject, predicate, object ID space. Since URIs which appear as subjects as well as objects are allocated the same IDs sequentially from 1 to $|V_{so}|$, for a S-O join, bitwise AND is performed only on the first $|V_{so}|$ bit positions of the respective subject and object bitarrays and rest all bits are cleared.

As mentioned earlier in Section 3.1, other cross-dimensional joins (S-P and P-O) are not that common in the Semantic Web instance (assertional) data and hence are not supported at present.

5 Multi Join Processing

In a multi-join two or more triple patterns join over two or more join variables, e.g. (:s1 *?p ?o* . :s2 *?p* ?y . ?z :p3 *?o*). In a conventional relational join processing, multi-join evaluation can be presented as an operator tree where each internal node is a self-contained representation of the materialized results of the join subtree below it. BitMat single join procedure, as elaborated in Section 4, does not materialize the query results (i.e. does not produce matching subgraphs), but represents the candidate result triples with the result BitMat. Thus, if we simply extend the single-join BitMat algorithm to multi-joins, evaluation of a later join can change the variable bindings produced by an earlier join. Specifically, if the dependency between different join variables is not captured and resolved then a BitMat join can produce *false positives*. Hence is the need for a different algorithm for multi-join queries.

5.1 BitMat Multi-Join Algorithm

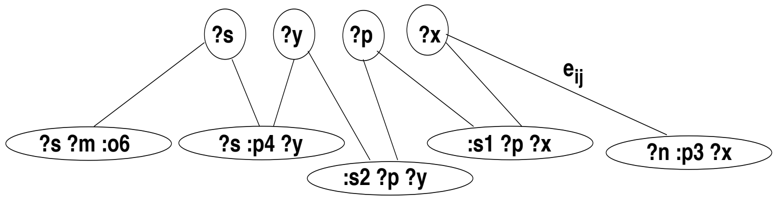


Fig. 4. Multi-join graph $\mathcal{G}$

For the present considerations, we do not handle joins with Cartesian products. For better understanding of the algorithm, we develop the theory by constructing a bipartite graph $\mathcal{G}$ which captures the conjunctive triple pattern and join variable dependencies (see Figure 4).

- Each join variable in the multi-join is a node (denoted as *jvar-node*).
- Each triple pattern is a node (denoted as *tp-node*).

– There is an edge between a jvar-node and a tp-node if the join variable represented by the jvar-node appears in the triple pattern represented by the tp-node.

Algorithm 2 BitMat_MultiJoin(BM, $\mathcal{G}$) returns BitMat

```
/* $BM$ is the BitMat of the original triple-set */
/* Initialize graph $\mathcal{G}$ */
for all $v_k$ in graph $\mathcal{G}$
    if $v_k$ is jvar-node then
        Set $BitArr_k$ to a bitarray with all bits set to 1.
    else
        $BitMat_k$ = filter($BM$, getTriplePattern($v_k$))
    end if
end for
repeat
    Set $changed = false$
    for each $v_i$ as jvar-node in $\mathcal{G}$ do
        Let $PrevBitArr_i = BitArr_i$
        /* $TP$ is a set of all triple patterns
        * having join-var $v_i$ */
        Let $TP = \{v_j \mid \exists e_{ij}\}$
        for each $v_j$ in $TP$ do
            Let $dim$ = getDimension ($v_i$, getTriplePattern($v_j$))
            Let $TempBitArr$ = fold($BitMat_j$, $dim$)
            $BitArr_i$ = ($BitArr_i$) AND ($TempBitArr$)
        end for
        if $PrevBitArr_i$ not equal $BitArr_i$ then
            Set $changed = true$
        end if
        /* Now unfold the result */
        for each $v_j$ in $TP$ do
            Let $dim$ = getDimension ($v_i$, getTriplePattern($v_j$))
            $BitMat_j$ = unfold($BitMat_j$, $BitArr_i$, $dim$)
        end for
    end for
until ($changed == true$ and there are more than one join var)
Let $B_{res}$ be an empty BitMat
for each $v_j$ as tp-node in $\mathcal{G}$ do
    $B_{res} = B_{res}$ OR $BitMat_j$
end for
```

Although for graph $\mathcal{G}$ shown in Figure 4 there are exactly two edges per jvar-node corresponding to a join variable, one could have more than two edges per join variable.

The algorithm to evaluate a multi-join using graph $\mathcal{G}$ is given in Algorithm 2. For simplicity, we assume the existence of certain methods without describing them in the algorithm, viz. method *getTriplePattern(v_j)* returns the triple pattern associated with the tp-node node v_j, and *getDimension(v_i, tp)* returns the position (dimension) of the join variable v_i in triple pattern *tp*, e.g. *getDimension(?s, (?s :p1 ?x))* returns *subject*, *getDimension(?s, (?y :p2 ?s))* returns *object*, and *getDimension(?s, (?s :p2 ?s))* returns *subject* and *object* (this is a special form of S-O cross dimensional join and is captured by the implementation of the BitMat multi-join algorithm, but not shown in Algorithm 2 for simplicity).

Associated with each jvar-node is a bitarray of the most recent result of a join evaluated over that join variable. Initially all the bits in these bitarrays are set to 1 (as explained in the Algorithm 2). Each tp-node has a BitMat associated with it, which is initially set to the result of the *filter(BM, getTriplePattern(v_j))* where v_j is the tp-node. The *repeat-until* loop (between lines 10 and 31) in Algorithm 2 iterates over all the join variables in the query, processing those joins until none of the $BitArr_i$ change. For each join variable, it *folds* the BitMats associated with all the triple patterns which have that join variable (lines 16, 19) and performs a bitwise AND on the generated *BitArrays* (line 20). At the end of the loop (17-21) the final AND result ($BitArr_i$) is *unfolded* back on all the BitMats in the set TP (line 28). Lastly, after the *repeat-until* loop ends, result BitMat is generated by a bitwise OR of all the BitMats associated with all the triple patterns (line 34) in the query.

Although the multi-join algorithm is constructed as a continuous loop, it can be proved that this loop will converge in a finite number of iterations. In each iteration of the loop, we are performing a bitwise AND on the previous $BitArr_i$ and the new $TempBitArr$ folded from the BitMat of the triple pattern having that join variable. After each bitwise AND, the resulting $BitArr_i$ is unfolded on the BitMats associated with all the triple patterns having that join variable. Since we are doing a bitwise AND operation and *unfold* which expands the $BitArr_i$ on the BitMats, only a bit set to 1 can be flipped to 0. Hence the number of set bits (and hence the triples in the BitMats) reduce monotonically per iteration of the loop and the loop ends at a point when none of the $BitArr_i$ change after an AND (lines 20, 22) (in the worst case when all the bits in all the BitArrays are set to 0, in which case the final join result is *null*).

We provide experimental results in Section 6 for the typical number of iterations taken by the loop. It can be seen that for each join variable, we employ the same basic operations as used for a single join operation.

6 Experiments

This section describes our experiments and evaluation of the BitMat structure and join queries.

6.1 Programming Environment

The BitMat structure and join algorithms have been developed as a C program meant to be run on a Linux distribution. All the experiments were carried out on Gentoo Linux distribution on a Dual Core AMD Opteron Processor 870 with 8GB of RAM. The BitMat program was run as a normal user process with the default priority as set by the Linux system. *Gcc ver. 4.1.1* compiler is used to compile the code with compiler optimization flag set to *-O6*. The RDF N-triple file is first preprocessed using a Perl script to generate a raw RDF triple file by encoding all the triples using the sequence based identifiers allocated to URIs and literals (refer to Section 3.1). Simple bash `sort` command is used to sort these encoded triples on subject-ID, predicate-ID, object-ID. This preprocessing is needed to be carried out only once per dataset, and the time taken by it varies linearly with respect to the size of the triple-set (e.g. it takes around 30 minutes for Wikipedia 47 million triple-set). BitMat's load function expects either a raw RDF triple file or a disk image of the previously generated BitMat. Given a conjunctive triple pattern (join query), another small script is used to transform the conjunctive triple pattern by encoding all the fixed URIs and literals present in the query using the corresponding identifiers, so that the BitMat join processing can operate on the ID-based values.

6.2 Loading a BitMat

We used different RDF triple sets of varying sizes for testing BitMat structure's memory utilization. UniProt-0.2million and UniProt-22million triple sets were extracted from a larger UniProt dataset [22] ($\sim$730 million triples). Uniprot-0.2million and 22 million datasets were selected from first 50 million triples in the Uniprot 730million triple file. LUBM 1 million and 6 million triple sets were generated using LUBM's [11] RDF data generator program. LUBM 1 million was generated by generating data of 10 universities and LUBM 6 million was generated by generating data of 50 universities. Wikipedia 47 million [24] was used as is available on the web without any modifications in it. The characteristics of these datasets are given in Table 1.

Table 1. Dataset characteristics

Dataset	#Triples	#Subjects	#Predicates	#Objects
Uniprot 0.2million	199,912	30,007	55	45,754
LUBM 1million	1,272,953	207,615	18	155,837
LUBM 6million	6,656,560	1,083,817	18	806,980
Uniprot 22million	22,619,826	5,328,843	91	4,516,903
Wiki 47million	47,054,407	2,162,189	9	8,268,864

Table 2 lists size of the BitMats of respective datasets plus the size of forward and reverse mapping dictionaries to map each literal or URI to an integer ID. A compressed BitMat and these dictionary mappings represent the original RDF data completely. The results given in Table 2 show that this representation is much smaller than the original raw RDF data presented in N-triples format.

The small size of the compressed BitMat is due to two reasons: i) since the actual RDF triple set covers only a small set of the total $V_s \times V_p \times V_o$ space (refer to Section 3.1), BitMat makes a very sparse structure, ii) D-gap compression scheme achieves superior results on sparse bit-vectors.

The raw RDF triple file read by the BitMat load procedure is sorted on (subject, predicate, object) IDs. Hence although conceptually we use the D-gap compression scheme, internally our algorithm exploits the sorted triple list to build a compressed BitMat directly instead of building uncompressed bitarrays

Table 2. BitMat Size and Load Time

Dataset (#triples in millions)	Size of compressed BitMat + mapping dictionary / Size of raw RDF data (MB)	Time to load (sec) from Raw file / from Disk Image
UniProt (0.2)	1.5 + 6.5 / 23	0.34 / 0.04
LUBM (1)	11.6 + 47 / 222	1.54 / 0.36
LUBM (6)	60.8 + 248 / 1193	8.35 / 1.95
UniProt (22)	213.5 + 1367 / 4037	17.11 / 8.4
Wikipedia (47)	371.1 + 932 / 7054	34.4 / 4.5

and then compressing them. This results into smaller load times to construct a BitMat from a raw RDF file. The memory-image of a compressed BitMat can be written out to the disk as a binary file. Loading from a disk-image just reads this binary file into a BitMat structure in memory, hence loading from a disk image takes even lesser time than loading the BitMat from sorted triple-ID file.

Note that each conjunctive triple pattern query is converted into an internal representation where all the fixed values in the query are replaced by their corresponding integer mapping IDs (refer Section 3.1). Although the mapping dictionary consumes larger space than the primary BitMat structure, it does not need to be kept in memory while processing the queries.

6.3 Join Query Performance

To test our implementation of the join algorithm, we executed a list of single join queries on a smaller dataset (UniProt 0.2 million) and also measured the response times (for the list of queries, see Table 3). Typically, the subject join query times varied from 0.019sec to 0.04sec, for predicate joins the variation was from 0.0041sec to 0.062sec, for object dimension join it was from 0.0094sec to 0.128sec, and for S-O cross dimensional joins it was from 0.08sec to 0.28sec. Variation in the time depended on different factors such as the selectivity[4] of the triple pattern and join condition, number of total variables in the query, dimension of the variables in the query, etc. as explained further below. For multi-joins, we used a mix of queries taken from UniProt queries [17], LUBM queries available on OpenRDF [15], and some constructed by us for the Wikipedia dataset. Table 4 lists some of these queries (due to space limitations we cannot enlist all the queries). We noted several parameters that characterize these queries as given in the columns of Table 4.

Memory requirements: The "Sum of BitMat sizes" is the maximum memory size of all the BitMats associated with the triple patterns including the result BitMat at any point during the query execution. Note that BitMat size for a single pattern is the size obtained after applying the *filter*, which usually is much smaller than the original BitMat since a majority of the triple patterns have a fixed value in at least one of the S, P, O positions. But we have successfully tried queries having all variable positions in one or more triple patterns as well (e.g.

[4] A lower selective triple pattern has more triples associated with it and vice versa.

Table 3. Single Join Queries on UniProt-0.2million

Query	#Result Triples	Time (sec)
Subject Joins		
(?s :author ?x)(?s rdf:type ?y)	31,044	0.019
(?s ?p :taxonomy:5875)(?s rdf:type ?y)	2	0.04
(?s ?p ?o)(?s rdf:type ?y)	199,912	0.042
(?s ?p ?o)(?s :author ?y)	43,408	0.02
Predicate Joins		
(?x ?p :P15711) (?y ?p :Q43495)	10	0.062
(:UniProt.rdf#_F ?p ?o) (?y ?p :Q43495)	6	0.034
(:UniProt.rdf#_A ?p ?x) (:UniProt.rdf#_F ?p ?y)	10	0.0041
(?s ?p ?x) (:UniProt.rdf#_F ?p ?y)	75,572	0.062
Object Joins		
(?x :created ?o)(?y :modified ?o)	2056	0.016
(:P15711 ?p ?o)(?y :modified ?o)	33	0.010
(?x ?p ?o)(?y :modified ?o)	2056	0.128
(?x :created ?o)(:P28335 :modified ?o)	19	0.0094
S-O Joins		
(?o :begin ?x)(?y :range ?o)	11,830	0.28
(?x ?p ?o)(?o rdf:type ?y)	51,232	0.08
(?x ?n ?o)(?o ?m ?y)	135,325	0.081

a Wikipedia query in Table 4). The sizes of these BitMats are highest at the beginning of the query, and they go on reducing monotonically as the multi-join algorithm executes. This is due to the fact that *filter*, *fold*, and *unfold* operate on a compressed BitMat, and in every iteration of the multi-join algorithm, triples get eliminated monotonically, hence the BitMat size shrinks. Also we do not materialize the intermediate join results, but represent them as candidate triples in the result BitMat. The size variation also depended on the selectivity of the triple patterns. The higher the selectivity, the lower the BitMat size (due to D-gap compression).

The variation of the query execution times can be attributed to three key factors: i) join-dimension (e.g. whether it is a subject, predicate, object, or S-O cross dimension join), ii) selectivity of the triple patterns, and iii) the order of the join evaluation.

Join dimension: Since we are using a subject BitMat for joins on all the dimensions, subject-dimension joins inherently benefited, as folding and unfolding of a compressed BitMat by retaining the subject dimension involves only updating the relevant subject rows without accessing the compressed content. Since the number of distinct predicates is usually low in the datasets, predicate joins performed well too. However, accessing object dimension in a compressed subject BitMat needed special handling, and hence we observe that the structure of the subject BitMat is unsuited for the object joins since *unfold* requires accessing every O-bit position within each subject row, and for each predicate in

Table 4. Multi-join Queries

Query	Dataset (million triples)	Sum Bit-Mat sizes	#Resulting triples	Time (sec) / #multi-join loop-iterations	#Join var / #all vars / #triple patterns
(?protein rdf:type :Protein) (?protein :annotation ?annotation) (?annotation rdf:type :Transmembrane_Annotation) (?annotation :range ?range) (?range :begin ?begin) (?range :end ?end)	UniProt (0.2)	355KB	3712	0.066 / 3	3 / 5 / 6
(?p1 rdf:type :Protein)(?p1 :enzyme :enzymes:2.7.1.105) (?p2 rdf:type :Protein) (?p2 :enzyme :enzymes:3.1.3.-) (?interaction rdf:type rdf:Statement) (?interaction rdf:subject ?p1) (?interaction rdf:subject ?p2)	UniProt (0.2)	434KB	0	0.068 / 3	3 / 3 / 7
(?X rdf:type ub:GraduateStudent) (?Y rdf:type ub:University) (?Z rdf:type ub:Department) (?X ub:memberOf ?Z) (?Z ub:subOrganizationOf ?Y) (?X ub:undergraduateDegreeFrom ?Y)	LUBM (1)	2.1MB	994	0.39 / 3	3 / 3 / 6
	LUBM (6)	10.4MB	20,808	3.24 / 3	3 / 3 / 6
(?X rdf:type ub:UndergraduateStudent) (?Y rdf:type ub:FullProfessor) (?Z rdf:type ub:Course) (?X ub:advisor ?Y) (?Y ub:teacherOf ?Z) (?X ub:takesCourse ?Z)	LUBM (1)	4.6MB	10,113	2.17 / 15	3 / 3 / 6
	LUBM (6)	23.1MB	52,029	55.53 / 18	3 / 3 / 6
(?s :title "Dilbert_Bit_Characters") (?s ?p ?x)(?s2 ?p ?y) (?s2 rdf:type wiki:Article) (?s2 ?n ?z) (?s2 wiki:internalLink ?m)	Wikipedia (47)	1.9GB	1	5.04 / 2	3 / 9 / 6
(?s :title "Dilbert_Bit_Characters") (?s ?p :Bully)(:Johnny_the_Homicidal_Maniac ?p ?o) (?o rdf:type wiki:Article)	Wikipedia (47)	26.5MB	128	3.34 / 2	3 / 3 / 4

turn. Also for the queries with a triple pattern having variable predicate dimension and a fixed object dimension, e.g. *(?s ?p :o1)*, *fold* operation will require to access a single bit position within all the subject rows, for all the predicates. This effect was observed specifically on very large datasets when the selectivity of the triple pattern was low. This further brought our attention to the aspect of using all or some of the six possible BitMats that can be flattened from a 3D bit-cube, as we explained in Section 3. Usage of different BitMat structures requires changes in the present multi-join algorithm, as *fold* and *unfold* operations have to interoperate between multiple types of BitMats. We are currently exploring this aspect.

Selectivity: Initial selectivity of the triple pattern as well as the selectivity of join played a role in the faster convergence of the multi-join loop. Selectivity of the triple pattern or join result also plays key role in the memory utilization. Higher selectivity of the triple patterns and join results reduces the BitMats' size faster.

Join order: Although in our experiments the multi-join algorithm's loop typically converged in 3 or 4 iterations, as it can be noted from Table 4, the second LUBM query took many more iterations (15-18) for 1 million as well as 6 million dataset. As join ordering affects the query execution time and size of the intermediate results in a relational scheme, it affects the number of iterations of the BitMat multi-join algorithm as well. In case of a BitMat join, due to the use of compressed BitMats and the fact that we are not materializing the intermediate results, memory utilization was not affected though. For the second LUBM query, we observe that evaluating join over *?Y* first brought down the multi-join algorithm iterations from 15 to 12. We plan to use an augmented version of the multi-join bipartite graph $\mathcal{G}$ (not shown in Figure 4) that can be used to capture the cyclic or acyclic dependency, so that we can minimize the number of iterations needed to complete the multi-join processing.

7 Conclusions and Future Work

As shown by our experiments, one of the main advantages of the BitMat structure and joins is that the memory requirement of the system is very low. Since the intermediate or final results in a multi-join are not completely materialized, the result size is always bounded by the size of the original BitMat. If the size of the original BitMat is $Size_{BM}$ and n is the number of triple patterns in a multi-join query, then the instantaneous memory utilization while performing a join is always bounded by $O(n * Size_{BM})$ and the final join result size is always bounded by $O(Size_{BM})$.

We are presently developing an efficient algorithm to enumerate all the matching subgraphs from a result BitMat (i.e. final variable bindings). With this the BitMat main-memory triple store can be used as an independent SPARQL join query engine. Also as pointed out in the experimental section, we are exploring the usage of BitMats created by flattening the 3D bit-cube on different dimensions for further improving the performance of the multi-join queries. Development of this algorithm for a fair comparison of query performance and memory utilization with any other state-of-the-art RDF triplestore, is a part of our ongoing work.

The present BitMat query processing algorithm only performs the step of pruning the candidate RDF triples but does not produce final *matching subgraphs*. Hence in this paper, we have just presented empirical results of query performance and memory utilization.

The key aspect of BitMat's algorithms is that they always work on compressed data without uncompressing it at any point.

Due to the ability to create a disk image of a BitMat in memory (as explained in Section 3.1), the BitMat system can be used as a persistent data-store as well, by exploiting the benefits of performing all the operations in main-memory once the BitMat is reloaded from the disk image.

Finally, we conjecture that by creating clusters of machines, each deploying BitMats, extremely large RDF stores can be created and processed in memory. Our future work includes exploring how this can be realized on server-farm like clusters as well as shared memory supercomputers for very large RDF datasets.

Acknowledgements

We would like to thank Dr. Jagannathan Srinivasan of Oracle Corp. for his invaluable inputs during the early development of this work.

References

1. TDB - A SPARQL Database for Jena. http://jena.sourceforge.net/TDB/.
2. ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/.
3. M. Atre, J. Srinivasan, and J. Hendler. BitMat: A Main Memory RDF store. *Technical Report*, TW-2009-02, January 2009.
4. D. Beckette and B. McBride. RDF/XML Syntax Specification. W3C Recommendation, February 2004. http://www.w3.org/TR/rdf-syntax-grammar/.
5. M. Cai and M. Frank. RDFPeers: A Scalable Distributed RDF Repository based on a Structured Peer-to-Peer Network. In *Proceedings of WWW*, May 2004.
6. R. Cyganiak. A Relational Algebra for SPARQL. *Technical Report, HP Laboratories Bristol*, September 2005.
7. D-gap Compression Scheme. http://bmagic.sourceforge.net/dGap.html.
8. O. Erling. Advances in Virtuoso RDF Triple Storage (Bitmap Indexing), October 2006. http://virtuoso.openlinksw.com/-wiki/main/Main/VOSBitmapIndexing.
9. O. Hartig and R. Heese. The SPARQL Query Graph Model for Query Optimization. In *Proceedings of ESWC*, 2007.
10. M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *Proceedings of ISWC*, 2005.
11. Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.
12. A. Matono, S. M. Pahlevi, and I. Kojima. RDFCube: A P2P-based Three-dimensional Index for Structural Joins on Distributed Triple Stores. In *DBISP2P in Conjunction with VLDB 2006*, September 2006.
13. T. Neumann and G. Weikum. RDF-3X: A RISC-style Engine for RDF. In *VLDB*, 2008.
14. T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD*, 2009.
15. OpenRDF LUBM SPARQL Queries. http://repo.aduna-software.org/viewvc/org.openrdf/?pathrev=6875.
16. E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query/.
17. Queries on UniProt RDF dataset. http://dev.isb-sib.ch/projects/expasy4j-webng/query.html#examples.
18. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. *CoRR*, abs/0806.4627, 2008.
19. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation. In *Proceedings of WWW*, April 2008.
20. SwiftOWLIM Semantic Repository. http://www.ontotext.com/owlim/index.html.
21. O. Udrea, A. Pugliese, and V. Subrahmanian. GRIN: A Graph Based RDF Index. In *AAAI*, 2007.
22. UniProt RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.
23. C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of VLDB*, 2008.
24. Wikipedia RDF Dataset. http://labs.systemone.at/wikipedia3.