# BitMat: A Main Memory RDF Triple Store

## Technical Report

### Medha Atre
Dept. of Computer Science
Rensselaer Polytechnic Inst.
Troy NY, USA

### Jagannathan Srinivasan
Oracle
1 Oracle Drive
Nashua NH, USA

### James A. Hendler
Dept. of Computer Science
Rensselaer Polytechnic Inst.
Troy NY, USA

## 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 Bit-Mat structure, its use in processing join queries, describe our experimental results with RDF datasets of different sizes (from 200k to 47 million), and discuss the use case scenarios.

## 1. INTRODUCTION

RDF [3], a W3C standard for representing information about Web resources, and SPARQL [15], a query language for RDF are gaining importance as semantic data is increasingly becoming available in the RDF format. To meet the storage and querying needs of large scale RDF stores, numerous systems are being developed. These systems can be broadly classified as persistent disk-based and main-memory-based systems. The work described in this report belongs to the latter category proposing a main-memory RDF triple store.

Main-memory based RDF triple stores are becoming more desirable for two reasons: 1) due to the continuous technological advances in hardware, personal computers with much higher memory and computing power are readily available, e.g. a desktop system with 2GB of main-memory (RAM) is commonplace now, and 2) these machines can be used to build peer-to-peer distributed RDF stores thereby allowing handling of arbitrarily large RDF data in memory.

Among these main-memory stores, a majority of them (such as Jena [10], Sesame [18] etc.) are implemented as a straightforward extension of the corresponding relational persistent storage systems. There are a few systems that radically try to exploit the main-memory aspects of the triple stores. These include BRAHMS [9], RDFCube [13],

and SwiftOWLIM [20]. Most of the 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 inverted index structure itself makes up the primary storage for RDF triples and is thus exploratory in nature.

We chose to investigate this inverted structure because a RDF triple (subject predicate object) is a 3-dimensional entity which conceptually gives rise to a single universal table holding all RDF triples, which can be horizontally partitioned into multiple fragments based on the usage requirements (denoted as separate RDF Models). The fact that we have a single table or multiple fragments of a single table, can be exploited by using an inverted structure and to query it in turn. 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 representing it in memory. 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)

**Figure 1: BitMat of sample RDF data**

Thus RDF triple-sets can be represented in memory as BitMats. 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 in achieving the compactness of the internal representation of the BitMat. BitMat is maintained as an array of bit-rows, where each row is a collection of all the triples having the same subject. Each bit-row is compressed using D-gap compression scheme [6].

Presently we support only *conjunctive triple pattern* (join)

queries on a BitMat. These queries are processed using bit-wise AND, OR operations on the 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 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.

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 |

**Figure 2: Result BitMat of a sample query**

Unlike the conventional RDF triple stores, where size of the intermediate join results can grow very large, our BitMat based multi-join[1] 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).

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. Although currently we do not have an algorithm to enumerate matching subgraphs from the result BitMat, our scheme 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 [1]) to identify the result triples from a large triple set.

BitMat is designed to be mainly a *read-only* RDF triple store. Dynamic insertion or deletion of RDF triples is not supported at present. But since our load time for large datasets is small enough, insertion and deletion of triples can be emulated by constructing a new BitMat from a modified triple-set. Conjunctive triple pattern queries with large number of triple patterns and join variables are the most performance intensive queries in SPARQL. Hence 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 report 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 than previous description of the ongoing work [2].

---

[1]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.

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 report 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 [7] to efficiently process queries over low cardinality data. Bitmap indexes can be used for effectively filtering predicates on multiple columns by performing bit-wise AND/OR operations on bitmaps extracted from the individual bitmap indexes. In a similar manner, BitMat performs join operations through bit-manipulation.

Other data structures employed by the main-memory stores include hash tables and indexes (SwiftOWLIM [20] and BRA-HMS [9]). BRAHMS system focuses on *semantic association discovery* (finding paths between two nodes in a RDF graph), for which it uses graph algorithms like depth-first-search and breadth-first-search. It employs six indexes i.e. two per dimension (e.g. subject dimension ordered on (predicate, object) and (object, predicate)) to speed up these queries.

Recently, Hexastore [22] has also proposed the use of six different ways of indexing RDF data, one for each possible ordering of the three RDF elements. Although our work primarily uses subject BitMat, which corresponds to one of the above orders, one could create six BitMats in a similar manner. This aspect is further discussed in Section 3.

With respect to the different index structures, our approach is closer to the space-filling approach used to create a multi-dimensional index in TriStarp Group Triple Store [11]. Here the basic idea is to regard potential values of the key sets as the points in an n-dimensional Cartesian product space. This product space is referred to as *key-space* and the points which correspond to the actual records are referred to as the *datum-points*. The Hilbert space filling curve is used to generate a linear order of the datum-points in the key-space. The datum-points corresponding to the consecutive (broken-up) sections of the curve are stored together in a single page. The first datum-point on the page forms the page key thereby giving pages a logical ordering. This allows creating a conventional B-Tree or some variant index that maps page-keys to the corresponding physical page addresses.

However, BitMat differs from the space filling approach and the other index structures in that it is the primary storage structure as opposed to being an index.

In this regard, the structure that comes closest to BitMat is RDFCube [13], 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 (RDFPeers [4]) 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.

Also, SPARQL query language, which is structurally quite similar to the SQL query language [5], is being studied specifically with respect to the join processing [8, 19] and query benchmarking [17]. In contrast to the conventional SPARQL query processing scheme, we employ an iterative 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 indexes on the RDF data). Each structure aids particular set of single-join queries better. But since in our approach, we handle multi-join queries without materializing the intermediate results completely, it will need special handling to use BitMats formed by flattening the cube on different dimensions. Hence for the present considerations, we have used subject BitMat to perform all the experiments. Usage of different structures of BitMats in a multi-join query is a topic of further exploration. Figure 1 is an example of a subject BitMat.

### 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 cross dimensional join as discussed in Section 4. $V_s \times V_p \times V_o$ gives the conceptual bit-cube from which a BitMat is constructed.

The original RDF triple set is converted to an ID-based triple set using the above ID mappings and the corresponding bit position in the BitMat is set. 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. All the (S,O) matrices are concatenated together horizontally thereby resulting in $|V_s|$ subject rows. Each subject row is compressed using D-gap compression scheme. Although this is the conceptual structure of a Bit-Mat, we build the compressed BitMat directly from the encoded triple set as explained further in Section 6.

### 3.2 BitMat Operations

BitMat is created and manipulated using the following operations:

**Load:** This operation is specified as *'load(filename, filetype) returns BitMat'*. It loads the RDF data from the input filename into memory. Two modes of loading are supported:

- *filetype = 'Raw' mode:* In this mode, the input file is expected to contain a list of RDF triples represented using the sequence based identifiers, and a BitMat is constructed by setting the appropriate bits.

- *filetype = 'Image' mode:* In this mode, the input binary file is a memory image of the previously generated BitMat using *create_disk_image* function. Hence the image file is directly read into a BitMat structure.

**Create_disk_image:** This function, specified as *'create-_disk_image(BitMat, filename)'* writes the given BitMat to a binary file on the disk. This BitMat image can be used for the subsequent load 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*.

Let $\varphi$ be a propositional formula used in the SQL selection operation $\sigma$. Then a triple pattern (TP) *(?s :p1 ?o)* can be expressed as $\sigma_\varphi$ with $\varphi$ as *P=':p1'*. Thus if $B$ represents a set of RDF triples, *filter* operation can be expressed as:

$$\text{filter(B, TP)} = \sigma_\varphi(\text{B})$$

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*, a bound predicate value retains the bits in the corresponding single (S,O) matrix, a bound object retains the bits in the corresponding list of (S,P) columns. A triple pattern with more than one bound values retains a subset of the bits retained in the one bound value case, e.g. (*:s1 :p1 ?o*) retains only the part corresponding to predicate *:p1* in the row corresponding to subject *:s1* and rest all bits are cleared.

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

Let *dim* be the dimension to be retained, then *fold* operation can be expressed using the SQL projection operator $\pi$ and a *Distinct* operator denoted by $\delta$. If B represents a set of RDF triples then the result of a fold operation can be expressed as
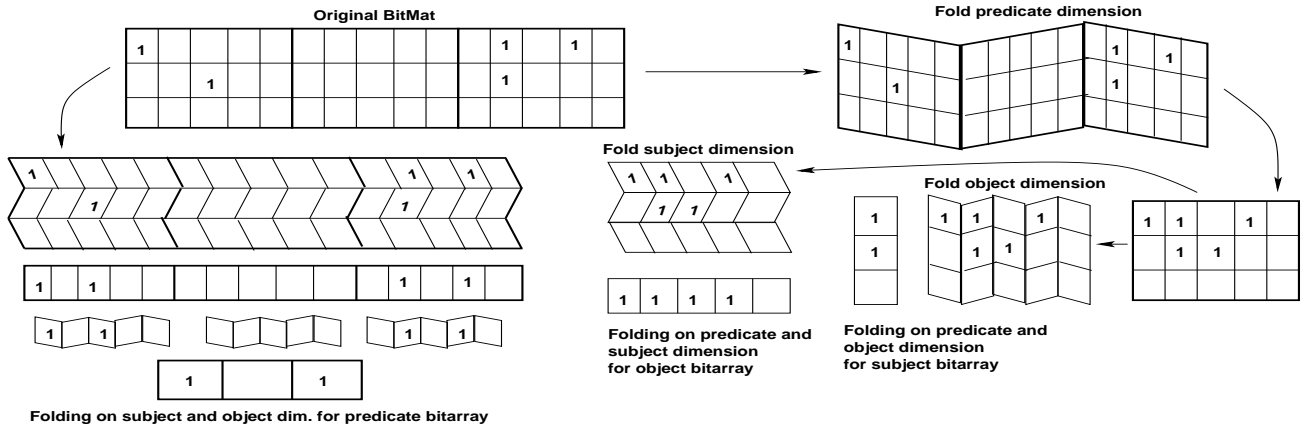
**Figure 3: Folding a BitMat**

$$\text{fold (B, dim)} = \delta(\pi_{dim}(B))$$

For example, if *RetainDimension* is set to *'object'*, then BitMat is folded along the subject and predicate dimensions resulting into a single bitarray (see Figure 3). 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.

Let *'dim'* be the *RetainDimension* as used in *fold*, and M be a single column in a relational table (corresponding to the *RetainDimension*) containing a set of subject, predicate, or object values. These values correspond to the bits set to 1 in the *MaskBitArray*. Then *unfold* operation can be expressed using the SQL join operator ⋈. Specifically, if B represents a set of RDF triples then the result of an *unfold* operation, can be expressed as

$$\text{unfold(B, M, dim)} = M \bowtie_{dim} B$$

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

In this section, we present the basics of a BitMat join showing how it relates to the join processing done in a relational database, and the corresponding join algorithm is given next.

### 4.1 BitMat Join Basics

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 4). 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).

Similarly, a BitMat based join takes the original RDF graph $G$ (represented in the form of a BitMat), and produces another graph $G_{res}$ represented by the result BitMat, which identifies a distinct set of candidate triples among the matching subgraphs. The set of triples in $G_{res}$ is a subset of $G$ such that:

- A triple being absent means it is *not* part of *any* matching subgraph.

- A triple being present means it is part of *at least* one matching subgraph.

Let $T_{triple}$ be a transformation function which takes a matching subgraph with variable bindings and produces a set of distinct triples which make that matching subgraph. Let $G_{ms_i}$ be a matching subgraph that would be produced by a conventional SPARQL query engine on a given RDF graph $G$, let there be $n$ such matching subgraphs, then:

$$G_{res} = \bigcup_{i=1}^{n} T_{triple}(G_{ms_i})$$

The transformation function $T_{triple}$ is the combined effect of three BitMat join operations (filter, fold, unfold) each of which can be mapped to the operators in relational algebra. For simplicity, assume that the given set of RDF triples in a graph $G$ are stored as a single relational table $T$ with three columns S, P, O corresponding to subject, predicate, and object respectively. Let the two join triple patterns be $TP_1$ and $TP_2$. Let $\varphi_1$ and $\varphi_2$ be the propositional formulae used in the SQL selection operator such that if $TP_1$ is *(?s :p1 ?x)* then $\varphi_1$ is *(P=:p1)*. Let *'dim'* be the column over which the join is performed, then a relational SPARQL join is expressed as

$$T_{join} = \sigma_{\varphi_1}(T) \bowtie_{dim} \sigma_{\varphi_2}(T)$$

Let $T_1 = \sigma_{\varphi_1}(T)$ and $T_2 = \sigma_{\varphi_2}(T)$, then the transformation $T_{triple}$ can be defined as

$$T_{triple} = \pi_{a_1}(T_{join}) \cup \pi_{a_2}(T_{join})$$

where $a_1$ and $a_2$ are the projection operators such that $a_1$ is $(T_1.S, T_1.P, T_1.O)$ and $a_2$ is $(T_2.S, T_2.P, T_2.O)$.

## 4.2 BitMat Join Algorithm

The algorithm to obtain $G_{res}$ for a single join pattern using a BitMat is given in Algorithm 1:

---
**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: $BM_{tp1} = \text{filter}(BM, tp_1)$
5: $BM_{tp2} = \text{filter}(BM, tp_2)$
6: $BitArr_1 = \text{fold}(BM_{tp1}, RetainDimension_{tp1})$
7: $BitArr_2 = \text{fold}(BM_{tp2}, RetainDimension_{tp2})$
8: $BitArr_{res} = BitArr_1 \text{ AND } BitArr_2$
9: $BM_{tp1} = \text{unfold}(BM_{tp1}, BitArr_{res}, RetainDimension_{tp1})$
10: $BM_{tp2} = \text{unfold}(BM_{tp2}, BitArr_{res}, RetainDimension_{tp2})$
11: /* Produce the final result BitMat */
12: Let $B_{res}$ be an empty BitMat
13: $B_{res} = B_{tp1} \text{OR } BM_{tp2}$

---

On lines 4, 5 *filter* operation is used to get two BitMats containing only triples satisfying the first and second triple pattern respectively. *Fold* is used on these two BitMats to get $BitArr_1$ and $BitArr_2$. 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 4.

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 4.3.

Algorithm 1 produces the same set of triples as would be produced by applying the $T_{triple}$ transformation on the resulting matching subgraphs obtained from a conventional SPARQL engine. A formal proof of this equivalence is given using relational algebra operators in Appendix A.

## 4.3 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.

Other cross-dimensional joins (S-P and P-O) are not that common in the Semantic Web instance data. Supporting such joins would involve an extra step of mapping the bit position from one dimension to the appropriate bit position of the other dimension prior to performing bitwise AND. Presently, we support only S-O cross dimension joins, and supporting S-P, P-O cross-dimensional joins is a part of future work.

## 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 the need for a different algorithm for multi-join queries.

Generation of the *false positives* can be elaborated with an example join (:s1 *?p ?o* . :s2 *?p* ?y . ?z :p3 *?o*) over the BitMat given in Figure 5 in three steps.

1. Evaluation of the predicate join (:s1 *?p* ?o . :s2 *?p* ?y), involves unfolding the bitwise AND of bitarrays $BA_1$ and $BA_2$ (join result) on the BitMats returned by *filter(BM, :s1 ?p ?o)* and *filter(BM, :s2 ?p ?y)* (not shown in the figure to avoid the clutter), resulting in BitMats $BM_1$ and $BM_2$.

2. Evaluation of the object join (:s1 ?p *?o* . ?z :p3 *?o*), involves folding $BM_1$ and the BitMat returned by *filter(BM, ?z :p3 ?o)* to get object bitarrays $BA_3$ and $BA_4$. Unfolding the bitwise AND of $BA_3$ and $BA_4$ gives $BM_1'$ and $BM_3$.

3. ORing $BM_1'$, $BM_2$, and $BM_3$ gives resulting BitMat $BM_{res}$.

$BM_{res}$ contains *false positive* triples as highlighted with circles in Figure 5. Hence the correct operation is to re-evaluate the predicate join, since the variable bindings produced by the predicate join change after the object join. We propose a scalable algorithm to resolve this dependency in a multi-join and evaluate multi-join queries without completely materializing the join results at any stage.

## 5.1 BitMat Multi-Join Algorithm

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 6).
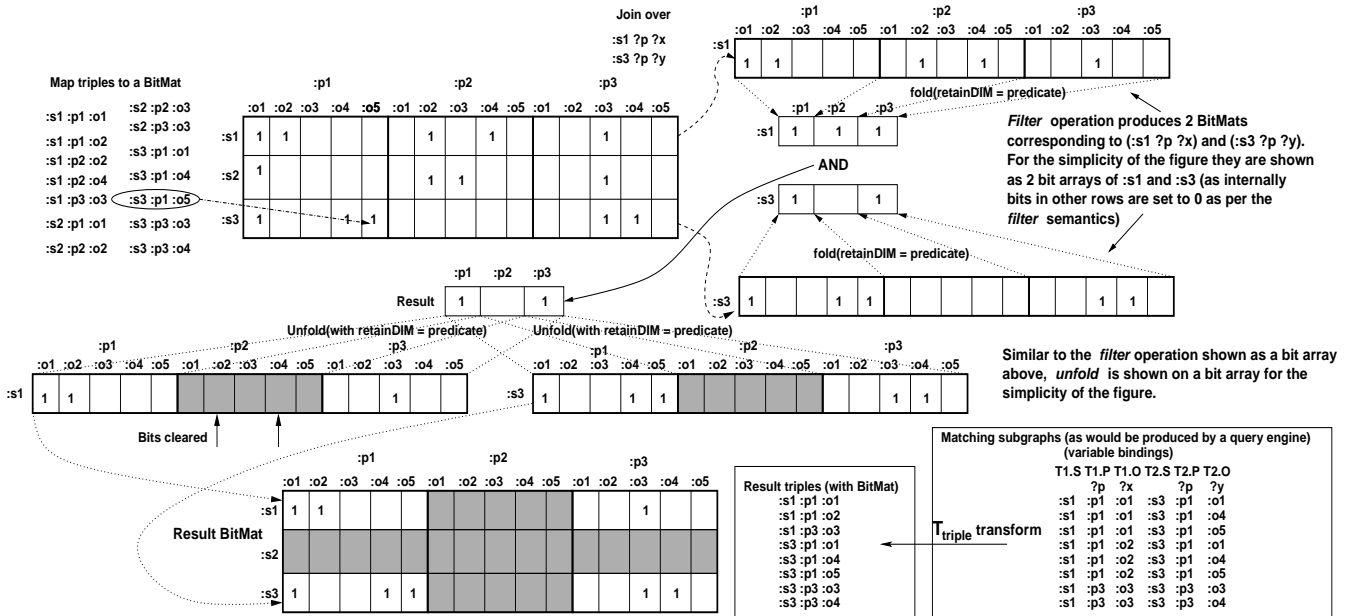
**Figure 4: Single Join on a BitMat**

- Each join variable in the multi-join is a node (denoted as *var-node*).

- Each triple pattern is a node (denoted as *tp-node*).

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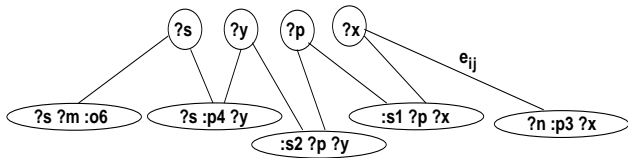


**Figure 6: Multi-join graph $\mathcal{G}$**

Although for graph $\mathcal{G}$ shown in Figure 6 there are exactly two edges per var-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 var-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.

---

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

1: /* BM is the BitMat of the original triple-set */
2: /* Initialize graph $\mathcal{G}$ */
3: **for all** $v_k$ in graph $\mathcal{G}$
4:    **if** $v_k$ is var-node **then**
5:       Set $BitArr_k$ to a bitarray with all bits set to 1.
6:    **else**
7:       $BitMat_k$ = filter(BM, getTriplePattern($v_k$))
8:    **end if**
9: **end for**
10: **repeat**
11:    Set $changed = false$
12:    **for each** $v_i$ as var-node in $\mathcal{G}$ **do**
13:       Let $PrevBitArr_i = BitArr_i$
14:       /* TP is a set of all triple patterns
15:       * having join-var $v_i$ */
16:       Let $TP = \{v_j \mid \exists e_{ij}\}$
17:       **for each** $v_j$ in $TP$ **do**
18:          Let $dim$ = getDimension ($v_i$, getTriplePattern($v_j$))
19:          Let $TempBitArr$ = fold($BitMat_j$, $dim$)
20:          $BitArr_i = (BitArr_i)$ AND $(TempBitArr)$
21:       **end for**
22:       **if** $PrevBitArr_i$ not equal $BitArr_i$ **then**
23:          Set $changed = true$
24:       **end if**
25:       /* Now unfold the result */
26:       **for each** $v_j$ in $TP$ **do**
27:          Let $dim$ = getDimension ($v_i$, getTriplePattern($v_j$))
28:          $BitMat_j$ = unfold($BitMat_j$, $BitArr_i$, $dim$)
29:       **end for**
30:    **end for**
31: **until** ($changed == true$ and there are more than one join var)
32: Let $B_{res}$ be an empty BitMat
33: **for each** $v_j$ as tp-node in $\mathcal{G}$ **do**
34:    $B_{res} = B_{res}$ OR $BitMat_j$
35: **end for**

---

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,
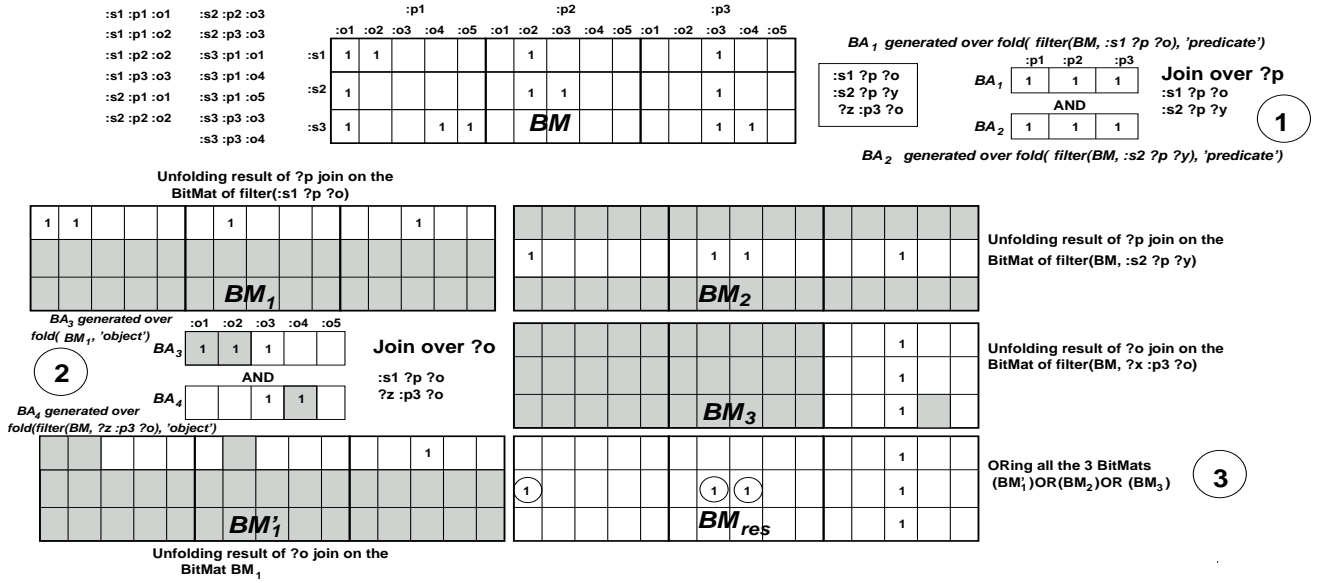
**Figure 5: False Positives in a Multi-join on a BitMat**

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. The correctness of a single join is proved in Section 4 and Appendix A. A formal proof of the correctness of the multi-join algorithm as an extension of the single join algorithm is beyond the scope of this report.

# 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. The program is mainly run on Gentoo Linux distribution on a Dual Core AMD Opteron Processor 870 with 4GB 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). Bit-Mat'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. Table 1 lists our results. UniProt-200k and UniProt-22million triple sets were extracted from a larger UniProt [21] dataset ($\approx$730 million triples). LUBM [12] 1 million and 6 million triple sets were generated using LUBM's RDF data generator program. Wikipedia [23] 47 million was used as is available on the web without any modifications in it.

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 *load* function (refer to Section 3.2) of the BitMat program is sorted on (subject, predicate, object) IDs. Hence although conceptually

**Table 1: BitMat Size and Load Time**

| Dataset (#triples in millions) | Compressed BitMat size | Time to load (sec) from Raw file / from Disk Image |
|---|---|---|
| UniProt (0.2) | 1.5MB | 0.34 / 0.04 |
| LUBM (1) | 11.6MB | 1.54 / 0.36 |
| LUBM (6) | 60.8 MB | 8.35 / 1.95 |
| UniProt (22) | 213.5 MB | 17.11 / 8.4 |
| Wikipedia (47) | 371.1 MB | 34.4 / 4.5 |

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 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.

## 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 200k) and also measured the response times (for the list of queries, see Appendix B). 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[2] 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 [16], LUBM queries available on OpenRDF [14], and some constructed by us for the Wikipedia dataset. Table 2 lists some of these queries. We noted several parameters that characterize these queries as given in the columns of Table 2.

**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 contain a bound variable. 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 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.

**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.

**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 2, 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 much. 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 6) 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 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 $O(Size_{BM})$.

Although currently the BitMat query processing interface does not produce *matching subgraphs* as done by a standard SPARQL query engine, BitMat structure and join processing can be used as a *"plugin"* to any other triple store

---

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

Table 2: 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) (?s3 wiki:internalLink ?m) | Wikipedia (47) | 1.1GB | 46,129,999 | 47.9 / 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) | 549.2MB | 840,582 | 34.4 / 2 | 3 / 3 / 4 |

(persistent or main-memory) where updates on the triple store are less frequent and join query optimization is of more importance. If we denote $T_{BM}$ as the time taken by the BitMat join algorithm to produce the join result BitMat, $T_{SoBM}$ as the time taken by a SPARQL query engine to produce matching subgraphs by using the BitMat results, and $T_{SPARQL}$ as the time taken by an in-memory SPARQL query engine to produce the results without BitMat interception, then BitMat will be useful for a class of queries for which $T_{BM} + T_{SoBM} \ll T_{SPARQL}$.

In a standalone mode, BitMats can be used to perform 'EXISTS' or 'ASK' queries involving large multi-joins.

As a part of the future work, we are working on developing a procedure to enumerate all the matching subgraphs from a result BitMat whereby 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 will explore the usage of BitMats created by flattening the 3D bit-cube on different dimensions for further improving the performance of the multi-join queries.

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.

Our experiments have shown that our join query performance is no worse (in most cases better) than the existing main-memory or persistent RDF triple stores, but we avoid specifically comparing BitMat query evaluation performance with other RDF triple stores at present, because the structure and purpose of the BitMat is very specific, hence unless there exists a system deploying similar structures and query interface, any comparison with other systems will be misleading.

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.

## 8. REFERENCES

[1] ARQ - A SPARQL Processor for Jena. http://jena.sourceforge.net/ARQ/.
[2] M. Atre, J. Srinivasan, and J. Hendler. BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries. In *ISWC, Poster and Demo track*, Karlsruhe, Germany, October 2008.
[3] D. Beckette and B. McBride. RDF/XML Syntax Specification. W3C Recommendation, February 2004. http://www.w3.org/TR/rdf-syntax-grammar/.
[4] 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.
[5] R. Cyganiak. A Relational Algebra for SPARQL. *Technical Report, HP Laboratories Bristol*, September 2005.
[6] D-gap Compression Scheme. http://bmagic.sourceforge.net/dGap.html.
[7] O. Erling. Advances in Virtuoso RDF Triple Storage (Bitmap Indexing), October 2006. http://virtuoso.openlinksw.com/-wiki/main/Main/VOSBitmapIndexing.
[8] O. Hartig and R. Heese. The SPARQL Query Graph Model for Query Optimization. In *Proceedings of ESWC*, 2007.
[9] M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *Proceedings of ISWC*, 2005.
[10] Jena - A Semantic Web Framework for Java. http://jena.sourceforge.net/.
[11] J. K. Lawder and P. J. H. King. Querying Multi-dimensional Data Indexed Using the Hilbert Space-Filling Curve. *SIGMOD Record*, 30(1):19–24, 2001.
[12] Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.
[13] 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.

[14] OpenRDF LUBM SPARQL Queries. http://repo.aduna-software.org/viewvc/org.openrdf/?pathrev=6875.

[15] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Recommendation, January 2008. http://www.w3.org/TR/rdf-sparql-query/.

[16] Queries on UniProt RDF dataset. http://dev.isb-sib.ch/projects/expasy4j-webng/query.html#examples.

[17] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP2Bench: A SPARQL Performance Benchmark. *CoRR*, abs/0806.4627, 2008.

[18] Sesame - RDF Schema Querying and Storage. http://www.openrdf.org/.

[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] UniProt RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.

[22] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *Proceedings of VLDB*, 2008.

[23] Wikipedia RDF Dataset. http://labs.systemone.at/wikipedia3.

# APPENDIX

## A. SINGLE JOIN PROOF

**Definitions:** $T$ is a relational table with three columns S, P, O corresponding to subject, predicate, object. Two join triple patterns are $TP_1$ and $TP_2$. $\varphi_1$ and $\varphi_2$ are the propositional formulae such that if $TP_1$ is *(?s :p1 ?x)* then $\varphi_1$ is *(P=:p1)*. $T_1$ and $T_2$ are the auxiliary tables such that

$$T_1 = \sigma_{\varphi_1}(T) \qquad T_2 = \sigma_{\varphi_2}(T) \qquad (1)$$

$$T_{join} = T_1 \bowtie_{dim} T_2 \qquad (2)$$

where *'dim'* is the column (either S, P, or O) over which the join is performed. $a_1$ is $(T_1.S, T_1.P, T_1.O)$ and $a_2$ is $(T_2.S, T_2.P, T_2.O)$.

**Proof:** Each row in the join table $T_{join}$ is a *matching subgraph*. Let $\delta$ denote the *Distinct* operator. To get the distinct triples from $T_{join}$, let's apply transformation $T_{triple}$ as follows (refer to Section 4.1):

$$\delta(\pi_{a_1}(T_1 \bowtie_{dim} T_2)) \cup \delta(\pi_{a_2}(T_1 \bowtie_{dim} T_2)) \qquad (3)$$

This gives us the candidate triples that are part of one or more matching subgraphs.

We express a BitMat join using relational operators (as explained in Section 3.2). Expression (1) is a *filter* operation. Let $T_1'$ and $T_2'$ be:

$$T_1' = \delta(\pi_{dim}(T_1)), \quad T_2' = \delta(\pi_{dim}(T_2)) \qquad (4)$$

$$T_{res} = T_1' \bowtie_{dim} T_2' \qquad (5)$$

$$(T_{res} \bowtie_{dim} T_1) \cup (T_{res} \bowtie_{dim} T_2) \qquad (6)$$

Expression (4) is a *fold* operation, (5) is a *join* over *'dim'* or bitwise AND of the bitarrays folded by retaining *'dim'*, (6) is *unfolding* and ORing the BitMats associated with the triple patterns to get the result BitMat. Since $T_1$ and $T_2$ have the same columns (S, P, O), expression (6) can be written as

$$T_{res} \bowtie_{dim} (T_1 \cup T_2) \qquad (7)$$

Resolving expression (5) using (4) we get:

$$T_{res} = \delta(\pi_{dim}(T_1)) \bowtie_{dim} \delta(\pi_{dim}(T_2))$$
$$T_{res} = \delta(\pi_{dim}(T_1 \bowtie_{dim} T_2))$$

Hence expression (7) can be written as:

$$(\delta(\pi_{dim}(T_1 \bowtie_{dim} T_2))) \bowtie_{dim} (T_1 \cup T_2)$$

which can be expanded as

$$\underbrace{(\delta(\pi_{dim}(T_1 \bowtie_{dim} T_2)) \bowtie T_1)}_{\delta(\pi_{a_1}(T_1 \bowtie_{dim} T_2))} \cup \underbrace{(\delta(\pi_{dim}(T_1 \bowtie_{dim} T_2)) \bowtie T_2)}_{\delta(\pi_{a_2}(T_1 \bowtie_{dim} T_2))}$$

$$\delta(\pi_{a_1}(T_1 \bowtie_{dim} T_2)) \cup \delta(\pi_{a_2}(T_1 \bowtie_{dim} T_2)) \qquad (8)$$

Expression (8) gives the candidate triples produced by a Bit-Mat join, which is same as expression (3), which proves that the results produced by the join procedure on a BitMat are the same as applying transformation $T_{triple}$ on the results produced by a conventional SPARQL/SQL query engine.

## B. SINGLE JOIN QUERIES

**Table 3: Single Join Queries on UniProt-200k**

| 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 |