BitMat – Scalable Indexing and Querying of Large RDF Graphs
(Technical Report)

Medha Atre[1], Vineet Chaoji[2], Mohammed J. Zaki[1], and James A. Hendler[1]

[1] *Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy NY, USA*

[2] *Yahoo! Labs, Bangalore, India*

Abstract

The growing size of Semantic Web data expressed in the form of Resource Description Framework (RDF) has made it necessary to develop effective ways of storing this data to save space and to query it in a scalable manner. SPARQL – the query language for RDF data – closely follows SQL syntax. As a natural consequence most of the RDF storage and querying engines are based on modern database storage and query optimization techniques. Previous work has tried to use vertical partitioning using column stores (C-Store, MonetDB) and 6-way indexing (RDF-3X, Hexastore) for storage and querying of RDF data. Although these approaches perform well for highly selective queries, for queries having low-selectivity triple patterns, scalability of the querying method and optimizations still remain a challenge.

In this paper we present a new way of storing RDF graphs in run-length-encoded bit-vector format called BitMat, and we propose a novel two-phase SPARQL join query processing algorithm. In the first phase it prunes the candidate RDF triples, and in the next phase, it stitches the pruned RDF triples together to generate final results. Our query processing method does not build intermediate join tables and works directly on the compressed data. Our evaluation shows that BitMat not only provides an efficient method of storage of the RDF graphs, but our join query processing algorithm scales well for low-selectivity join queries, where state-of-the-art RDF query processors face problems.

1. Introduction

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

As the amount of RDF data on the web is increasing at break-neck speed, efficient storage and querying methods for RDF data are of prime importance. Various communities like bioinformatics, life sciences, social networks, as well as government are adopting RDF as a data standard. RDF datasets of few hundred million to a billion triples are becoming common. In this situation, the two major challenges are 1) efficient storage of RDF data, and 2) efficient querying of the stored data.

Disk-space is growing rapidly and a more efficient method of storing the data in compressed form allows even more data to fit in the same amount of disk space.More importantly, while querying, the disk-based data needs to be brought in main-memory to execute queries on it. Since

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

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

the amount of available main memory still remains much smaller than the secondary memory, efficient and scalable ways of querying the RDF data remain a big challenge.

Querying any large dataset involves fetching the data into main-memory, scanning the data and any indexes created over it, and executing query processing algorithm. For efficient storage, the data can be stored using standard compression algorithms, but it usually needs to be uncompressed while querying. A better alternative is to have a query algorithm that can work on the compressed data without uncompressing it. Previous work by D. J. Abadi et al [1, 2] addresses these issues to a certain extent by proposing compression techniques in column-oriented databases and using *lazy materialization* to work on compressed data as much as possible.

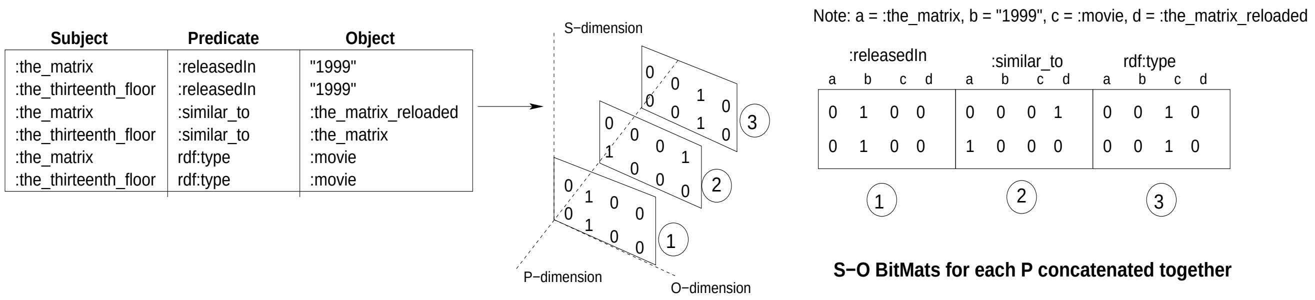
SPARQL join queries, also known as *Basic Graph Pattern matching* (BGP) or *conjunctive triple pattern* queries closely resembles SQL join queries. Any SPARQL join query can be systematically translated into a corresponding SQL join query [3]. These join queries can be broadly classified into three categories. The first class of queries are the ones having highly *selective*[3] triple patterns. E.g., consider a query *(?s :worksFor :Rensselaer)(?s :hassSSN "123-45-6789").* Since Rensselaer is small community and SSN is a unique attribute of a person, both the triple patterns are highly selective. The second class of queries are the ones with low-selectivity triple patterns, but which produce highly selective results. E.g., consider the query *(?s :residesIn :China)(?s :citizenOf :India).* Here each triple pattern in unselective – population of China is over 1.3 billion and over 1.2 billion people are citizens of India. But there are very few Indian citizens who reside in China, hence the result of the join of these two triple patterns is highly selective. The third class of queries are the ones having low-selectivity triple patterns producing low-selectivity join results. For example, consider a query *(?s :residesIn :India)(?s :hasProfession :Farming).* India being the country where agriculture is a large sector of the economy, this query will produce a lot of results.

Systems that generate various indexes on the RDF data do well on the first type of queries. Especially systems like Hexastore [5] and RDF-3X [6], which generate 6-way indexes can choose the appropriate index and pick the right set of triples at the beginning and make use of *merge-joins* instead of scanning a large amount of data. For the second type of queries – low-selectivity triple patterns but highly selective results – *join selectivity estimation* or precomputed join tables/indexes fetch benefits in query optimization to certain extent, although our evaluation shows that join selectivity estimation does not always help in improving the query performance. For the third type of queries – low-selectivity triple patterns generating a large number of results – even state-of-the art systems run into problems.

The main challenge while executing queries of second and third type as mentioned above, is to uncompress a large amount of data if it is in compressed form, fetch it in memory, and execute join queries over it by maintaining any intermediate results (which in turn can be quite large). Hence our goal is to build a scalable query algorithm which operates on the compressed data without generating intermediate join tables. This helps to keep the memory footprint of the system small in case of low-selectivity queries, compared to the contemporary approaches. Our key contributions in this work are:

1. A compressed data structure – BitMat – for storing the RDF data.
2. A novel 2-phase algorithm to execute SPARQL join queries, which precludes building large intermediate join tables in case of a low-selectivity query involving multiple joins.
3. Procedures and algorithms developed to work directly on the compressed data.
4. Thorough experiments across a wide range of dataset

[3] Selectivity of a triple pattern is low if there are more number of triples associated with it and vice versa [4].



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

Note: a = :the_matrix, b = "1999", c = :movie, d = :the_matrix_reloaded

:releasedIn				:similar_to				rdf:type			
a	b	c	d	a	b	c	d	a	b	c	d
0	1	0	0	0	0	0	1	0	0	1	0
0	1	0	0	1	0	0	0	0	0	1	0
1				2				3			

S−O BitMats for each P concatenated together

Figure 1: Basic method of BitMat construction

sizes ranging up to 1.33 billion triples, which constitutes some of the largest used to-date.

2. Related Work

Recent attention towards efficient storage and querying of RDF data has spawned a lot of different RDF-stores, such as – 4Store [7], BigOWLIM [8], AllegroGraph [9], Hexastore, RDF-3X, MonetDB [10], Jena-TDB [11], Jena-SDB, Jena with PostgreSQL, C-Store [12] and so on.

Out of these, some are commercial RDF stores and some are open-source efforts. Jena distributions, RDF-3X, and MonetDB are ongoing efforts to improve RDF storage and querying. MonetDB does not provide a native RDF store or a SPARQL query processor, but one can exploit the fact that RDF data typically has very less predicates as compared to the number of subjects and objects by storing the data into separate *predicate tables* using MonetDB's column storage. Also since most SPARQL join queries have bound predicate positions in the triple patterns, this kind of data organization turns out to be beneficial for efficient SPARQL join query processing.

RDF-3X and Hexastore make use of the fact that an RDF triple is a fixed 3-dimensional entity and hence they create all 6-way indexes (SPO, SOP, PSO, POS, OPS, OSP). Although Hexastore does share common indexes within these 6 indexes, e.g., SPO and PSO share the "O" index, without any compression, it suffers from 5-fold increase in the space required to store these indexes. RDF-3X goes one step further and compresses these indexes [13]. RDF-3X also implements several other join optimization techniques like RDF specific *Sideways-Information-Passing*, selectivity estimation, merge-joins, and using bloom-filters for hash joins [6].

Most RDF storage systems built on top of a database architecture typically use a left-deep join tree which requires materialization of the intermediate join results in case of a complex join query involving several join variables. Even in case of 6-way indexing, merge-joins cannot always be used to perform later joins if there is a long chain of join variables. In contrast to these, in our system instead of using sophisticated join optimization techniques, we have used some simple heuristics and a rule of keeping the data compressed without materializing the intermediate join results. This is explained further in Section 5.3. This helps to keep a large amount of required data in memory. We execute the join by following a novel algorithm, which propagates the constraints on the join-variable bindings among different join variables in the query. Our technique is reminiscent of the concept of *semi-joins* [14, 15] as discussed further in Section 5.2. We consider our query processing engine *light-weight* – light-weight on runtime memory consumption as well as optimization techniques. Through our evaluation, we have analyzed that our system outperforms the state-of-the-art systems like Hexastore, RDF-3X, and MonetDB in case of low-selectivity join queries, whereas for highly selective queries the contemporary systems perform better.

The work presented in this article presents comprehensive overview of previously unpublished work over BitMat structure, along with our work presented in [16, 17, 18], with the necessary details of the query processing algorithm which were not covered in any of the earlier published work.

3. BitMat – Compressed Index for RDF Data

Since RDF triple is a 3-dimensional entity (subject, predicate, object), conceptually RDF data can be represented as a 3D bitcube. Each dimension of the bitcube represents subjects (S), predicates (P), objects (O). Let V_s, V_p, V_o be the set of distinct subjects, predicates, and objects in the RDF data. Then the volume of the bitcube is $|V_s \times V_p \times V_o|$. Each cell in the bitcube represents a unique RDF triple that can be formed by the corresponding coordinates of S, P, O. A bit set to 1 represents presence of that triple in the RDF data. Further, this 3D bitcube can be flattened in 2D form by two methods.

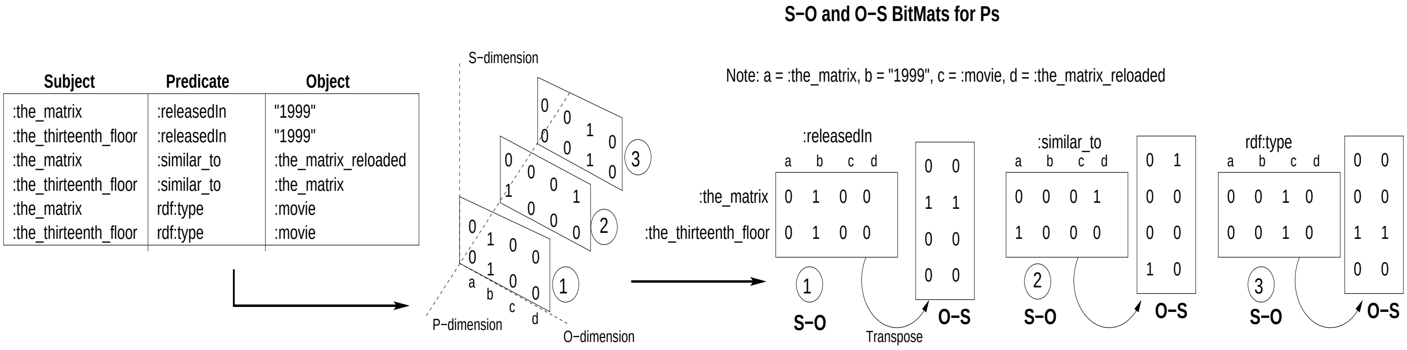
3.1. Basic Method

In this method, the bitcube is sliced along the P-dimension to get $|V_p|$ number of 2D S-O matrices. These matrices are concatenated together along the O-dimension to get one contiguous matrix of all the RDF triples – such a 2D matrix is called as BitMat. This procedure is elaborated in Figure 1. There are total $|V_s| \times |V_p| \times |V_o|$ possible triples with the given V_s, V_p, V_o sets, but it is observed that typically RDF data contains much fewer number of triples, hence the constructed BitMat is very sparse. We make use of this fact by applying *run length encoding* on each row of the BitMat. In run length encoding of a bit-vector, a bit-row of "0011000" will be represented as "[0] 2 2 3". That is, starting with the first bit value, we record alternating run lengths of 0s and 1s.

In the Basic Method, all the triples are first ordered on their subject IDs and in each subject row, in turn the triples are ordered as predicate IDs followed by the object IDs. We apply run length encoding on the entire row without maintaining the predicate boundaries. Conceptually this BitMat serves as a SPO index. This method of BitMat construction yields much better compression ratio to store the RDF data. But while executing queries it incurs additional overhead to process joins on the predicate and object dimensions (this will be elaborated further in Section 5).

3.2. Enhanced Method

In this method, similar to the Basic Method, the 3D bitcube is sliced along P-dimension, but instead of concatenating the S-O matrices and storing a single BitMat, we store the S-O matrices separately. Along with these, we store the transposed O-S matrices for each P as well. These essentially serve as PSO and POS indexes. Additionally, we slice the bitcube along S and O dimensions, which gives P-O and P-S BitMats respectively, and they serve as SPO and OPS indexes. We do not store O-P and S-P BitMats, because of the following reasons. In an RDF data, typically V_p is a much smaller set than V_s and V_o (for example, in the UniProt dataset containing 845 million triples, there are about 147 million distinct subjects and 128 million distinct objects, but only 95 predicates). This means that in an O-P BitMat, there will be many more rows than the number of columns. Since we apply run length encoding on each row, an O-P BitMat will yield much poorer compression ratio as compared to the respective P-O BitMat. Same analysis holds for S-P BitMat as well. Also since V_s and V_o are much larger sets than V_p, typically the number of triples associated with each unique S or O value are much smaller than the ones associated with each P value. Hence the number of triples in a P-O or P-S BitMat are much lesser than those in S-O or O-S BitMat. This essentially voids the need of having separate O-P or S-P BitMats, because if required, they can be constructed easily from the corresponding P-O or P-S BitMats or the required operations can be simply performed on the appropriate dimension of the P-O or P-



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

Figure 2: Enhanced Method of BitMat construction (example of S-O, O-S BitMats for each P)

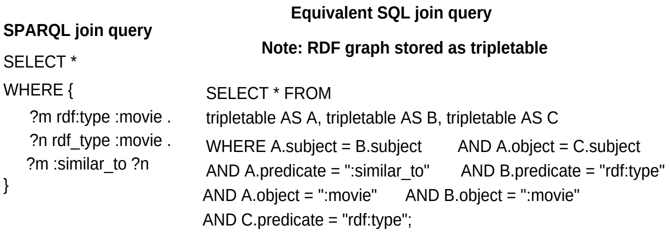
S BitMats (this will be elaborated further in Section 5). We apply run length encoding on each row of these BitMats. It is to be noted though that due to replication of the data (triples represented by S-O and O-S BitMat of each P value are the same), the amount of space required for BitMats created with the Enhanced Method is more than the Basic Method. To summarize, in the Enhanced Method of BitMat construction, we create 4 types of BitMats – S-O and O-S BitMats for each P value, P-O BitMat for each S value, and P-S BitMat for each O value (in all $2 * |V_p| + |V_s| + |V_o|$ BitMats). Figure 2 shows construction of S-O and O-S BitMats for each P value.

In the Enhanced Method, we also store the number of triples in each BitMat (this statistics is useful while executing our query algorithm as explained later). Along with this, we store two bitarrays – row and column bitarray – for S-O and O-S BitMats, which give a condensed representation of all the non-empty row and column values in the given BitMat. For example, in Figure 2, for the S-O BitMat of *":similar to"* predicate (marked by BitMat "2"), we store row bitarray "1 1" and a column bitarray "1 0 0 1". This means that there is at least one 1 in both rows and at least one 1 in column 1 and 4. Likewise for the O-S BitMat we store row bitarray "1 0 0 1", and column bitarray "1 1", respectively. For P-S and P-O BitMats we store only condensed representation of rows. We do not store condensed representation of columns for them, because the number of predicates (in turn the number of rows in the BitMat) are very less. Hence the condensed representation of columns can be easily formed on the fly by doing a bitwise OR of all the rows. These bitarrays are useful while performing "star join" queries (as elaborated in the Evaluation section).

The above construction reveals that each unique S, P, and O is mapped to a unique position along each dimension of the 3D bitcube and this position can be represented by an integer ID. We decide this mapping with the following procedure: Let V_{so} represent the $V_s \cap V_o$ set. Each element in V_{so}, along with the elements in V_s, V_p and V_o, is assigned an integer ID as follows:

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

The common subject-object identifier assignment facilitates the bitwise operations in join queries wherein an S position in one triple pattern is joined over an O position in another triple pattern (e.g. *?n* in the query in Figure 3). We store the meta-information of size of each of $|V_s|$, $|V_o|$, $|V_p|$, and $|V_{so}|$ sets in the configuration file. For the



SPARQL join query

```
SELECT *
WHERE {
    ?m rdf:type :movie .
    ?n rdf_type :movie .
   ?m :similar_to ?n
}
```

Equivalent SQL join query

Note: RDF graph stored as tripletable

```
SELECT * FROM
tripletable AS A, tripletable AS B, tripletable AS C
WHERE A.subject = B.subject       AND A.object = C.subject
AND A.predicate = ":similar_to"       AND B.predicate = "rdf:type"
AND A.object = ":movie"      AND B.object = ":movie"
AND C.predicate = "rdf:type";
```

Figure 3: An example of SPARQL join query

present considerations, we do not handle joins across S-P and P-O dimensions. Such queries are rare in the context of *assertional* RDF data. None of the benchmark queries published for the large RDF datasets have queries having joins over S-P or P-O dimensions. Hence overlapping S, P, O IDs except for the common S and O values do not pose a problem while processing a query. The BitMat construction described above is assuming a 3-dimensional RDF data. Sometimes RDF data can have forth dimension – context or graph name – which makes it a "*quad*" data as opposed to "*triple*" data. The forth dimension can be easily handled by building 3D bitcubes of quads having same context or graph name (the forth dimension). All quads having same context can be practically represented as *triples* having same forth dimension.

In practice, after mapping each string/URI in the RDF data to appropriate IDs as described above, we convert the entire list of triples in the ID form. We build each BitMat directly from the sorted list of triples instead of constructing an uncompressed row and then compressing it. For example, to build S-O BitMats of each P value, all the triples are first sorted on their P values followed by their S and O values. For each S-P pair, we read the O values and build a compressed row of the S-O BitMat directly. E.g., consider 3 triples with the same P-value "10", ("1122 10 3"), ("1122 10 1234"), ("1122 10 5678"). Note that they all have same S-value and hence will be placed in the same row in the S-O BitMat. Let $|V_o|$ be 10,000 (total number of O values). After reading the first O value as "3", we build the first part of a compressed row as "[0] 2", i.e., first 2 bits are 0s. Then we append a 1 bit for O value "3" and the encoding becomes "[0] 2 1". When we read the next 1 value "1234", we fill the gap in between with 1230 0s and update the encoding as "[0] 2 1 1230 1" and so on. The final compressed representation of this S row in the S-O BitMat will be "[0] 2 1 1230 1 4443 1 4321". This procedure of building a compressed row of a BitMat directly is same for all other type of BitMats. For P-S BitMats of each O value, we sort the ID based triples on O, P, S values and construct the BitMat as given above. For the Basic Method of BitMat construction, where the entire RDF data is represented using a single BitMat, we sort the ID based triples on S, P, O values respectively. The length of the row in this case is $|V_p| \times |V_o|$ and the bit position of a triple is decided as $(PID - 1) * |V_o| + OID$.

We build the condensed representation of rows and columns on the fly in a similar manner. The row bitarray helps in omitting storage of the empty rows and saves more space. While loading the BitMat the row bitarray assists in identifying the empty rows, which are not stored.

For the Enhanced Method, we store each type of BitMats (S-O, O-S, P-S, P-O) preceded by the number of triples and condensed representation of rows and columns in one giant file (one file for each type of BitMat) on the disk. We maintain meta-files which give the offset of each BitMat inside the giant file concatenating all BitMats of a type. Due to this, addition or deletion of triples might require moving a large amount of data, but if bulk updates are expected on the RDF data, all the BitMats can be rebuilt at once since the BitMat construction time even for very large data is very small. For addition of a single triple there are two cases: (1) S, P, O values of the triple are part of existing V_s, V_p, V_o sets respectively, (2) One or more of S, P, or O values of the newly added triple is *not* part of existing V_s, V_p, V_o sets. For the first case conceptually the volume of the 3D bitcube remains the same and only a particular bit-value gets toggled from 0 to 1. But in the second case, the new value of S, P, or O has to be added to the 3D bitcube which changes volume of the bitcube. The first case is easier to handle. For instance, if we want to add triple *(:the_thirteenth_floor :similar_to :the_matrix_reloaded)* to the sample data given in Figure 2, it affects 4 BitMats, i.e., S-O and O-S BitMats of P value *:similar_to*, P-S BitMat of O value *:the_matrix_reloaded*, and P-O BitMat of S value *:the_thirteenth_floor*. In such case, we can build four new BitMats, append them to the end of the giant file containing all BitMats of the respective type, and update the metafiles to give the new offset of the newly added BitMats.

Example of the second case is addition of triple *(:the_matrix_reloaded :rdf_type :movie)*. The value *:the_matrix_reloaded* does not exist in original V_s set. In such a case, we will need to rebuild all the BitMats to accommodate the newly added value (as it changes the volume of the bitcube from which the BitMats were built). Also if the newly added value changes the intersection set V_{so} then all other IDs will need to be reassigned (e.g., the example given above). For the data which is relatively stable, addition of such triples can be done by rebuilding all the BitMats occasionally (as building the BitMats from the ID representation of triples is much faster than initial parsing). However this can be expensive for highly dynamic data.

With respect to the construction described above, the RDFCube [19] system is conceptually closest to BitMat. RDFCube also builds a 3D cube of S, P, and O dimensions. However, RDFCube's design approximates the mapping of a triple to a cell by treating each cell as a hash bucket containing multiple triples. They primarily used this as a distributed structure in a peer-to-peer setup (RDFPeers [20]) to reduce the network traffic for processing join queries in a conventional manner. In contrast, BitMats compressed structure maintains unique mapping of a triple to a single bit, and also employs a different query processing algorithm. Further, RDFCube has demonstrated their results on a bitcube of only up to 100,000 triples, whereas we have used more than 1.33 billion triples in this paper.

4. BitMat Operations

The join query algorithm for BitMat structure is based on four primitive procedures: (a) *initialization*, (b) *filter*, (c) *fold*, (d) *unfold*. For the Basic Method of BitMat, we need to use all the four procedures, whereas for the Enhanced Method, we need to use only *initialization*, *fold*, and *unfold* methods.

4.1. Initialization (Basic Method)

We define *initialization* as an abstract procedure which depends on the underlying BitMat structure. In the Basic Method, the original BitMat – with all the triples – is always kept in memory. *Initialization* involves applying *filter* operation on the original BitMat to generate separate BitMats for each triple pattern in the query. The operation is described briefly in Algorithm(1). The initialization in case of the Enhanced Method depends on the triple pattern and is described later in Section 5.1.

Algorithm 1 Initialization (for the first type of BitMats)

```
Let $BM$ be the BitMat for the original RDF graph
for each triple pattern $\mathcal{T}$ in query do
    $BM_{\mathcal{T}}$ = filter($BM$, $\mathcal{T}$)
end for
```

4.2. Filter

Filter operation is used only for the Basic Method of BitMat construction. It is represented as '*filter(BitMat, TriplePattern) returns BitMat*'. It takes an input BitMat and returns a *new* BitMat which contains only triples that match the *TriplePattern*. For the query in Figure 3, *filter(BitMat, '?m :similar_to ?n')*, clears all the bits from BitMat except those having *:similar_to* as a predicate and returns a new BitMat containing only those triples. A triple pattern can have any of the following positions with fixed values – only S, only P, only O, S and P, S and O, or

P and O. For instance, triple pattern *(?m rdf:type :movie)* has P and O as fixed values.

The *filter* operation is performed as follows: if only S value is fixed, only the row corresponding to the S value in the original input BitMat is retained in the resulting BitMat and rest all rows are set to null. If only P value is fixed, we set a compressed mask bitarray of the size $|V_p| \times |V_o|$, with all the bits corresponding to the P value set to "1". For example, if $|V_p| = 3$ and $|V_o| = 10$ and the fixed value of P in the triple pattern is mapped to ID "2" (ref. Section 3, each unique string value of S, P, O is mapped to an ID unique within V_s, V_p, V_o subsets), the mask bitarray would look like, *"0000000000 1111111111 0000000000"* (but it will be in a compressed form). Note that this is an uncompressed representation with a whitespace inserted between group of object bits per predicate and is presented in this form only for the ease of reading a long bitarray. Whereas the actual storage of this bitvector is in compressed form without any delimiter for group of objects of each predicate. We do a bitwise AND of each row of the original BitMat and this mask bitarray. The procedure of doing bitwise AND and OR of compressed bit-vectors is described later in Section 4.5. The result of the bitwise AND is set as the corresponding row in the resulting BitMat. If O value in a triple pattern is fixed, we create a compressed mask bitarray of the size $|V_p| \times |V_o|$, with all the bits corresponding to the O value set to "1". For example, if $|V_p| = 3$ and $|V_o| = 10$ and the fixed value of O is mapped to ID "5" the mask bitarray would look like, *"0000100000 0000100000 0000100000"*. The compressed form of this mask bitarray is ANDed with each row of the original BitMat and the result is set as the corresponding row in the resulting BitMat.

For fixed values in S and P positions, the resulting BitMat just has one row corresponding to the S value. Within the row bits corresponding to other P values are masked out. For example, if S value is mapped to "5" and P value is mapped to "2", with $|V_p| = 3$ and $|V_o| = 10$, the fifth row in the original BitMat is ANDed with compressed form of "0000000000 1111111111 0000000000" and the result of this is set as the fifth row in the resulting BitMat. For fixed values in S and O positions, we create a run-length-encoded bitarray as is done for a fixed O value (shown above) and AND it with the corresponding S row in the original BitMat. The resulting BitMat has just one row which is the result of the AND operation. For fixed values in P and O positions, we build a compressed bitarray with just one bit corresponding to P and O values to 1 and AND it with each row of the original BitMat. For a triple pattern containing all variable positions, the original BitMat is just replicated as is and output as a resulting BitMat.

4.3. Fold

Fold operation represented as *'fold(BitMat, retainDimension) returns bitArray'* folds the input BitMat by retaining the *retainDimension*. In the Basic Method, a BitMat is actually flattened version of a 3D bitcube and hence has all 3 dimensions – subjects as rows and all the object columns per predicate are grouped together (while concatenating S-O matrices for each P). Hence for the Basic Method, the *fold* operation folds the two dimensions other than the one specified as *retainDimension*. For example, if *retainDimension* is set to 'object', then BitMat is folded along the predicate and subject dimensions resulting into a single bitarray. Intuitively, a bit set to 1 in this array indicates presence of at least one triple with the object corresponding to that position in the given BitMat. For the example BitMat given in Figure 4, it has 5 triples represented in ID form (1 1 1), (1 3 2), (1 3 4), (2 1 3), (2 3 2). Folding predicate dimension, gives us a single S-O matrix having with unique bit positions (1 1), (1 2), (1 4), (2 2), (2 3). Note that these are S-O values from original triples without the predicate value. A fold on subject dimension (i.e., removing subject positions) gives us bits in unique O position 1, 2, 3, 4, which is nothing but the object bitarray

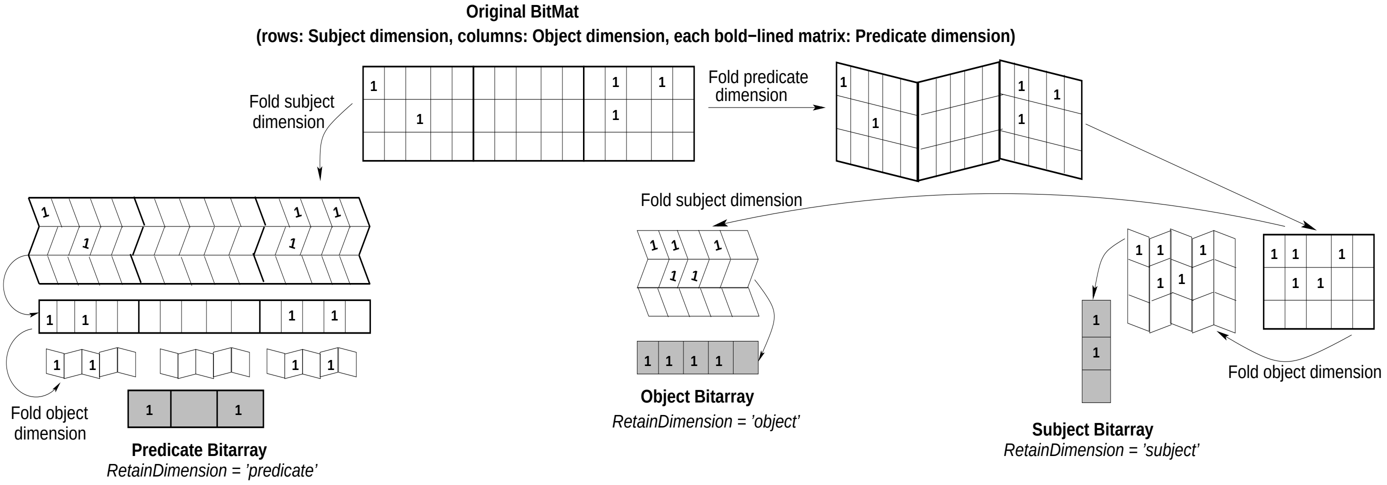

Original BitMat
(rows: Subject dimension, columns: Object dimension, each bold−lined matrix: Predicate dimension)
Fold subject dimension
Fold predicate dimension
Fold subject dimension
Fold object dimension
Fold object dimension
Predicate Bitarray
RetainDimension = 'predicate'
Object Bitarray
RetainDimension = 'object'
Subject Bitarray
RetainDimension = 'subject'

Figure 4: Conceptual view of folding an BitMat for a *retainDimension* – Basic method of BitMat construction

"11110".

Conceptually, the *fold* operation is similar to a *projection* operation (SELECT clause) on a relational table and the *retainDimension* is same as the column/attribute of the table that is projected out. As in case of projection operation, only the unique values of the *retainDimension* are returned in the form of bitarray.

In the implementation of the *fold* operation, if the *retainDimension* is set to 'subjects', we simply check each row of the BitMat for emptiness. In the resulting bitarray, a bit corresponding to a row in the BitMat is set to 1 if that row is not empty. If the *retainDimension* is set to 'predicate', we maintain an initially empty bitarray of $|V_p|$ bits, say *P-bitarray*. We scan each compressed row in $|V_p|$ parts (without actually uncompressing it). If any of the object bits grouped for a given predicate has a 1 bit, we set the corresponding predicate bit in P-bitarray. E.g., if $|V_p| = 3$ and $|V_o| = 10$ and the subject row is *'1000100000 1001100100 0000000000'*, we check the 3 parts of the subject row *'1000100000', '1001100100', '0000000000'* separately. Since only the first 2 parts have at least one '1' bit, the P-bitarray is updated as *P-bitarray = P-bitarray OR '110'*. We check the next subject row similarly in $|V_p|$ parts and update the P-bitarray. If the *retainDimension* is set to 'objects', similar to the previous example, we check each subject row in $|V_p|$ parts and do a bitwise OR of each part. E.g., for a subject row *'1000100000 1001100100 0000000000'*, *O-bitarray = O-bitarray OR '1000100000' OR '1001100100' OR '0000000000'*, where O-bitarray is initially empty with $|V_o|$ bits.

In the Enhanced Method of BitMat construction, any given BitMat has only 2 dimensions and the orientation of these dimensions in the 4 types of BitMats is different. E.g., an S-O BitMat of a P value has S and O dimensions as rows and columns respectively, whereas for a P-S BitMat P and S dimensions are rows and columns respectively. Hence for a *fold* operation, *retainDimension* is specified as 'rows' or 'columns'. If in an S-O BitMat of a P value, *retainDimension* is set to 'columns', then BitMat is folded along the subject 'rows' resulting into a single bitarray, i.e., all the subject rows are bitwise ORed together to give an O-bitarray. If the *retainDimension* is 'rows' for a S-O BitMat, we simply check each row for non-emptiness. A 1 bit in S-bitarray represents that the corresponding row in S-O BitMat is not empty. S-bitarray is $|V_s|$ bit long.

4.4. Unfold

Unfold operation specified as *'unfold(BitMat, MaskBitArray, retainDimension)'*, unfolds the *MaskBitArray* on the BitMat on the *retainDimension*. Intuitively, in the unfold operation, for every bit set to 0 in the *MaskBitArray* all the bits corresponding to that position of the *retainDimension* in the BitMat are cleared.

Unfold is exactly opposite to the fold operation. In the Basic Method of BitMat construction, if *retainDimension* is 'subject', *MaskBitArray* has $|V_s|$ bits. For each bit set to 0 in the *MaskBitArray*, the corresponding row in the BitMat is deleted completely. If the *retainDimension* is set to 'predicate'. We scan each row of the BitMat in $|V_p|$ parts. For a bit set to 0 in the *MaskBitArray*, the object bits corresponding to that predicate value in a row are all set to 0. E.g., if the original row is *'1000100000 1001100100 0100110000'* and MaskBitArray is '101', it is unfolded on the row as *(ResultRow = '1000100000 1001100100 0100110000' AND '1111111111 0000000000 1111111111').* *MaskBitArray* is unfolded on 'object' dimension in a similar manner. Keeping the example row same as before, if *MaskBitArray* is *'1101101101'* with the *retainDimension* set to 'object', each row of the BitMat is updated as *(ResultRow = '1000100000 1001100100 0100110000' AND '1101101101 1101101101 1101101101').*

For the Enhanced Method, unfold operation is simpler. For an unfold on 'rows' dimension, all rows corresponding to a 0 bit in the *MaskBitArray* are deleted and for an unfold on 'columns' of a BitMat, the *MaskBitArray* is ANDed with each row of the BitMat and the result is set as the updated row in the BitMat.

4.5. Bitwise Operations of Compressed Bit-vectors

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

5. Join Processing Algorithm

Before describing our join processing algorithm, we would like to note some properties of the join process [14, 15].

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

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

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

To elaborate the use of these properties, consider the query given in Figure 3. *?m* and *?n* appear in the same triple pattern *(?m :similar_to ?n).* If we perform a join of *(?m :similar_to ?n) (?m rdf:type :movie)* first, we get two bindings for the variable *?m* viz. *:the_matrix* and *:the_thirteenth-_floor* and two for *?n :the_matrix_reloaded*

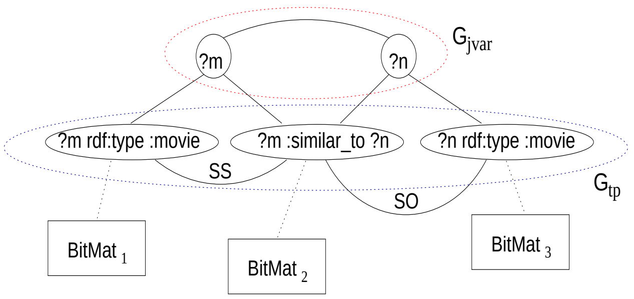


Figure 5: Graph $\mathcal{G}$ for the query in Figure 3

and *:the_matrix* corresponding to *?m*'s bindings. When we do the join between *(?n rdf:type :movie)(?m :similar_to ?n)*, we consider bindings generated for *?n* after the first join on *?m*. After the second join on *?n*, binding *:the_matrix_reloaded* for *?n* gets dropped, since the triple *(:the_matrix_reloaded rdf:type :movie)* is not present in the dataset. Hence the triple *(:the_matrix :similar_to :the_matrix_reloaded)* gets dropped from the triples associated with *(?m :similar_to ?n)* which in turn drops the binding *:the_matrix* for *?m*.

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

The join processing algorithm for both methods of BitMat construction is the same, except that for the Basic Method, an additional step of *initialization* is required (ref. Algorithm 1).

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

1. Each triple pattern in the join query is denoted by a *tp-node* in $\mathcal{G}$. Hence forth we use the terms "tp-node" and "triple pattern" interchangeably. A *jvar-node* in $\mathcal{G}$ corresponds to a join variable in the query. Hence forth we use terms "jvar-node" and "join variable" interchangeably.
2. An undirected, unlabeled edge between a jvar-node and a tp-node exists in $\mathcal{G}$ if that join variable appears in the triple pattern represented by the tp-node. This edge represents the dependency between triples associated with the tp-node and the join variable bindings (ref. Property 1).
3. An edge exists between two jvar-nodes if the two join variables appear in the same triple pattern. This undirected, unlabeled edge represents the dependency between their bindings (ref. Property 2).
4. An edge between two tp-nodes exists if they share a join variable between them. This is an undirected, *labeled* edge with potentially multiple labels. Multiple labels can appear if the two triple patterns share more than one join variables. The labels denote the type of join between the two triple patterns – *SS* denotes subject-subject join, *SO* denotes subject-object join etc.

For a query having no *Cartesian joins*[5], the constraint graph $\mathcal{G}$ is always connected. Figure 5 shows the constraint graph for the join query given in Figure 3.

5.1. Initialization (Enhanced Method)

Before starting the pruning algorithm, we *initialize* each tp-node by loading the triples which match that triple pattern. This step differs based on the BitMat construction method. For the Basic Method, we make use of the *filter* operation as described in Section 4.2. That is, for each triple pattern, we apply *filter* operation on the original BitMat and keep only triples that satisfy the triple pattern. For the Basic Method, the original BitMat is always kept in memory.

For the Enhanced Method, we initialize the BitMat associated with each triple pattern using the four types of

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

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

stored BitMats. In Section 3.2 we elaborated the construction of four types of BitMats viz. S-O and O-S for each P value, P-S for each O value and P-O for each S value. We assume that a given query does not have any triple pattern with all variable positions (e.g. *(?x ?y ?z)*. The case of handling all-variable triple patterns is discussed later in Section 5.6. E.g., if the triple pattern in the query is of type *(?s :p2 :o321)* then we load only one row corresponding to ":p2" from the P-S BitMat created for ":o321". If the triple pattern is of type *(?s :p6 ?o)* then we load either the S-O or O-S BitMat created for ":p6". If *?s* is a join variable and *?o* is not, we load S-O BitMat and vice versa. If both, *?s* and *?o*, are join variables, then the decision depends on whether *?s* will be processed before *?o*. If a join over *?s* is processed before *?o*, we load S-O BitMat and vice versa.

If we have a triple pattern of type *(:s2 ?p :o6)*, then first we decide whether P-S BitMat for ":o6" has fewer triples or P-O BitMat for ":s2" has fewer triples. If P-S BitMat has less number of triples, then we load the P-S BitMat by keeping only the bit corresponding to ":s2" in each row and mask out all other bits. Note that all these operations are done directly on the compressed BitMats.

This way, for both, Basic and Enhanced methods, at the end of initialization step each triple pattern in the constraint graph $\mathcal{G}$ has a separate BitMat associated with it which contains only triples that satisfy that triple pattern.

5.2. Pruning RDF Triples

First, we consider an induced subgraph $\mathcal{G}_{jvar}$ of $\mathcal{G}$ containing only jvar-nodes. Since we do not handle queries with *Cartesian joins*, by the construction of graph $\mathcal{G}$, $\mathcal{G}_{jvar}$ is also always connected (see Figure 5). $\mathcal{G}_{jvar}$ can be cyclic or acyclic. Next, we embed a tree on $\mathcal{G}_{jvar}$ discarding any cyclic edges. To propagate the constraints on join variable bindings (Property 2), we walk over this tree from root to the leaves and backwards in breadth-first manner. At every jvar-node, we perform an intersection of the bindings generated by its adjacent tp-nodes and after the intersection, we drop the triples from tp-node BitMats as a result of the dropped bindings.

It can be seen that by the construction of graph $\mathcal{G}$ and following the tree over $\mathcal{G}_{jvar}$, constraints on the join variable bindings get propagated to other jvar-nodes through the tp-node BitMats (when the triples get dropped). This procedure is elaborated in Algorithm(2) and (3).

A *topological sort* of an undirected tree is nothing but enumerating all the nodes from root to leaves in a breadth-first-search fashion. For each node in the topological sorted list of join variables, we call *prune_for_jvar* (Lines 2 – 4 in Algorithm(2)). A topological sort ensures that a child jvar node always gets processed after all of its ancestors. The bitwise AND between folded bitarrays in *prune_for_jvar(J)* computes the *intersection* of all the bindings generated by the tp-nodes which contain J (Lines 2 – 5 in Algorithm(3)). According to Property 1, for any binding dropped in the intersection, the respective triples are removed from the BitMats associated with the tp-nodes which contain J using the *unfold* operation (Lines 6 – 9 in Algorithm(3)). *getDimension* returns the position of J in the BitMat of the triple pattern. For the Enhanced Method of BitMat construction, *getDimension(?n, (?m :similar_to ?n))* can return *column* or *row* depending on whether it is an S-O or O-S BitMat. For the Basic Method *getDimension* returns the position of the variable, i.e., "subject", "predicate", or "object".

Algorithm 2 Pruning Step

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

One such pass over all the jvar-nodes ensures that the constraints are propagated to the adjacent jvar-nodes in the direction from root to leaves of the tree. For the

Algorithm 3 prune_for_jvar(jvar-node J)

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

backward propagation of constraints, we traverse jvar-nodes second time by following the reverse order of the first pass (effectively a bottom-up pass) (Lines 5 – 8 in Algorithm(2)). The leaves of the tree embedded on $\mathcal{G}_{jvar}$ appear last in queue q. Since they are processed last in the first traversal over the tree, in the second traversal, we directly start with the parent nodes of these leaves (Line 5 in Algorithm(2)). Notably, since we take *intersection* of the bindings in each pass, the number of triples in the tp-node's BitMat decrease monotonically as the constraints are propagated.

During the pruning phase, we use a simple statistical optimization technique, which we call, *early stopping condition*. That is, while performing the pruning at each jvar-node, at any point if the $MaskBitArr_J$ contains all 0 bits, that is a direct evidence of the query generating empty set of results. If such a condition occurs we exit the query processing at that point telling that the query has 0 results. This avoids unnecessary further processing of other join variables and *fold/unfold* operations over BitMats.

At the end of Algorithm(2), each tp-node contains a much reduced set of triples. Typically, when $\mathcal{G}_{jvar}$ is acyclic, this set of triples is also *minimal*, i.e., each triple in the BitMat of a triple pattern is necessary to generate one or more final results. In other words, if we consider the final result of a SPARQL/SQL query and project out the unique triples matching each triple pattern in the query, they will be same as the set of triples left in the BitMats of respective tp-nodes. But this depends on the way the $\mathcal{G}_{jvar}$ is traversed and is elaborated below in Section 5.4. If $\mathcal{G}_{jvar}$ is cyclic, then the set of triples is not guarantied to be minimal. But in any case, any unwanted set of triples get dropped in the following phase of final result set generation.

SPARQL join queries that BitMat handles fall in the category of *equi-joins* in the database systems. An equi-join is where columns in different tables are joined by the condition of equality only. Equi-join queries do not contain FILTER conditions, e.g., ">", "<=" or regular expressions. Equi-joins in SPARQL are nothing but join queries without OPTIONAL, UNION, or FILTER clauses. OPTIONAL, UNION, or FILTER queries can be processed with additional modifications to BitMat's join query processing algorithm. SPARQL basic graph pattern (BGP) queries are the building blocks of the SPARQL queries. They are also the most performance intensive queries. In view of space limitations and to keep the scope of this article limited, we have not discussed ways of processing OPTIONAL, UNION, and FILTER queries.

5.3. *Results of Intermediate Joins*

As explained by the pruning phase in the previous subsection, we monotonically go on removing triples from the BitMats associated with each triple pattern as the pruning algorithm progresses. This process is distinctly different from the pairwise joins performed in a conventional join query processing algorithm. For example, consider a query *(?s :p1 ?x)(?s :p2 ?y)*. After the first pruning, i.e., the intersection between bindings of *?s* generated by each triple pattern, let the number of triples remaining in the BitMat associated with (?s :p1 ?x) be 5 and the ones in (?s :p2 ?y) be 8. So the total number of triples are 13. In the conventional query processor, the results of this join will be materialized to generate (8*5) 40 intermediate results (rows). Thus by not materializing intermediate join results, and by only removing the triples from BitMat, we achieve a better control over the required memory for the query processing.

5.4. Minimal Triple Set Generation

Our pruning algorithm closely resembles the idea of *semi-joins* [14, 15]. Semi-joins are *half-joins* which *reduce* the number of candidate tuples in the tables involved in a join, such that only the tuples necessary to produce the final join results are left in them. Semi-join's algorithm ensures that a join query satisfying certain properties can be *fully reduced*, i.e., all the tables involved in the query have a minimal set of tuples. To show BitMat's pruning algorithm can fully reduce all the BitMats of the triple patterns in a query we show: 1) a correspondence between the structure of semi-join's query graph and $\mathcal{G}_{jvar}$, and 2) an analogy between semi-join operations on the query graph and pruning method on $\mathcal{G}_{jvar}$.

Semi-join: A semi-join between two tables R_1 and R_2 over attributes i and j is represented as $R_1.i \ltimes R_2.j = \{r_1 \in R_1 | r_1.i \in (\sigma_i(R_1) \cap \sigma_j(R_2)\}$. In other words, only tuples r_1 in R_1 whose i attribute value exists in one of the j attribute values of R_2 are retained. Semi-joins are restricted to only join over a single attribute between two tables, e.g., $R_1.i \ltimes R_2.j$ and $R_1.k \ltimes R_3.l$ is a valid semi-join, but $R_1.i \ltimes R_2.j$ and $R_1.k \ltimes R_2.l$ is not.

Semi-joins construct a *query graph* G_Q such that each table in the join represents a node in G_Q. If two tables are joined over a column, there is an edge between the corresponding nodes in G_Q. In case of a SPARQL join query, each triple-pattern is equivalent to a table in the SQL join. Consider the SPARQL query on a movie database as shown in Table 1. Treating each triple pattern in the SPARQL query as a table, the query graph G_Q created by semi-joins is given in Figure 6. The node IDs in G_Q corresponding to each triple pattern are given in the adjacent column.

In semi-joins, the query graph construction takes care of removing *redundant cycles*. For example, in case of the query graph given in Figure 6, nodes l, m, p, q, r can be connected to each other as they all share join variable *?a*. By transitivity, *($l.a = m.a$ and $m.a = p.a$)* implies *($l.a = p.a$)*. As a result, an explicit edge between nodes l and p creates a redundant cycle. Hence all the conditions and corresponding edges that create a redundant cycle in G_Q are dropped. Semi-join's algorithm has proved in [14, 15] that when G_Q is acyclic, all the tables involved in the query can be *fully reduced*.

Table 1: Example query

Triple Pattern	Node ID in graph G_Q
?a :similar_to ?b	*l*
?a rdf:type :movie	*m*
?a :has_director ?e	*q*
?e rdf:type :Person	*v*
?e :name "Andy Wachowski"	*w*
?a :has_actor http://keanu.personal.com	*p*
?a :has_actress http://moss.personal.com	*r*
?b :similar_to ?c	*n*
?b rdf:type :movie	*o*
?c :has_actor ?d	*s*
?d rdf:type :Person	*t*
?d :name "Leonardo DiCaprio"	*u*

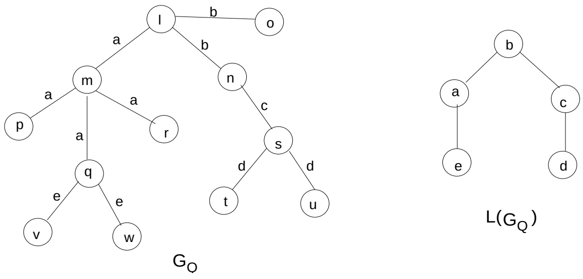


Figure 6: Example of G_Q and its line graph

BitMat Reducibility: We show BitMat's reducibility via semi-join's reducibility. First we show an analogy between the structure of G_Q and $\mathcal{G}_{jvar}$. We can show that graph $\mathcal{G}_{jvar}$ in BitMat's pruning algorithm is a *line graph* [21] of G_Q. A line graph $L(G)$ of original graph G is a graph, such that each node in $L(G)$ represents a unique edge in G. Two nodes in $L(G)$ are connected if their corresponding edges in G have a common vertex between them.

As shown previously, we can construct graph G_Q from any SPARQL join query. Let G_Q be an edge-labeled graph such that the label on the edge represents the variable over which two triple patterns are joined. For instance, edge *(l, m)* has label *?a* in Figure 6. Let us construct a line graph of G_Q such that each edge in G_Q is represented by a node. Let the label of this node be same as the original edge label. In other words we want to construct a graph of SPARQL join query where each node is a join variable and there exists an edge between two join variables if they share at least one triple pattern among them. G_Q can have multiple edges having same label which gives rise to multiple nodes with the same label, and hence redundant cycles and cliques in the line graph. We merge all the cliques having same node labels and represent them with a single node. This construction is exactly same as $\mathcal{G}_{jvar}$ which is an induced subgraph of the constraint graph $\mathcal{G}$ (ref Section 5).

Next we want to show that when G_Q is acyclic, $\mathcal{G}_{jvar}$ is also acyclic. Let us define the *effective degree* of a node in G_Q as the number of incident edges having *distinct edge labels*. E.g., node l in Figure 6 has effective degree of 2, since it has a and b as the only 2 distinct edge labels incident on it. As per the construction of line graph of an acyclic G_Q, $L(G_Q)$ can have cycles if G_Q has one or more nodes having effective degree greater than 2. A triple pattern in a SPARQL join query is a 3-dimensional entity (S, P, O positions which can either be variables or fixed values). Hence in a graph G_Q of a SPARQL equijoin query, each vertex can have maximum effective degree of 3 (when all 3 are join variables in case of a triple pattern like *(?x ?y ?z)*). In the current BitMat pruning algorithm, we do not consider triple patterns with all variable positions (each triple patterns has at least one fixed position).

Property 4. *The above construction ensures that in G_Q for a SPARQL join query, each vertex has effective degree of 1 or 2.*

Property 5. *Construction of G_Q ensures that a node having a specific incident edge label "e" has at least one path to every other node having edge "e" incident on it, such that the only edge labels on that path are "e" (we call it an e-path). This also implies that when G_Q is a tree, two edges cannot share the same label "e", unless they have an e-path between them (because there is a unique path between any two nodes in a tree).*

Lemma 1. *$\mathcal{G}_{jvar}$ is acyclic if G_Q is acyclic and holds Property 4 and 5.*

Proof. We prove this by contradiction. Let line graph $\mathcal{G}_{jvar}$ have one or more cycles. We consider the shortest cycle in $\mathcal{G}_{jvar}$.

Case 1: Let $\mathcal{G}_{jvar}$ have a shortest cycle of length 3 formed by 3 nodes, a, b, c[6]. Let these nodes correspond to 3 distinct edge labels a, b, c in G_Q. They will form a cycle in $\mathcal{G}_{jvar}$ if they are incident on same node in G_Q. This is a contradiction to Property 4 as given above. Consider two edges with labels a and b in G_Q. By construction, they share at least one node among them, say n_1. n_1 cannot have any other edge label incident on it (Property 4). Let another edge with label b share a node n_2 with edge c. By Property 5, the first and second edges with label b must have a *b-path* between them. Consider another edge with label c and an edge with label a. Let them share a node n_3 among them. By Property 5, the first and second edges with label c have a *c-path* among them. Similarly the first and second a edges have an *a-path* between them. This gives rise to a cycle in G_Q which is a contradiction. Hence you cannot have a 3-cycle in $\mathcal{G}_{jvar}$.

Case 2: Let $\mathcal{G}_{jvar}$ have a shortest cycle of length $m > 3$ formed by m nodes, $e_1..e_m$. These nodes represent edges with labels $e_1..e_m$ in G_Q. These edges cannot be all incident on the same node in G_Q as that will violate Property 4 and it will also give rise to a clique in $\mathcal{G}_{jvar}$, making the shortest cycle of length 3 which violates our original assumption – that the shortest cycle is of length m. Consider two adjacent edges e_1, e_2 in G_Q. They share a node among them and as per Property 4 this node cannot have any other edge label incident on it. Consider some other

[6] An undirected graph cannot have a cycle of length lesser than 3.

two edges with label e_2 and e_3 that share a node between them. The first and second e_2 edges have an e_2*-path* between them. Considering this way up to edges e_m and e_1, let them share a node among them which is not same as the nodes on which all earlier edges $e_1..e_{m-1}$ are incident. But by Property 5, the second e_1 edge has to have a e_1-path to the first e_1 giving rise to a cycle in G_Q.

Hence it is proved that $\mathcal{G}_{jvar}$ is acyclic when G_Q is acyclic with effective degree of any node 1 or 2. □

Next we show a correlation between the semi-join operations on G_Q and our method of traversing $\mathcal{G}_{jvar}$ in the pruning step. Semi-joins define two operations **UP** and **DN** on a G_Q and they prove that an **UP** operation followed by **DN** on G_Q fully reduces all the tables in the query if G_Q is a tree. For that let us select any table-node and make it the root of G_Q, l is the root in our example. Semi-joins define an **UP** operation as: convert G_Q into a directed graph by directing each edge upward from child to its parent. A directed edge, say *(m, l)* in G_Q, represents a semi-join ($R_l \ltimes R_m$). Do a topological sort of such directed G_Q to get a series of semi-joins from the bottom to the root of the tree. Lemma 4 in [15] proves that one such bottom-up pass on directed G_Q removes all the tuples not needed for the final result generation from the root of the tree (e.g., table l in our example). That is – the root node is *fully reduced*. Further extending this Lemma, Theorem 1 in [15] proves that a top-down pass on G_Q by following the reverse order of semi-joins fully reduces each table in the query. This is defined as the **DN** operation on G_Q. Together it proves that a **UP.DN** pass on G_Q fully reduces each table in the query.

To complete the proof sketch for BitMat's reducibility, we finally show that a **UP.DN** operation on acyclic G_Q is equivalent to a bottom-up followed by a top-down pass (**UP.DN**) on $\mathcal{G}_{jvar}$. This in turn fully reduces the set of candidate RDF triples. For the sake of simplicity, we adhere to the following 2 rules:

- *Rule 1:* While making a bottom-up pass on G_Q, when we encounter a new edge label, say "e", we process all edges (corresponding semi-joins) with label "e" in G_Q. We might face a situation, where a particular "e"-edge cannot be processed since its children are not processed yet in the **UP** operation. In that case, we do not process any of the "e"-edges until all the children of all the "e"-edges are processed first. This restriction ensures that in a bottom-up sequence of edges, all edges with the same label appear together and are not interleaved with any other edge labels. Due to Property 5 outlined above, enforcing this restriction is possible and it still gives a legitimate bottom-up pass on G_Q.

- *Rule 2:* We choose root node of $\mathcal{G}_{jvar}$ such that the corresponding edge label in G_Q is incident on the root of G_Q. Alternately, we can choose a root of $\mathcal{G}_{jvar}$ first and choose root of G_Q such that the edge label corresponding to the root of $\mathcal{G}_{jvar}$ is incident on it.

Lemma 2. *A bottom-up and top-down pass (**UP.DN** operation) on an acyclic G_Q is equivalent to **UP.DN** operation on its line graph $\mathcal{G}_{jvar}$ with respect to the ordering between edge labels when Rule 1 and 2 above hold.*

Proof. By construction of $\mathcal{G}_{jvar}$, all the edge labels which are incident *only* on the leaf nodes in G_Q make the leaf nodes in $\mathcal{G}_{jvar}$ (e.g. edge labels "e" and "d" in Figure 6). Also the adjacency between edge labels in G_Q is preserved in $\mathcal{G}_{jvar}$ in the form of adjacency between corresponding nodes. Hence a bottom-up pass over G_Q (adhering to Rule 1) above produces a legitimate bottom-up pass on $\mathcal{G}_{jvar}$. □

We would like to point out a key difference between semi-joins and BitMat's pruning algorithm. In semi-joins, a bottom-up pass over G_Q in Figure 6, represents ($q.e \ltimes v.e$, $q.e \ltimes w.e$, $s.d \ltimes t.d$ $n.c \ltimes s.c$, $m.a \ltimes q.a...l.b \ltimes o.b$). This means that 0 or more tuples in table q are dropped as a result of the semi-join between q and v, but v is not updated. Similarly, a semi-join between q and w updates *only* tuples in q. Unlike the semi-join algorithm, in case of BitMat's pruning algorithm, while processing a join variable, say e which is shared among 3 triple patterns v, q, w, updates are made to all 3 BitMats associated with the respective triple patterns (ref. Lines 6 – 9 in Algorithm 3). In this case, the set of triples left in the BitMats is a subset of the triples left by the semi-join's algorithm, hence BitMat's pruning method not only fits the semi-joins concept but also does more aggressive pruning.

Although we have proved that **UP.DN** operation over $\mathcal{G}_{jvar}$ ensures minimal triple set generation, if the leaves of the $\mathcal{G}_{jvar}$ tree are join variables associated with low-selectivity triple patterns, it can often hamper the overall performance. We observed that instead if we process the join variables associated with more selective triple patterns first (although they might not follow the order enforced by **UP.DN** operation), they help in faster pruning of the triples overall. Although it might not guaranty minimal triple generation, this avoids the overhead of processing non-selective BitMats first, thereby improving the query performance. Hence as elaborated by Algorithms 2 and 3 in Section 5.2, we process $\mathcal{G}_{jvar}$ nodes in the root-to-leaves order first (top-down pass), followed by the reverse order of it (bottom-up pass), essentially making a **DN.UP** operation instead of **UP.DN**. We also carefully choose the root of $\mathcal{G}_{jvar}$ as follows.

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

After choosing the root of $\mathcal{G}_{jvar}$, a **DN** pass can turn out to be is equivalent to **UP** pass and vice versa if the $\mathcal{G}_{jvar}$ has one single path in it. For example, in case of a $\mathcal{G}_{jvar}$ having 3 join variables connected as *(?a – ?b – ?c)*, if either "a" or "c" is chosen as the root of $\mathcal{G}_{jvar}$, then the **UP** operation is same as **DN** operation and vice versa.

5.5. Generating Final Results

After the pruning phase, we are left with a much reduced set of triples associated with each triple pattern. Intuitively, each BitMat of a triple pattern can be viewed as a compressed table in a relational database. Hence, one way of producing the results is to simply materialize these BitMats into tables and perform standard joins over them. But our goal is to avoid building intermediate join results; which precludes a 2-way sequence of joins as done in a typical SQL query processor. In our method, we build and output an entire resulting row of variable bindings, which is similar to *multi-way* joins [22].

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

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

A simple brute force approach can be as follows: Choose say $BitMat_1$, pick a triple from it. This triple will generate bindings for the variables in tp_1. Store these bindings in the map. Next pick the first triple from $BitMat_2$ and generate bindings for the variables in tp_2. If tp_2 and tp_1

share one or more join variables, check the map if the variable bindings generated by both of them are the same, if not, pick a second triple from $BitMat_2$. Repeat this procedure until you get the variable bindings consistent with the ones stored in the map. Then consider $BitMat_3$ and repeat the same procedure as described above. Repeat this procedure up to the last $BitMat_n$ in the query. If a triple in $BitMat_n$ generates valid bindings for all k variables in the map, output one result. Now start with $BitMat_1$ again and choose the second triple, store the bindings for the variables in tp_1, and repeat the same process. In general, while generating variable bindings from any $BitMat_i$, check all the variable bindings stored in the map.

Since BitMat is a fully inverted index structure, we instead devise the following method which speeds up the above procedure by several orders of magnitude: In general, a BitMat having fewer triples generates fewer unique bindings for the variables in its tp-node. This means that in the final results of the query, these bindings will get repeated more often in different result rows than other bindings (just like the product of two columns where the first column has fewer rows than the other – values from the first column get repeated more often in the product). Making use of this fact, we choose a BitMat as $BitMat_1$, which has the least number of triples, to be processed first (similar to the way of choosing the table having least number of triples to join first), generate bindings for the variables in tp_1, and store them in the map. Next instead of picking $BitMat_2$ randomly, we pick a tp_2 which shares a join variable with tp_1. Depending on the variable bindings stored in the map, we directly locate the triples which can satisfy these bindings inside $BitMat_2$. Recall that BitMat being a completely inverted index structure, it is easy to locate specific triples. If no such triple exists in $BitMat_2$, we discard the variable bindings in the map, go back to $BitMat_1$, and pick the second triple from it to generate new bindings (this can happen in case of a cyclic $\mathcal{G}_{jvar}$ or if we do a **DN.UP** pass on $\mathcal{G}_{jvar}$). If $BitMat_2$ generates variable bindings consistent with $BitMat_1$, pick tp_3 which shares join variables either with tp_1 or tp_2. Considering the constraint graph given in Figure 5, let $\mathcal{G}_{tp}$ be an induced subgraph of $\mathcal{G}$ having only triple patterns (tp-nodes) and edges between them. We make use of $\mathcal{G}_{tp}$ to make the choice of the next tp-node at every step. Hence it can be seen that after one walk over all the tp-nodes of $\mathcal{G}_{tp}$, if the map has all k variables mapped to bindings, we output one result. The procedure is repeated again until all the triples in $BitMat_1$ are exhausted.

The *worst-case* complexity of this operation is $\prod_{i=1}^{n} triples_i$, where $triples_i$ are the number of triples left in BitMat of tp_i. But since we exploit the indexed nature of BitMat and its small size with a reduced set of triples, this process is faster than the worst case complexity.

Presently the final result generation phase doesn't have a facility of selecting specific variables for projection. It always projects out bindings of all variables in the query unless it is a "star join" (involving only one join variable and only that is projected out in the results). However if graph $\mathcal{G}_{jvar}$ has a single path in it and only the join variable at the either end of the path is being projected out by the SELECT clause, we can generate the final results by following procedure. Make the SELECTed join variable as the root of $\mathcal{G}_{jvar}$, and make a single **UP** pass on $\mathcal{G}_{jvar}$. This ensures complete reduction of BitMats associated with that join variable and we can simply project out unique bindings of it. This will avoid the **DN** pass and improve the query performance further. Similar optimization can be applied depending on which variables in the query are projected out in the SELECT clause and the structure of $\mathcal{G}$.

5.6. Memory Requirements

In our current implementation, we load the BitMat associated with each triple pattern at the beginning of query processing and then *never* seek a disk access in the entire lifespan of the query. This necessitates that for a query having n triple patterns it needs $\sum_{i=0}^{n} size(BitMat_i)$ amount of memory at the beginning. This poses limitations for queries having triple patterns with all variable positions *(?x ?y ?z)*, as it is not feasible to load a BitMat for the all-variable triple pattern containing the entire dataset in memory. Although this seems like a large memory-overhead, due to our Enhanced Method of BitMat construction, and run length encoding applied on each BitMat, for typical queries $\sum_{i=1}^{n} size(BitMat_i)$ is small due to small size of the individual $BitMat_i$ (but we have given example of an exception in the Evaluation section). In future, all-variable triple patterns can be handled with dynamic construction of a BitMat associated with them after the first pass of pruning. Details of the procedure are omitted due to space constraints. Notably, in the set of the queries obtained from UniProt [23] and LUBM [24] datasets, none of the queries had a triple pattern having all variable positions. Two of the UniProt queries published by the authors of RDF-3X [6] have an all-variable triple pattern. But it is to be noted that these queries are not part of the queries published by UniProt [23].

With the Basic Method of BitMat construction, handling all-variable triple patterns is possible. In this case, we simply make a replica of the original BitMat for the BitMat associated with the all-variable triple pattern. Please note that due to this the Basic Method has restrictions on the amount of RDF data that can be loaded in memory.

Since each triple pattern has a separate BitMat associated with it, for highly selective queries, the memory requirement of the contemporary query processors, using 6-way indexes (e.g. RDF-3X), can be lesser than BitMat as they do not need to load entire indexes in memory to perform joins. But notably, BitMat's memory requirement remains linear in terms of the triples associated with the triple patterns, because it does not generate intermediate join results. Whereas for conventional query processors, it can increase polynomially for low-selectivity multi-join queries due to construction of intermediate results (e.g., $T_1(A, B, C) \bowtie_A T_2(A, D, E) = \{\langle a, b, c, d, e\rangle \in A \times B \times C \times D \times E \mid \langle a, b, c\rangle \in T_1 \; and \; \langle a, d, e\rangle \in T_2\})$.

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

We would like to point out some key differences between BitMat's algorithm and a typical bitmap index join. BitMat's structure is similar to the idea of compressed bitmap indexes [25, 26, 27]. Typically bitmap indexes are created on the columns having fewer distinct entries and those which are known to participate in more joins. But in an SQL join between multiple tables over *different* columns, after the first level of join, a query processor materializes results of the previous join to carry out the next join and the materialized intermediate tables do not always have bitmap indexes (unless join-indexes are precomputed based on heuristics). As opposed to that, BitMat's pruning and final result generation steps *always* use compressed BitMats, without materializing the intermediate join results. However, currently we handle only equijoins.

6. Evaluation

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

6.1. Choice of competitive RDF stores

We had a wide choice to select the systems for competitive evaluation due to the availability of numerous RDF triplestores. We experimented with Hexastore[7] [5], Jena-TDB [11], RDF-3X [6], and MonetDB [28].

Hexastore served as an in-memory store and the rest as persistent stores. For the Basic Method, we load the entire BitMat in memory before query processing. As the query is submitted, we run the query processing algorithm as described in Section 5.2 and 5.5. Due to the ability to load the entire data in-memory, we compared the Basic Method with Hexastore.

We chose RDF-3X (v0.3.3) and MonetDB (v5.14.2) – the persistent stores – for our evaluation of the Enhanced Method as they could load a large amount of RDF data, gave better performance than others (e.g. Jena-TDB).

Like BitMat, Hexastore and RDF-3X map strings/URIs in RDF data to integer IDs and mainly operate on these IDs, building the entire result of a query in the integer ID format. They convert the IDs to strings using their dictionary mapping just before outputting the results in a user readable format. Current BitMat system doesn't support a formal SPARQL query parser interface and the interface to output the results in the string format is still under preliminary development. Hence for a fair comparison, all the query times reported are only the core query processing times without counting query parsing or ID to string mapping after generating the results for all the systems.

We loaded MonetDB[8] by inserting the integer IDs generated out of BitMat dictionary mapping (ref. Section 3). Hence essentially all the MonetDB queries were performed on S, P, Os as integer IDs. We created separate predicate tables in MonetDB by inserting the respective triples by ordering on S-O values [28] and used these predicate tables in the query whenever there is a bound predicate in the triple pattern instead of the giant triple-table containing all the triples.

6.2. Choice of datasets and queries

Due to the limitation of main-memory and the amount of the RDF data that can be completely contained within it, we chose smaller datasets for evaluation of the Basic Method. To test the scalability of Basic and Enhanced method, we chose 4 datasets of increasing sizes. The dataset characteristics are given in Table 2. UniProt 0.2 million tripleset was extracted from the larger UniProt 845 million dataset. LUBM 6 million dataset was created over 50 universities using LUBM synthetic data generator [29], and LUBM 1.33 billion dataset was created over 10,000 universities.

Table 2: Dataset characteristics

Dataset	#Triples	#S	#P	#O
Uniprot 0.2m	199,912	30,007	55	45,754
LUBM 6m	6,656,560	1,083,817	18	806,980
UniProt 845m	845,074,885	147,524,984	95	128,321,926
LUBM 1.33b	1,335,081,176	217,206,845	18	161,413,042

Out of these we used the smaller two datasets to build the BitMat using Basic Method. Since Hexastore keeps all the data including the string to ID mappings in memory, the largest dataset it could load from our set was LUBM 6 million triples, hence we used the first two datasets to compare BitMat's performance with it. The other two datasets, UniProt 845 million and LUBM 1.33 billion, were used to build BitMats using the Enhanced Method and were evaluated by comparing with RDF-3X and MonetDB.

For the two UniProt datasets, we used some of the queries published by the UniProt dataset owners [23] and we used 6 out of the 8 queries published by RDF-3X in [6]. We modified two of the RDF-3X queries by removing some bound positions to reduce the selectivity of the triple patterns. For the LUBM datasets, OpenRDF has published a list of queries [24], but many of these queries are simple 2-triple pattern queries or they are quite similar to each other. Hence we chose only some of the representative queries out of this list. For more detailed description of the choice of queries please refer to our previously pub-

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

[8] MonetDB was compiled using "--enable-optimization" flag to enable highest possible optimization of MonetDB server.

lished work [16]. All the queries are listed in Appendix A.

6.3. Comparison

For the evaluation, we measured the following parameters:

1. Query execution time – This is an end-to-end time counted from the time of query submission until the output of final results in ID form – this is the "CPU (user + system) time". Since the machine that we used for query execution was not a shared machine, we ensured that except required kernel processes, no other miscellaneous user processes were running on the machine while calculating query times. We evaluated separate *cold* and *warm* cache times only for the Enhanced Method and hence only on the two larger datasets (UniProt 845 million and LUBM 1.33 billion triples). For cold cache, we dropped the file systems caches using `/bin/sync` and `echo 3 > /proc/sys/vm/drop_caches`. Query times were averaged over 10 consecutive runs as follows: (1) for cold cache times – we ran each query 10 times consecutively on a given system (BitMat, RDF-3X, or MonetDB) by dropping the file-system caches *every time before running the query.* E.g., for cold-cache performance, Q1 (UniProt 845 million) is run on BitMat 10 times, by dropping the file-system caches every time before running the query. Next Q1 (UniProt 845 million) is again run on RDF-3X by same procedure and then on MonetDB with same procedure (note that MonetDB server is restarted while doing so). This ensured that the file-system caches were empty while running the query each time, (2) for warm cache times – we ran the query first time to "warm up" file-system caches and then ran the same query 10 times consecutively. The procedure of running the query 10 times is same as described above for cold-cache method except that we did not drop file-system caches in between the consecutive runs. BitMat and RDF-3X do not run as servers. They are stand-alone applications which execute each query independently. MonetDB runs as a server, and hence for cold-cache times, we restarted MonetDB every time along with dropping file-system caches. But for warm-cache times, we neither restarted MonetDB nor dropped the file-system caches. Please note that for the Basic Method the original BitMat containing the entire RDF data always remains in memory once the BitMat process is started. Hence there is no cold-cache time reported for this structure.

2. Initial number of triples – the sum of triples matching all the triple pattern in the query.

3. The number of final results.

4. BitMat *filter* time – For the evaluation of the smaller datasets with the Basic Method, we measured another parameter – time taken for the *filter* operation in the initialization step (ref. Section 4.1). This helped us gain an insight into the overheads in query processing which are discussed further. In the Enhanced Method, for a triple pattern with at least one fixed position, the BitMat associated with it can be directly loaded from the disk as outlined in Section 5.1 without the need of filtering from the entire set of triples.

The evaluation is given in Tables 3 and 4. *Geometric mean** is the geometric mean of the query times excluding the ones on which one or the other RDF stores failed to complete the processing.

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

Table 3: Small RDF datasets – in-memory comparison (time in seconds, best times are boldfaced)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Geom. Mean
UniProt 0.2 million triples									
BitMat (Basic)	0.019	**0.046**	**0.22**	**0.022**	0.036	0.017			**0.037**
BitMat *filter* time	0.01	0.018	0.014	0.021	0.019	0.012			0.015
BitMat (Enhanced)	0.0063	0.033	0.090	0.0018	0.033	0.0023			0.011
Hexastore	**0.0011**	1.67	7.28	0.04	**0.025**	**0.007**			0.067
#Results	1	696	28600	137	28	1			
#Initial triples	6622	27470	35235	9046	41	2			
LUBM 6 million triples									
BitMat (Basic)	1.69	**1.00**	**10.31**	0.19	**0.81**	**11.05**	0.42	**0.55**	**1.27**
BitMat *filter* time	0.54	0.49	0.66	0.17	0.49	0.60	0.31	0.21	0.39
BitMat (Enhanced)	0.86	0.65	1.05	0.0063	0.67	1.11	0.016	1.44	0.29
Hexastore	**0.66**	3.35	75.69	**0.02**	1.16	104.04	**0.02**	26.99	1.96
#Results	130	5916	2199	146	1874	2188	125	54052	
#Initial triples	7302	17778	77160	292	5635	83066	280	108104	

6.3.1. Smaller Datasets (Basic Method)

Since Hexastore keeps all the data including their string to ID mappings in memory, it failed to load the LUBM 6 million dataset on our original evaluation platform having 4 GB of physical memory. Hence we used another machine with much higher physical memory (32 GB) to evaluate over LUBM 6 million dataset.

From the evaluation of the smaller datasets in comparison to Hexastore, it was apparent that for highly selective queries, producing very small number of results, Hexastore came out as a winner against the Basic Method (e.g., Q1, Q5, Q6 of UniProt 0.2 million dataset and Q1, Q4, Q7 of LUBM 6 million dataset). But for queries involving large number of initial triples, involving two or more join variables BitMat performed better (e.g., Q2, Q3, Q4 of UniProt 0.2 million dataset and Q2, Q3, Q6, Q8 of LUBM 6 million dataset). Specifically, Q2, Q3 and Q6 of LUBM-6m data, have *cyclic* dependency among their join variables (ref. Section 5.2). Hence the minimal triple set generation in the pruning step is not guarantied. For Q2 and Q6, BitMat did much better than Hexastore. It is our hypothesis that due to three join variables with a cyclic dependency between them, Hexastore could not exploit the advantage of "merge-joins" using 6-way indexes and had to resort to the conventional way of joins while performing later joins.

During the evaluation of smaller datasets, we also measured the time taken for the *filter* operation in the initialization step. This time is given in the evaluation Table 3. It was noted that in the query evaluation, the *filtering* time constituted a large part, especially for queries with highly selective triple patterns producing smaller number of results. For example, Q1, Q5, Q6 of UniProt-0.2m and Q4 and Q7 of LUBM-6m datasets are such queries. We also noted that most of the queries had at least one fixed position in their triple patterns. While evaluating Basic Method over larger datasets (up to 100 million triples) we noted that due to inherent limitation of the main memory size, we cannot load and process very large datasets using this method. We also compared the Basic Method to the Enhanced method (warm cache) – which eliminated the *filter* operation – and noted the significant improvement in query times (ref. Table 3). This further motivated our idea of using the Enhanced Method instead the Basic Method.

6.3.2. Larger Datasets (Enhanced Method)

Among the larger datasets, for Q1, Q2 of UniProt-845m and Q1, Q2, Q3 of LUBM-1.33b BitMat excelled over,

Table 4: Large RDF datasets (time in seconds, best times are boldfaced)

UniProt 845 million triples								
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Cold cache								
BitMat	**451.365**	**269.526**	173.324	9.396	78.35	1.34	9.33	13.06
MonetDB	548.21	303.2134	**124.3563**	9.63	97.28	11.28	9.91	15.93
RDF-3X	Aborted	525.105	244.58	**1.38**	**4.636**	**0.902**	**0.892**	**1.353**
Warm cache								
BitMat	**440.868**	**263.071**	168.6735	8.305	77.442	0.448	8.36	10.87
MonetDB	495.64	267.532	**113.818**	0.584	96.02	0.822	0.861	0.362
RDF-3X	Aborted	487.1815	226.050	**0.077**	**1.008**	**0.0064**	**0.003**	**0.0299**
#Results	160,198,689	90,981,843	50,192,929	0	179,316	0	0	19
#Initial triples	92,965,468	73,618,481	78,840,372	16,626,073	60,260,006	15,408,126	16,625,901	53,677,336

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

LUBM 1.33 billion triples								
	Q1	Q2	Q3	Q4	Q5	Q6	Geom. Mean	Geom. Mean* (without Q1)
Cold cache								
BitMat	**51.21**	**2.71**	**6.56**	2.45	0.503	3.81	**4.0285**	**2.4227**
MonetDB	109.35	27.17	455.23	34.12	18.89	14.6	48.3195	41.0377
RDF-3X	Aborted	34.868	2328.753	**0.588**	**0.425**	**1.129**	N/A	7.4474
Warm cache								
BitMat	**48.57**	**2.11**	**1.94**	0.686	0.27	2.85	**2.1719**	1.1666
MonetDB	96.65	6.56	398.46	3.209	0.566	0.542	7.9301	4.8094
RDF-3X	Aborted	29.033	2028.6855	**0.0024**	**0.0029**	**0.1814**	N/A	**0.5947**
#Results	2528	10,799,863	0	10	10	125		
#Initial triples	165,397,764	224,805,759	219,416,877	438,912,513	3,000,966	9,100,649		

both, RDF-3X and MonetDB. Notably, UniProt Q1, Q2 had a high number of initial triples and the join results were quite unselective. As was our initial conjecture, BitMat did much better on such queries. On the other hand, LUBM Q1 and Q3 were more complex queries having a high number of initial triples associated with the triple patterns, but the final number of results were quite small (2528 and 0 respectively). These queries too have cyclic dependency among their join variables, which does not guaranty minimal triple generation and can potentially prohibit conventional query processors from exploiting merge-joins.

For these queries BitMat was up to 3 orders of magnitude faster than RDF-3X and MonetDB due to its way of producing join results without materializing the intermediate join tables. RDF-3X aborted while executing UniProt Q1 and LUBM Q1 as the system ran out of its physical memory and swap space on the 4GB Dell PC. We executed the same queries on RDF-3X on a higher configuration server having 16 GB physical memory. Please note that the 16 GB machine was used only for the specific queries where RDF-3X aborted on the 4 GB Dell PC and we are reporting these times on 16 GB machine for investigating the query performance further. All other queries were run on same 4 GB Dell PC using the same procedure as outlined in Section 6.3. RDF-3X processed UniProt Q1 in 858.464 sec and was observed to consume ∼11 GB resident memory. For LUBM Q1, RDF-3X took 1613.178 sec and the peak memory consumption was ∼11 GB. For these queries BitMat took 448.169 and 50.70 sec, and consumed ∼2.6 GB and ∼3 GB, respectively, on the same server.

A "star join" query is the one where many triple patterns are joined on one variable, the query has only one join variable, and that variable is projected out in the final results. LUBM Q2, Q4, Q5 were star-join queries. For star-joins, BitMat worked much better, because our query processor doesn't need to load the BitMats of all the triple patterns in memory. It just loads the pre-computed row or column bitarrays of each BitMat associated with the triple pattern (ref. Section 3.2). The final result generation phase consists of just listing out the 1-bit positions from the bitwise AND of the loaded bitarrays (similar to the bitmap index joins). Notably, although Q2 has only two triple patterns in it, each has very low selectivity and the query generates a lot of results compared to query Q4 and Q5.

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

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

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

In view of these results, we would like to mention one specific LUBM query which turned out be an *outlier* (LUBM Q7 of LUBM-1.33b data). RDF-3X aborted due to the system running out of memory on the 4 GB Dell PC. BitMat took several hours to process this query, although the processor clock showed that the query spent only ∼200 seconds actually executing on the processor. MonetDB processed it in 449.048 sec. For further investigations, we evaluated this query on the server having 16 GB of memory and we found the following:

- BitMat finished processing this query in 139.94 sec on the 16 GB memory server. The peak resident memory consumption was reported to be 6.3 GB, and on an average the process consumed 5 GB of the resident memory. The BitMat associated with tp-node *'?x ub:takesCourse ?z'* was very large, ∼3.4 GB, having 288,017,530 triples (22% of the total triples) in it – largest among all the predicates. Thus, on the 4 GB Dell PC, BitMat process spent a lot of time in the kernel waiting for the pages to be allocated. MonetDB handled this situation well due to its better cache-memory management.

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

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

This query has 442,351,492 initial triples associated with it – largest among the listed LUBM-1.33b queries – and generates 439,994 results. We believe that with a "lazy loading" strategy along with "proactive cache management" BitMat would be able to handle these type of queries in a better manner in future.

6.3.3. Index Sizes and Construction Time

The on disk size of the BitMats created with the Basic Method for UniProt 0.2 million data is 1.6 MB and for LUBM 6 million data is 61 MB. The cumulative size of BitMats created with Enhanced Method are 6.3 MB and 202 MB respectively for the same datasets. We used an external Perl script to parse the raw triples and build string to ID dictionary mapping. The total time for parsing the UniProt 0.2 million and LUBM 6 million data was 6sec and 126sec respectively. The total time to construct Basic BitMats for UniProt 0.2 million and LUBM 6 million data was 0.1sec and 4sec respectively. Comparing the index sizes with the query performance (ref. Table 3), it was apparent that the Basic Method offers benefit of saving a lot of disk space, but suffers from the overheads of *filter* operation required while processing the queries. We observed that as the size of the data increases, the *encoding size* needed to store the entire row in the Basic Method of BitMat creation increases very fast prohibiting loading of very large datasets. Hence as the size of the data increases, it becomes beneficial to store the data using Enhanced Method and take the advantage of avoiding the *filter* operation which constitutes significant amount of query processing time.

The on disk cumulative size of all BitMats created with the Enhanced Method was 48 GB and 67 GB for UniProt 845 million and LUBM 1.33 billion data respectively and corresponding LZ77 compressed dictionary mappings were 3.2 GB and 1.8 GB. These BitMat sizes include the size of the meta-file too. But note that for any given query with n triple patterns, the runtime memory requirement is just $\sum_{i=1}^{n} size(BitMat_i)$; which is typically a much smaller fraction of the total datafile size. For RDF-3X and MonetDB the on-disk size of datafiles were 42 GB and 16 GB for UniProt-845m, and 70 GB and 25 GB for LUBM-1.33b respectively. Note that MonetDB's storage requirements do not include dictionary mapping, because we loaded RDF triples in integer ID representation in MonetDB.

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

7. Conclusion and Future Work

In this article, we have demonstrated a new method – BitMat – of storing a large number of RDF triples in a compressed format. This method can be implemented with *Basic* or *Enhanced* way of constructing the BitMats. Basic Method packs all the RDF triples in a single BitMat using much lesser disk space, but it suffers from the overheads of *filter* operation while executing queries, as well as the increase in the *encoding size* required to encode a compressed row in the BitMat for larger datasets. On the other hand, in case of the Enhanced Method, although the cumulative disk storage of the compressed BitMats is larger than the Basic Method, it avoids the overhead of *filter* operation.

As shown in our extensive evaluation, our novel join query processing algorithm delivers 2-3 orders of magnitude better performance than the state-of-the-art triplestores, specifically for queries with non-selective triple patterns. This is due to our way of *not* materializing the intermediate join results and keeping the data in the compressed BitMat form in memory. This ensures a smaller memory footprint even for queries involving a large number of triples. For queries with highly selective triple patterns, the contemporary triplestores, e.g., RDF-3X and MonetDB work better, because of their optimization techniques which are tuned for highly selective queries.

Presently we process only equijoin queries, but BitMat's structure and pruning algorithms can as well be integrated with a conventional query processor to achieve fast pruning of RDF triples for the other class of SPARQL queries, e.g., OPTIONAL, UNION, FILTER, which can involve underlying equijoin operation. With dynamic BitMat construction, we can handle all-variable triple patterns. A BitMat can also be thought as a compressed adjacency matrix between subject and object nodes, classified as per predicates. Therefor this structure can also be used for performing graph operations on the RDF data where the conventional in-memory graph data structures can be prohibitive as the size of the graph increases, and this is the focus of our future work.

Acknowledgments

We would like to thank Dr. Jagannathan Srinivasan of Oracle Corp. for his invaluable input in the initial phase of the work. This research is continuing through participation in the International Technology Alliance sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence, and it was supported in parts by NSF grant EMT-0829835 and NIH grant 1R01EB0080161-01A1.

References

[1] D. J. Abadi, S. R. Madden, M. C. Ferreira, Integrating Compression and Execution in Column Oriented Database Systems, in: SIGMOD, 2006.

[2] D. J. Abadi, D. S. Myers, D. J. DeWitt, S. R. Madden, Materialization Strategies in a Column-Oriented DBMS, in: International Conference on Data Engineering, 2007.

[3] R. Cyganiak, A Relational Algebra for SPARQL, Tech. Rep. HPL-2005-170, HP Laboratories (2005).

[4] R. Ramakrishnan, J. Gehrke, Database Management Systems, McGraw-Hill Science, 2002.

[5] C. Weiss, P. Karras, A. Bernstein, Hexastore: Sextuple Indexing for Semantic Web Data Management, in: Proceedings of Very Large Data Bases, 2008.

[6] T. Neumann, G. Weikum, Scalable join processing on very large RDF graphs, in: SIGMOD, 2009.

[7] 4store, `http://4store.org/`.

[8] BigOWLIM, `http://www.ontotext.com/owlim/index.html`.

[9] Allegrograph, `http://agraph.franz.com/allegrograph/`.

[10] MonetDB, `http://monetdb.cwi.nl/`.

[11] Jena TDB, `http://jena.sourceforge.net/TDB/`.

[12] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable Semantic Web Data Management using Vertical Partitioning, in: Proceedings of Very Large Data Bases, 2007.

[13] T. Neumann, G. Weikum, RDF3X: a RISC style Engine for RDF, in: Proceedings of Very Large Data Bases, 2008.

[14] P. A. Bernstein, N. Goodman, Power of natural semijoins, SIAM Journal of Computing 10 (4) (1981) 751–771.

[15] P. A. Bernstein, D.-M. W. Chiu, Using semi-joins to solve relational queries, Journal of the ACM 28 (1) (1981) 25–40.

[16] M. Atre, V. Chaoji, M. J. Zaki, J. A. Hendler, Matrix "Bit"loaded: A Scalable Lightweight Join Query Processor for RDF Data, in: WWW, 2010.

[17] M. Atre, J. A. Hendler, BitMat: A Main-memory Bit-Matrix of RDF Triples, in: Scalable Semantic Web Knowledge Base Systems workshop at International Semantic Web Conference, 2009.

[18] M. Atre, J. Srinivasan, J. Hendler, BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries, in: Poster and Demo track, International Semantic Web Conference, Karlsruhe, Germany, 2008.

[19] A. Matono, S. M. Pahlevi, I. Kojima, RDFCube: A P2P-based Three-dimensional Index for Structural Joins on Distributed Triple Stores, in: Workshop on Databases, Information Systems and Peer-to-Peer Computing at the conference on Very Large Data Bases, 2006.

[20] M. Cai, M. Frank, RDFPeers: A Scalable Distributed RDF Repository based on a Structured Peer-to-Peer Network, in: WWW, 2004.

[21] Line graph, `http://en.wikipedia.org/wiki/Line_graph`.

[22] N. Mamoulis, D. Papadias, Multiway spatial joins, ACM Trans. Database Syst. 26 (4) (2001) 424–475.

[23] UniProt RDF Queries, `http://dev.isb-sib.ch/projects/expasy4j-webng/query.html#examples`.

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

[25] P. O'Neil, G. Graefe, Multi-Table Joins Through Bitmapped Join Indices, SIGMOD Record 24 (3) (1995) 8–11.

[26] T. Johnson, Performance Measurements of Compressed Bitmap Indices, in: Proceedings of Very Large Data Bases, 1999.

[27] S. Sarawagi, Indexing OLAP data, Data Engineering Bulletin 20 (1) (1997) 36–43.

[28] L. Sidirourgos, R. Goncalves, M. Kersten, et al., Column-store Support for RDF Data Management: not all swans are white, in: Proceedings of Very Large Data Bases, 2008.

[29] LUBM, `http://swat.cse.lehigh.edu/projects/lubm/`.

Appendix A. Queries

Appendix A.1. UniProt 0.2 million

PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX uni: <http://purl.uniprot.org/core/> PREFIX uni2: <http://purl.uniprot.org/>

Q1: SELECT ?protein ?name ?gene WHERE { ?protein rdf:type :Protein . ?protein uni:name ?name . ?protein uni:gene ?gene . ?gene uni:name “CRB” .}

Q2: SELECT ?protein ?annotation ?range ?begin ?end WHERE { ?protein rdf:type :Protein . ?protein uni:annotation ?annotation . ?annotation rdf:type :Transmembrane_Annotation . ?annotation uni:range ?range . ?range uni:begin ?begin . ?range uni:end ?end . }

Q3: SELECT ?protein ?author ?title WHERE { ?protein rdf:type uni:Protein . ?protein uni:modified ?modified . ?protein uni:citation ?citation . ?citation uni:author ?author . ?citation uni:title ?title .}

Q4: SELECT ?protein ?related WHERE { ?protein rdf:type uni:Protein . ?protein ?p uni2:keywords/482 . ?protein rdfs:seeAlso ?related .}

Q5: SELECT ?gene ?name ?text WHERE { ?annotation rdf:type uni:Disease_Annotation . ?annotation rdfs:comment ?text . ?protein rdf:type uni:Protein . ?protein uni:gene ?gene . ?gene uni:name ?name . ?protein uni:organism uni2:taxonomy:9606 . ?protein :annotation ?annotation .}

Q6: SELECT ?protein ?x ?s WHERE { ?protein rdf:type :Protein . ?protein uni:organism uni2:taxonomy:287 . ?protein uni:sequence ?s . ?s rdf:value ?x .}

Appendix A.2. LUBM 6 million

PREFIX ub: <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#>

Q1: SELECT ?x ?y ?z WHERE { ?z ub:subOrganizationOf ?y . ?y rdf:type ub:University . ?z rdf:type ub:Department . ?x ub:memberOf ?z . ?x rdf:type ub:GraduateStudent . ?x ub:undergraduateDegreeFrom ?y . }

Q2: SELECT ?x ?y ?z WHERE { ?x rdf:type ub:UndergraduateStudent . ?y rdf:type ub:Department . ?x ub:memberOf ?y . ?y ub:subOrganizationOf <http://www.University0.edu> . ?x ub:emailAddress ?z . }

Q3: SELECT ?x ?y ?z WHERE { ?y ub:teacherOf ?z . ?y rdf:type ub:FullProfessor . ?z rdf:type ub:Course . ?x ub:advisor ?y . ?x rdf:type ub:UndergraduateStudent . ?x ub:takesCourse ?z}

Q4: SELECT ?x WHERE { ?x rdf:type ub:GraduateStudent . ?x ub:memberOf <http://www.Department0.University0.edu> . }

Q5: SELECT ?x ?y ?z WHERE { ?x rdf:type ub:GraduateStudent . ?y rdf:type ub:Department . ?x ub:memberOf ?y . ?y ub:subOrganizationOf <http://www.University0.edu> . ?x ub:emailAddress ?z . }

Q6: SELECT ?x ?y ?z WHERE { ?x rdf:type ub:UndergraduateStudent . ?y rdf:type ub:AssistantProfessor . ?z rdf:type ub:Course . ?x ub:advisor ?y . ?y ub:teacherOf ?z . ?x ub:takesCourse ?z . }

Q7: SELECT ?x ?y WHERE { ?x rdf:type ub:FullProfessor . ?y rdf:type ub:Department . ?x ub:worksFor ?y . ?y ub:subOrganizationOf <http://www.University0.edu> . }

Q8: SELECT ?x ?y WHERE { ?x rdf:type ub:Course . ?x ub:name ?y .}

Appendix A.3. UniProt 845 million

Q1: SELECT ?modified ?author ?citation ?title ?protein WHERE { ?protein rdf:type uni:Protein . ?protein uni:/modified ?modified . ?protein uni:citation ?citation . ?citation uni:author ?author . ?citation uni:title ?title .}

Q2: SELECT ?a ?vo ?b ?ab ?x ?z ?p WHERE { ?a uni:encodedBy ?vo . ?a schema:seeAlso ?x . ?b uni:sequence ?z . ?b uni:replaces ?p . ?b rdf:type uni:Protein . ?a uni:replaces ?ab . ?ab uni:repla-cedBy ?b . }

Q3: SELECT ?a ?x ?vo ?b ?ab ?z ?p WHERE { ?a schema:seeAlso ?x . ?a uni:encodedBy ?vo . ?b uni:sequence ?z . ?b uni:replaces ?p . ?b uni:modified "2008-07-22" . ?b rdf:type uni:Protein . ?a uni:replaces ?ab . ?ab uni:replacedBy ?b . }

Q4: SELECT ?name ?gene ?protein WHERE { ?protein rdf:type uni:Protein . ?protein uni:encodedBy ?gene . ?gene uni:name "hup" . ?protein uni:name ?name }

Q5: Same as Q4 in UniProt 0.2 million data

Q6: SELECT ?p2 ?interaction ?p1 WHERE { ?p1 uni:enzyme uni2:enzyme/2.7.7.- . ?p1 rdf:type uni:Protein . ?interaction uni:participant ?p1 . ?interaction rdf:type uni:Interaction . ?interaction uni:participant ?p2 . ?p2 rdf:type uni:Protein . ?p2 uni:enzyme uni2:enzyme/3.1.3.16 .}

Q7-Q10: Same as Q1, Q3, Q6, Q8 in [6] respectively.

Q11: Same as Q6 in UniProt 0.2 million data

Q12-Q13: Same as Q5 and Q7 respectively in [6],

Note: We project all the variables in the queries Q7-Q10, Q12, Q13 above.

Appendix A.4. LUBM 1.33 billion

PREFIX ub: <http://www.lehigh.edu/˜zhp2/2004/0401/univ-bench.owl#>

Q1: Same as Q1 in LUBM 6 million data

Q2: Same as Q2 in LUBM 6 million data

Q3: SELECT ?x ?y ?z WHERE { ?x rdf:type ub:Undergraduate-Student. ?y rdf:type ub:University . ?z rdf:type ub:Department . ?x ub:memberOf ?z . ?z ub:subOrganizationOf ?y . ?x ub:under-graduateDegreeFrom ?y . }

Q4: SELECT ?x WHERE { ?x ub:worksFor <http://www.-Department0.University0.edu> . ?x rdf:type ub:FullProfessor . ?x ub:name ?y1 . ?x ub:emailAddress ?y2 . ?x ub:telephone ?y3 . }

Q5: SELECT ?x WHERE { ?x ub:subOrganizationOf <http-://www.Department0.University0.edu> . ?x rdf:type ub:Research-Group}

Q6: Same as Q7 in LUBM 6 million data

Q7: Same as Q3 in LUBM 6 million data