BitMat: An In-core RDF Graph Store for Join Query Processing

Technical Report
August 1, 2009

Medha Atre, Vineet Chaoji, Jesse Weaver, and Gregory Todd Williams
Department of Computer Science
Rensselaer Polytechnic Institute, Troy NY, USA
{atrem, chaojv, weavej3, willig4}@cs.rpi.edu

ABSTRACT

With the growing size of RDF data sources, the need for a compact representation providing efficient query interface has become compelling. In this paper, we introduce BitMat, a main memory based compressed bit-matrix structure. The key aspects of BitMat are as follows: i) its RDF graph representation is very compact compared to the conventional disk-based and existing main-memory RDF stores, and ii) SPARQL Basic Graph Pattern (BGP) queries are processed using a novel method employing initial pruning technique, followed by a subgraph matching algorithm on an BitMat, without uncompressing it at any point. This enables processing queries on very large RDF graphs completely in-memory. We present the key concepts of the BitMat structure and our BGP query processing algorithm. Finally we compare memory utilization and query performance of BitMat with contemporary triplestores for up to 51 million triples.

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 RDF format. RDF is already extensively used for representing information in the field of bioinformatics. Recently, search giant like Google too announced supporting microformats and RDFa [1]. To meet the storage and querying needs of this large scale RDF data, numerous systems are already being developed. These systems can be broadly classified as persistent disk-based or memory-based. In this work we describe an *in-core system*, a term commonly used for memory-based systems[3]. In the past, scarce memory resources have restricted system designers to adopt complex disk-based methods. With the advances in hardware technologies, systems with large memory and computing power are widely available. As a result, memory-based systems, including RDF triplestores, are becoming more desirable.

Even with the availability of large memory systems, in-core RDF stores aimed at specifically exploiting RDF structure and querying method have not been developed. Majority of the contemporary frameworks which support in-memory RDF stores, such as Jena [12], Hexastore [28], Redland [2] are implemented as a straightforward extension of relational persistent storage systems. These systems build auxiliary indexes on the RDF data which are then used to improve the overall query performance. Large RDF graphs[4] of the order of few million edges are common these days. Examples include RDF graphs of Uniprot [26], Barton [6], Wikipedia [29]. Due to the conventional representation methods adopted for RDF, the contemporary memory based methods fail to load such large graphs (as shown in the *Evaluation* section). In contrast, we propose a simple compressed representation for RDF data. The compressed representation, called *BitMat*, serves as the primary storage without any auxiliary indexes. Queries can be evaluated on this native compact representation without uncompressing it at any point. The compact representation is beneficial in the following application scenarios: (1) Mobile devices are becoming more sophisticated with their growing penetration in everyday life. Efforts to accommodate large amounts of data in these devices are ongoing [27]. In such a scenario, a data representation that enables storing and querying larger volume of data in the same space is desirable. (2) In contrast, consider a large cluster computer with 64 thousand nodes, each with a small private memory (1GB). The BlueGene/L [10] is a good example of such a system. Even though data can be partitioned to run concurrently on such a system, the I/O cost can be a dominating factor. BitMat's query processing algorithms provide a way to prune the set of RDF triples before generating final results. Hence a considerable reduction in the I/O cost can be achieved by transferring these intermediate triples encoded in an BitMat. The final results can be generated at the end nodes from the encoded representation. We claim that BitMat would excel in the above two scenarios. To the best of our knowledge, this work presents the first compact representation for RDF graphs. Our contributions in this work are:

1. A compressed data structure for RDF to increase the size of the data that can fit entirely in memory.
2. A novel algorithm that performs efficient pruning of the triples during the first phase of BGP query execution. In the second phase, it performs efficient subgraph matching to obtain the final results.
3. All the procedures and algorithms are implemented to work on compressed data without uncompressing it

[1]http://www.w3.org/TR/rdf-syntax-grammar/
[2]http://www.w3.org/TR/rdf-sparql-query/
[3]The two terms would be used interchangeably throughout the paper.
[4]RDF data can be represented as a graph structure.

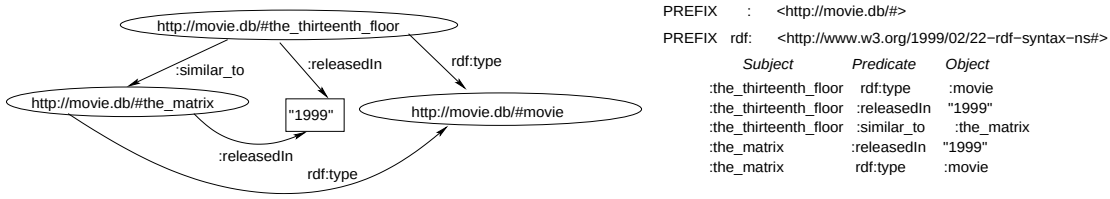

Figure 1: A sample RDF Graph

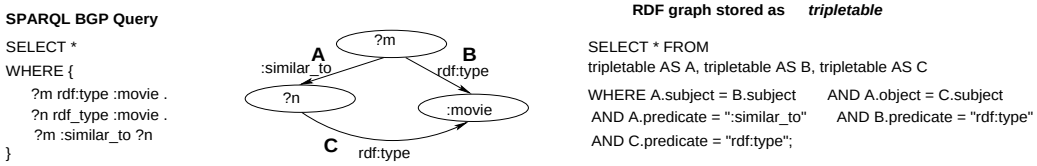

Figure 2: BGP Query Graph

any time. This controls size of the intermediate memory required for query execution.

4. An exhaustive set of experiments on popular RDF datasets, showing query performance comparable with other contemporary systems, and memory footprint much smaller than others.

The work presented in this paper is a considerable extension of our preliminary work introduced in [4].

The rest of the paper is organized as follows. Section 2 gives a brief background on RDF data processing and its connection with the traditional database query processing. Section 3 introduces BitMat and its representation. Section 4 outlines the BGP querying algorithm based on the BitMat structure. Experimental evaluation is covered in Section 5. Section 6 covers other related work and Section 7 concludes the paper with avenues for future exploration.

2. PRELIMINARIES

An RDF graph is a labeled directed graph like the one shown in Figure 1. Each edge in the graph can be represented by an *RDF triple*. An *RDF triple* represents the relationship between a *subject* and an *object* denoted by a *predicate*. For instance, the preposition *"The Matrix was released in 1999"* can be represented by the triple (*:the_matrix :releasedIn "1999"*). Figure 1 shows the list of triples for the example RDF graph.

A SPARQL *Basic Graph Pattern (BGP)* query, also known as a *join query* in the database terminology, is shown in Figure 2. A basic graph pattern contains a set of *triple patterns*. Triple patterns are like RDF triples except that each subject, predicate, or object entry can either be a variable or a fixed value.

All *unbound variables* in the triple pattern are preceded with a "?". For instance, nodes labeled *?m* and *?n* are unbound variables in Figure 2. The BGP matching problem is that of finding all instances of the query pattern in the RDF graph. The result of this BGP query generates bindings for the variables in the triple patterns. For instance, the query pattern in Figure 1 produces a single matching subgraph from the RDF graph by binding $?m \Rightarrow$ *:the_thirteenth_floor* and $?n \Rightarrow$ *:the_matrix*.

There exists a direct mapping between BGP queries and SQL style queries. For SQL queries the RDF graph can be conceptualized as a 3-column (subject, predicate, object) table in a relational database. An example SQL query is shown in Figure 2. Some contemporary methods use this approach for storing the RDF graph. Such systems build additional auxiliary indexes for efficient query processing. For example, Virtuoso [9] creates bitmap indexes in addition to the PGOS index. Some others depend on indexing or partitioning RDF data – Jena-TDB [3], vertical partitioning method using C-Store [5], and Hexastore [28]. Vertical partitioning method using C-Store, partitions RDF triples based on distinct predicates aiming at *predicate-bound* BGP queries. But this approach suffers when the number of predicates grow very high. Some others like Hexastore [28] create 6-way indexes to store the RDF graph. Hexastore also refers to an extensive list of other systems employing various auxiliary indexing methods for RDF data. But these representations are very bulky and hence they can neither hold large RDF graphs nor can they perform queries on them entirely in memory. For example, the five fold increase in the memory consumption with Hexastore's 6-way indexes restricts it from operating on massive RDF graphs in-memory.

In contrast to these approaches, our work proposes to build a 3-dimensional compressed *bit-cube* of RDF triples. The proposed structure, *BitMat*, provides a compact and complete representation for an RDF graph. The following section provides the details of building an BitMat.

3. BITMAT CONCEPTS

In an RDF graph let V_s, V_p and V_o denote the set of distinct subjects, predicates, and objects respectively. A bit-cube is created with the three dimensions as subjects (S), predicates (P), and objects (O). Each bit in this cube represents a unique combination of an element from V_s, V_p and V_o. A value of 1 at the bit location (s_i, p_j, o_k) denotes the presence of that triple in the graph. To represent this bit-cube as a 2-dimensional matrix, we slice it along the predicate dimension which gives us $|V_p|$ matrices of size $(V_s \times V_o)$. The $|V_p|$ matrices can be aligned either along the subject or object dimension. The resulting subject aligned matrix of size $(|V_s| \times |V_o||V_p|)$ is referred to as the *Subject BitMat*. The corresponding object aligned matrix is called the *Object*

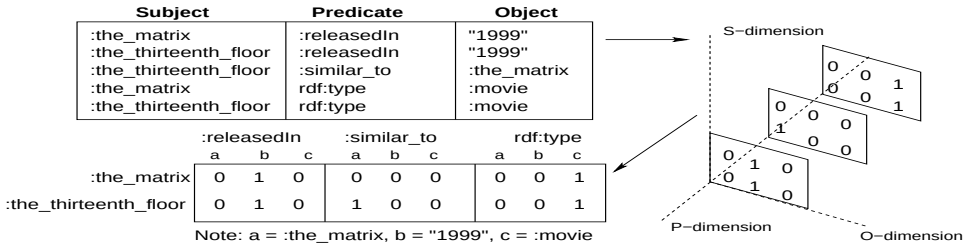

Figure 3: A Subject BitMat of sample RDF data

BitMat. Note that the bit-cube can also be sliced along the subject or the object dimension. Hence altogether there are 6 ways of flattening a bit-cube into an BitMat. Our initial experiments have shown that slicing along the predicate dimension generates the most favorable structure in terms of compactness and ease of BGP query processing. As a result, without loss of generality, we use the Subject BitMat in all our experiments. Figure 3 represents a Subject BitMat for the RDF graph shown in Figure 1. Each vertical column under a distinct predicate represents a distinct object.

To construct an BitMat from a given RDF graph, 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 identifier, as follows:

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

The common subject-object identifier assignment facilitates BGP queries wherein the unbound variable in the subject location is same as the unbound variable in the object position of another triple pattern (e.g. *?n* in the query graph in Figure 2). The three identifiers in each dimension uniquely identify a cell in the bit-cube. Due to the sparsity of the RDF graph, this bit-cube can be compressed efficiently. We compress an BitMat with the D-gap compression scheme [8]. D-gap compression is a form of *Run Length Encoding* used for sparse bit-vectors. Due to this compression we can completely load very large RDF graphs as BitMats in memory. For instance, an uncompressed bitcube holding 51 million Barton triples will take 6672TB of space, whereas its compressed BitMat takes around 0.6GB. We build the compressed BitMat directly using the above V_s, V_p, and V_o set mappings without building the uncompressed BitMat first.

Once the compressed BitMat is constructed, it can be stored on the disk as a raw byte-file and can be directly read into memory from the next time onwards. A compressed BitMat along with the mappings of the four sets V_{so}, $V_s - V_{so}$, V_p, and $V_o - V_{so}$ gives a complete representation of the original RDF graph.

3.1 BitMat Operations

A BGP query graph can be expressed as a *conjunctive triple pattern*. The graph shown in Figure 2 can be expressed as a conjunctive pattern of *(?m :similar_to ?n)* $\wedge$ *(?m rdf:type :movie)* $\wedge$ *(?n rdf:type :movie)*. As mentioned earlier, the goal of a BGP query processor is to find variable bindings which satisfy this pattern. A naïve solution could be to use a subgraph matching algorithm [25] to match the given BGP query graph against an entire RDF graph. This process is very inefficient with a branch and bound strategy on the whole RDF graph, especially for RDF graphs having few millions edges.

Hence we apply a pruning technique to reduce the number of RDF triples based on triple patterns in the query. We would like to point out though that our pruning algorithm is not same as a database join on tables using bitmap indexes [22]. This is elaborated further in Section 4.

The pruning algorithm is based on three primitive operations on an BitMat. They are as follows:
(1) **Filter:** Filter operation is represented as *'filter(BitMat, TriplePattern) returns BitMat'*. It takes an input BitMat and returns a new BitMat which contains only triples that match the *TriplePattern*. For the query graph pattern in Figure 2, *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.
(2) **Fold:** As the name suggests, the fold operation represented as *'fold(BitMat, RetainDimension) returns bitArray'* folds the input BitMat along the two dimensions other than the *RetainDimension*. For example, if *RetainDimension* is set to *'object'*, then BitMat is folded along the predicate and subject dimensions resulting into a single bitarray (see Figure 4). Intuitively, a bit set to 1 in this array indicates the presence of *at least* one triple with the object corresponding to that position in the given BitMat.
(3) **Unfold:** Specified as *'unfold(BitMat, MaskBitArray, RetainDimension)'*, unfolds the *MaskBitArray* on the *BitMat*. Intuitively, in the *unfold* operation, for every bit set to *0* in the *MaskBitArray* all the bits corresponding to that position of the *RetainDimension* in the *BitMat* are cleared. For example, *unfold(BitMat, '101', 'predicate')* would result in clearing all the bits in the second (S,O) matrix corresponding to *:similar_to* predicate in Figure 3.

Please note that *filter*, *fold*, and *unfold* operations are implemented to operate directly on a compressed BitMat without uncompressing it at any point. Since standard techniques are used for performing bitwise operations on compressed bitmaps, those details are omitted here.

4. BGP MATCHING ALGORITHM

A SPARQL BGP query evaluation over a 3-column ta-

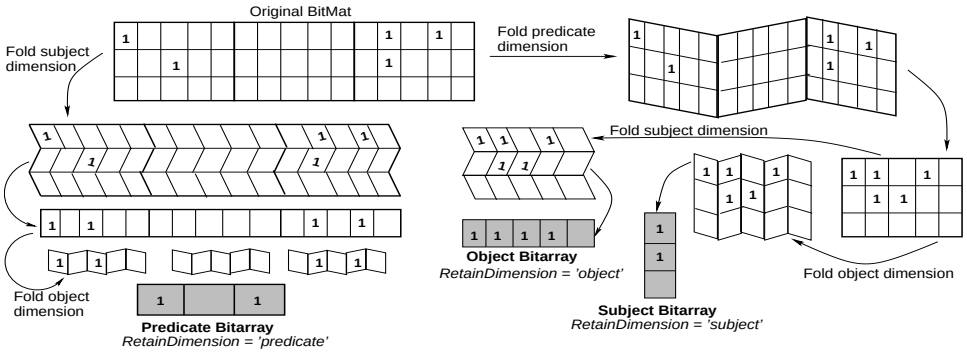

Figure 4: Conceptual view of folding an BitMat for a *RetainDimension*

ble makes use of relational join as shown in the SQL style query in Figure 2. Such an approach (a) needs creation of auxiliary indexes to facilitate query performance; which increases the memory requirement, and (b) a relational join algorithm can evaluate more than one join by creating intermediate join tables that retain the results of the previous joins. This is also called as *materialization* of intermediate results. Materialization of an intermediate join table needs larger memory resources. This mainly hinders processing very large RDF graphs entirely in memory[5]. In context of the BGP query in Figure 2, a relational join algorithm first evaluates join condition $A.subject = B.subject$ and creates an intermediate join table holding results of this join. Then it uses this table's $A.object$ column to evaluate join condition $A.object = C.subject$.

To overcome the above shortcomings, we propose a two phase algorithm for BGP query processing. In the first phase, RDF triples are pruned based on a novel pruning algorithm. In the second phase, the final results are generated from the pruned triples. Again, at no point are the triples uncompressed or intermediate results materialized. To explain the pruning technique, we introduce some definitions and properties.

Definition 1: An RDF sub-graph which matches the query pattern, maps each triple pattern to an RDF triple in the original graph. This mapping also *binds* the variables in the query graph to the corresponding values in the mapped triples. For the query graph shown in Figure 2, the result subgraph maps
(?m :similar_to ?n) ⇒ *(:the_thirteenth_floor :similar_to :the_matrix)*,
(?m rdf:type :movie) ⇒ *(:the_thirteenth_floor rdf:type :movie)*
(?n rdf:type :movie) ⇒ *(:the_matrix rdf:type :movie)* and generates bindings of *?m* ⇒ *:the_thirteenth_floor* and *?n* ⇒ *:the_matrix*.

Definition 2: A variable in a BGP query graph is called a *join variable* if it appears in more than one triple pattern. For instance, variables *?m* and *?n* are both join variables in Figure 2.

PROPERTY 1. *A join variable ?j common between two triple patterns, enforces that the RDF triples mapped to those triple patterns share the same value for ?j.*

The goal of pruning phase is to reduce the set of RDF triples to which each triple pattern can be mapped. This step is explained in the next section.

[5]We have considered the most common scenario of SQL query execution, but there can be various other strategies depending on the underlying data and characteristics of the queries.

4.1 Step 1 – Pruning the RDF Triples

The first step starts with *Initialization* (Algorithm 1). This step performs an initial pruning of RDF triples based on *fixed values* in the triple patterns in a query. Recall that a triple pattern is like an RDF triple which can have variables or fixed values in the S, P, or O positions. *Initialization* creates a separate BitMat associated with each triple pattern. This BitMat contains only triples satisfying that triple pattern (see Figure 5). For example, $BitMat_1$ associated with *(?m rdf:type :movie)* contains only triples having *rdf:type* as predicate and *:movie* as object value.

Algorithm 1 Initialization

```
1: Let BM be the BitMat for the original RDF graph
2: for each tp-node T in G do
3:     BM_T = filter(BM, T)
4: end for
```

Next, we make use of Property 1 to further prune RDF triples. For this purpose, we build a *constraint graph*[6] $\mathcal{G}$ from the query graph. This constraint graph captures the relationship between join variables which are responsible for enforcing constraints on the triples to which triple patterns can be mapped.

The constraint graph is constructed as follows:

1. Each triple pattern in the BGP query is denoted by a *tp-node* in $\mathcal{G}$. Hence forth we use the terms "tp-node" and "triple pattern" interchangeably. Additionally, a node is introduced in $\mathcal{G}$ for each join variable (denoted as *jvar-node*).

2. An undirected, unlabeled edge between a jvar-node and a tp-node exists in $\mathcal{G}$ if the join variable in the jvar-node appears in the triple pattern in the tp-node.

3. An edge exists between two jvar-nodes if the two join variables appear in the same triple pattern. This is again an undirected, unlabeled edge.

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

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

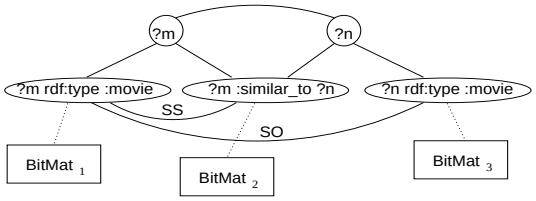

Figure 5: Constraint graph $\mathcal{G}$ for BGP query in Figure 2

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 dependency and *SO* denotes subject-object dependency.

Figure 5 shows the constraint graph for the query shown in Figure 2.

PROPERTY 2. *Suppose a jvar-node* ***?m*** *appearing in the S position in a triple pattern cannot be bound to a value* ***V****, then all the triples having* ***V*** *in the S position should be eliminated from the BitMat associated with that triple pattern. Similarly, this argument applies to P and O positions too.*

PROPERTY 3. *By the construction of graph $\mathcal{G}$, two adjacent jvar-nodes ?m and ?n have one or more common tp-nodes. Hence any updates performed on the BitMats associated with these tp-nodes change the variable bindings for ?m and ?n.*

An edge between two jvar-nodes in $\mathcal{G}$ represents a *co-dependency* between their binding values. Properties 2 and 3 establish that a change in the variable bindings of one jvar-node can change the variable bindings of its adjacent jvar-nodes. We make use of this fact to iteratively resolve the co-dependencies between jvar-node bindings and achieve further pruning of RDF triples.

Let $\mathcal{G}'$ be an induced sub-graph of $\mathcal{G}$ with only jvar-nodes. $\mathcal{G}'$ can be cyclic. Let us first consider the case when $\mathcal{G}'$ is acyclic. Since an acyclic, undirected graph is a tree, we create a topological ordering of all the nodes in this *tree*. Each parent node's ID is lesser than all of its successor nodes. We sort the node-IDs in decreasing order and call *prune_for_jvar* on each jvar-node in that order (Lines 1–4 in Algorithm 2). This ensures that *prune_for_jvar* is called first on all the successor nodes before being called on their ancestor nodes. *prune_for_jvar(J)* computes the *intersection* of all bindings for jvar-node J from the BitMats of tp-nodes in which J occurs (Lines 2–5 in Algorithm 3). For example, *prune_for_jvar(?m)* computes intersection of bindings for *?m* from $BitMat_1$ and $BitMat_2$. Recall that each tp-node's BitMat is generated during *Initialization*. The *intersection* of all the bindings captured in $MaskBitArr_J$ are then relayed back on the tp-nodes' BitMats to enforce the join variable constraints (Lines 6–9 in Algorithm 3). The *fold* operation on a tp-node's BitMat collects the bindings for J, whereas the *unfold* operation on an BitMat removes all the triples that should be dropped as a result of the above intersection. *getDimension* returns the position of J in a triple pattern. For instance, *getDimension(?n, (?m :similar_to ?n))* returns *object*.

One such pass over all the jvar-nodes ensures that the constraints are propagated to the adjacent jvar-nodes. For a complete propagation of constraints, we traverse jvar-nodes second time by following the reverse order of the first pass (Lines 5–8 in Algorithm 2). 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. Thus Algorithm(2) ensures, when $\mathcal{G}'$ is acyclic, a minimal triple set for each tp-node on termination (a formal proof of minimal triple set generation with this algorithm is given in Appendix A). This completes the pruning step.

Algorithm 2 Pruning Step

```
1: queue q = topological_sort(V(G'))
2: for each J in q do
3:     prune_for_jvar(J)
4: end for
5: queue q_rev = q.reverse()
6: for each K in q_rev do
7:     prune_for_jvar(K)
8: end for
```

Algorithm 3 prune_for_jvar(jvar-node J)

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

When graph $\mathcal{G}'$ has a cyclic dependency between jvar-nodes, there can be more than one path to reach a node from the root of the tree. Hence two passes over the jvar-nodes do not guaranty a minimal triple set generation. In such a case, the above algorithm has to be repeated iteratively. Constraints have to be propagated and resolved in a cyclic fashion until all are satisfied. These iterations terminate if the intersection of each jvar-node bindings, generated by the BitMats of its tp-nodes, is unaltered between two subsequent iterations. This is signified by the change in each $MaskBitArr_J$ – if the number of bits *set* in $MaskBitArr_J$ are the same as in the previous round.

Since a bitwise AND flips a 1 to 0 but never flips a 0 to 1, every pass over jvar-nodes in the graph $\mathcal{G}'$ reduces the number of triples in the BitMats monotonically. Hence it can be proved that these cyclic iterations converge (in the worst case when all the triples are eliminated from all the BitMats). But since the number of iterations in such a case depend on the specific query and dataset, we practically refrain from taking this approach. Instead, we execute Algorithm 2 and generate a reduced set of triples in each BitMat, which is not guaranteed to be minimal. LUBM queries 1, 3, and 6 in our set have such cyclic dependencies.

A similar approach needs to be taken if the original constraint graph $\mathcal{G}$ contains two or more tp-nodes sharing more than one jvar-nodes among them. When the query graph has such a condition, two adjacent jvar-nodes have more than one BitMat shared between them. This breaks the assumption of minimal triple set generation in the BitMats associated with jvar-nodes. None of the queries in our set had such a condition.

It is to be noted that in spite of these conditions, our

experiments show that above pruning algorithm achieves a significant reduction in the set of RDF triples. The *Evaluation* section provides few examples.

Complexity Analysis: We perform *fold* and *unfold* operations for each tp-node $\mathcal{T}$ associated with each jvar-node J. Hence the complexity of Algorithm(2) in terms of *fold* and *unfold* operation is given by $(2*\sum_J \sum_{\mathcal{T}\in\{\mathcal{T}|\exists E_{J\mathcal{T}}\}} O(\text{fold})_{\mathcal{T}} + O(\text{unfold})_{\mathcal{T}})$. $O(\text{fold})_{\mathcal{T}}$ and $O(\text{unfold})_{\mathcal{T}}$ depend on the selectivity[7] of the triple pattern $\mathcal{T}$ and selectivity of $MaskBitArr_J$. The complexity of *fold* and *unfold* operations is inversely related to the selectivity.

4.2 Step 2 – Subgraph Matching Algorithm

As explained in **Definition 1** – an RDF subgraph which matches the query pattern, maps each triple pattern to an RDF triple. In the previous section, we generated a much smaller BitMat associated with each tp-node. At the end of the pruning phase, each BitMat has a set of RDF triples to which the respective tp-node is mapped. In this step – we map each tp-node to an RDF triple from its BitMat such that all the restrictions enforced by the join variables are satisfied. This step conceptually resembles *subgraph matching* in pattern mining terminology.

Algorithm 4 Subgraph Matching

```
1: T_min = tp-node with least number of triples in BitMat after pruning step.
2: map m
3: Map_TP_Node(T_min)
4:
5: function Map_TP_Node(tp-node T)
6: set restrictions = null
7:
8: for each neighbor n of T in G_tp do
9:    if m.contains(n) then
10:      jvar = shared_var(n, T)
11:      restrictions.add(binding(jvar, n))
12:   end if
13: end for
14:
15: for each r in restrictions do
16:   restrict rows/columns of BM_T to access
17: end for
18: for each triple t in restricted BM_T do
19:   m.add(T, t)
20:   if m.size() == |V(G_tp)| then
21:      output m
22:      m.erase(T)
23:   else
24:      for each unmapped neighbor n of T do
25:         Map_TP_Node(n)
26:         m.erase(n)
27:      end for
28:   end if
29: end for
30: end function Map_TP_Node
```

A brute-force approach to subgraph matching would involve – 1) generating all possible permutations of mapping all tp-nodes to the respective RDF triples, and 2) checking each mapping for validity based on join variable restrictions. But intuitively, a tp-node having lesser number of RDF triples gets mapped to the same RDF triple in more mappings (on an average) as compared to the tp-nodes with more RDF triples. Making use of this fact, we order the tp-nodes by increasing number of triples associated with them. We select the first tp-node in this order as the starting node for *subgraph matching* step (Line 1 in Algorithm 4). This helps in reducing the iterative passes over BitMats to produce mappings.

For the purpose of subgraph matching algorithm, we consider $\mathcal{G}_{tp}$, an induced subgraph of $\mathcal{G}$ with only tp-nodes. At each tp-node $\mathcal{T}$, we first check all its neighboring tp-nodes that are already mapped to a triple. This restricts the triples to which $\mathcal{T}$ can be mapped (Lines 8–17 in Algorithm 4). Once each tp-node is mapped to a triple, we output a valid mapping – representing a matching RDF subgraph – and further continue with the remaining triples (Lines 18–29 in Algorithm 4). *shared_var(n, $\mathcal{T}$)* method returns the common join variable between two triple patterns, and *binding(jvar, n)* method gets the binding of the *jvar* from the triple mapped to tp-node n. *restrictions* set imposes constraints on the RDF triples that can be mapped to a tp-node, based on its neighbors in $\mathcal{G}_{tp}$. This completes the generation of SPARQL BGP query results.

Note that all the algorithms described previously work on a compressed BitMat. Only *Initialization* creates a new BitMat associated with each tp-node. Subsequent *fold* and *unfold* always operate on the same BitMat. In fact, in BitMat's pruning step, the memory requirements go on reducing as the query progresses due to the elimination of RDF triples from BitMats.

4.3 Comparison of Pruning Technique with Relational Join

BitMat's structure is similar to the idea of compressed bitmap indexes in relational joins and OLAP data warehousing techniques [17, 13, 21]. But there are some key differences. In a relational scheme, bitmap indexes are created mostly on the columns having fewer distinct entries and those which are known to participate in more joins. But a relational join between multiple tables over different columns cannot always make use of bitmap indexes for the later joins. This is due to the fact that after the first level of join, a relational query processor has to materialize the results of the previous join to carry out the next join, and this materialized table does not always have indexes formed on it (unless join-indexes are precomputed based on heuristics). As opposed to that, BitMat's pruning and subgraph matching algorithms *always* use compressed BitMats. They need neither materialization of intermediate results, nor explicit listing of triples at any step. Our final result generation using a constraint graph $\mathcal{G}$ differs significantly from the traditional database joins.

5. EVALUATION

The BitMat system is implemented in C on the Linux platform. All the experiments were carried out on a 2x Intel Xeon 5365 Quad Core 3.0GHz processor with 16GB of memory. We evaluated BitMat's performance against three contemporary RDF stores viz. Hexastore [28], Jena-TDB [3], and Redland [2]. Out of these, TDB served as a persistent RDF store; while Hexastore and Redland served as in-core RDF stores. We chose Hexastore since it presents the most recent developments in the field of BGP query

[7]Selectivity of a triple pattern is low if there are more number of triples associated with it and vice versa. Selectivity of $MaskBitArr_J$ is high if there are fewer 1 bits and vice versa.

processing[8]. Redland's results are with their *tree-structure* for storing triples. This in-memory representation was chosen over *hashes* because – in our experience – it performed better. We chose Jena-TDB as a persistent store due to its ease of use and a wide range of input RDF formats. We increased the maximum allowed memory usage of TDB (java -Xmx option) to 7GB[9] to allow execution of queries generating large result data. Please note that due to the small memory footprint of the BitMat system, we could have as well used a desktop machine with much lesser memory, but that would have rendered other systems unusable, or rather usable with very small datasets.

We used six datasets of varying sizes ranging from 0.2 to 51 million triples. They are: Uniprot's 0.2 and 22 million triples [26], LUBM's 1 million and 6 million triples, generated using LUBM data generator [14], Wikipedia's 47 million triples [29], and Barton's 51 million triples [6]. The characteristics of these datasets are given in Table 1. We used 7 of the SPARQL BGP queries for Uniprot available with Swiss Institute of Bioinformatics [20]. We used 19 SPARQL BGP queries for the LUBM dataset available as a part of OpenRDF [18] benchmark. Since we did not generate *inference triples* on the original LUBM data, we had to modify some of the queries to execute them on the original LUBM triples. Due to unavailability of standard queries for the Wikipedia dataset we generated 3 queries ourselves. For Barton data, we used 3 queries given in the original paper [5] and one constructed ourselves. Rest of the queries in [5] were unusable due to the specific group-by or UNION SQL constructs which are not supported by the BitMat query interface at present. A complete description of the queries is omitted due to space constraints, but can be provided upon request.

Table 1: Dataset characteristics

Dataset	#Triples	#S	#P	#O
Uniprot 0.2million	199,912	30,007	55	45,754
LUBM 1million	1,272,953	207,615	18	155,837
LUBM 6million	6,656,560	1,083,817	18	806,980
Uniprot 22million	22,619,826	5,328,843	91	4,516,903
Wiki 47million	47,054,407	2,162,189	9	8,268,864
Barton 51million	51,598,374	12,326,627	285	16,705,854

We evaluated our results over two main factors:

- Query times, i.e., time taken to evaluate SPARQL BGP queries.
- Dynamic memory requirements, i.e., *peak virtual memory* required at any point for in-core execution of queries.

All query times were averaged over 10 consecutive runs. For LUBM datasets we have not shown results of all 19 queries to avoid cluttering the graphs in Figure 7. Instead we have shown results of 9 out of 19 queries. The rest of the results can be provided upon request.

Some of the salient features of BitMat are – memory footprint of the system is much smaller than all other systems. In fact, Hexastore was observed to spend a long time (160+ minutes) to load 22 million triples, consuming almost entire virtual memory (∼15GB), and still did not finish loading it. Hence we aborted the operation and could not evaluate it for our datasets larger than 6 million triples. Redland could load up to 22 million triples of Uniprot dataset; but lack of additional memory caused queries to fail. TDB could load Wikipedia and Barton dataset in its persistent store but it ran out of resources while executing some Wikipedia and Barton queries. Those are the queries where there is no result for TDB system in Figure 7. Note that for smaller datasets like Uniprot 0.2 million and LUBM 1 million, BitMat's memory utilization is very small compared to others, hence we have shown a magnified view in inset for those in Figure 7. It was observed that memory utilization and query performance of other systems degraded much faster as the size of the dataset and result-set of the query increased. Whereas BitMat gave consistent performance even for very large triplesets and queries generating large output results.

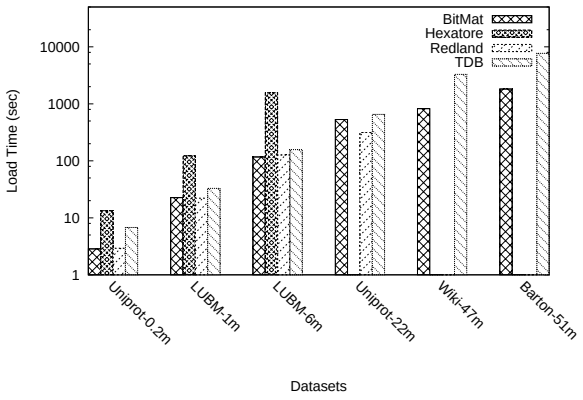

Figure 6: Initial load times (y-axis on log-scale)

Table 2: Compressed BitMat load time

Dataset	Time (sec)
Uniprot 0.2million	0.07
LUBM 1million	0.559
LUBM 6million	2.97
Uniprot 22million	14.47
Wiki 47million	6.31
Barton 51million	33.62

BitMat's *Initialization*, *fold*, and *unfold* operations always operate on the compressed BitMat. This sometimes adds computational overhead as compared to the direct access to the data. Hence for some queries BitMat reported more time than some of the other systems. Note that Y-axis of query times is drawn on a ***log-scale***. We would like to point out that BitMat's performance for the queries on larger datasets is much superior as compared to the queries on smaller datasets. This is due to the fixed cost incurred during *Initialization*. Query times also depend on the *selectivity* of triple patterns and the join results. For instance, Barton Query-1 generates ∼19 million results, which took 152 seconds. Our real goal is to build a *scalable* RDF graph representation while providing a comparable query performance which was demonstrated by BitMat's performance on larger datasets like Wikipedia and Barton.

[8]We obtained the compiled binaries from the authors of Hexastore to perform our experiments.

[9]Allocating entire 16GB would render the system without any resources for other processes.

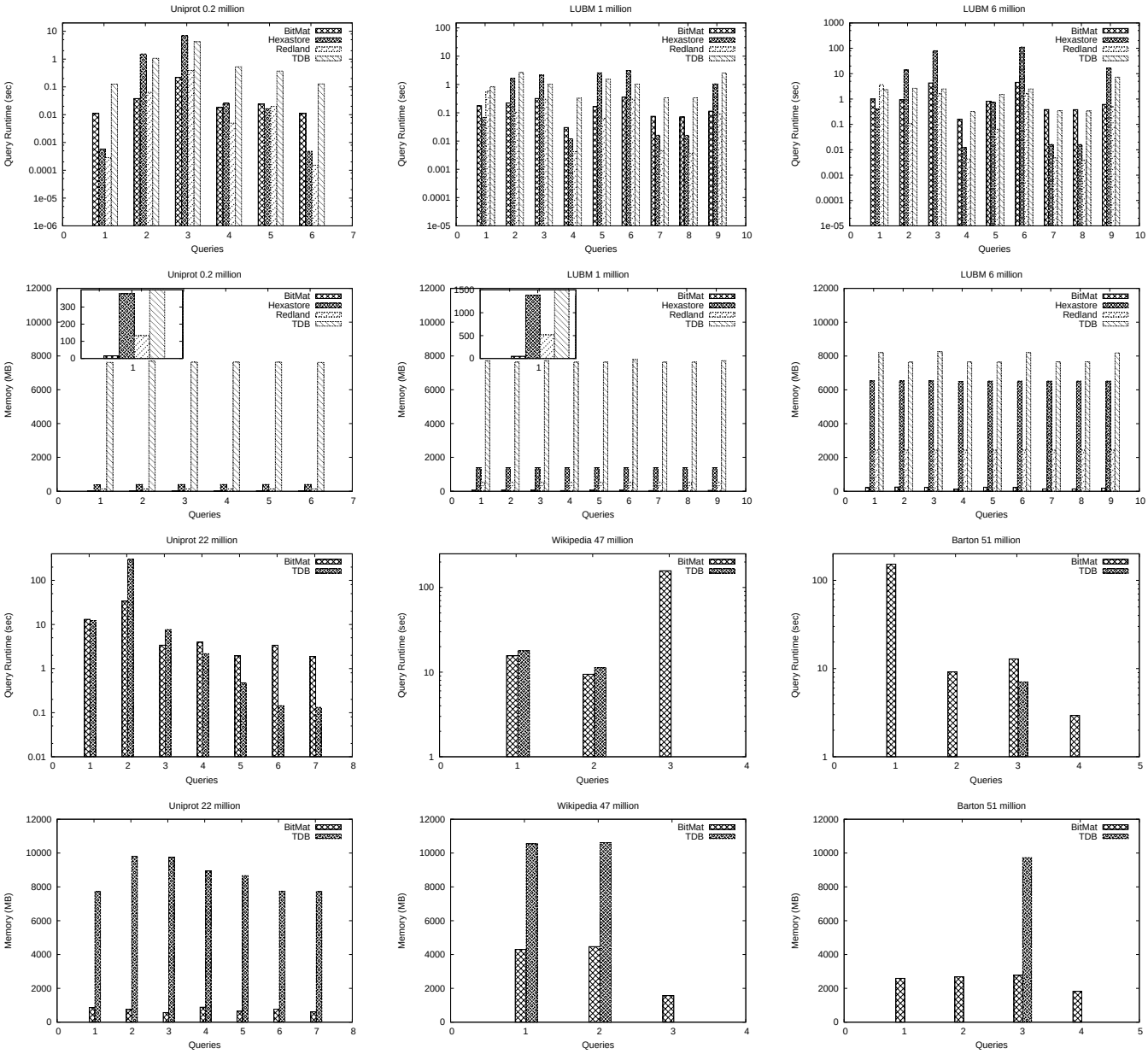

Figure 7: Query times on log-scale (first and third row) and Dynamic Memory Utilization on linear scale (second and fourth row) (Note: Absence of a bar indicates that the corresponding system ran out of resources)

Table 3: Effect of Pruning Algorithm after Initialization

Dataset(Q-ID)(Size)	TP1	TP2	TP3	TP4	TP5	TP6
LUBM(1)(6m)	126,112⇒6254	1000⇒50	999⇒998	519,842⇒6254	15,860⇒998	162,085⇒6254
LUBM(5)(1m)	24,019⇒1874	189⇒15	99,566⇒1874	15⇒15	106,409⇒1874	
Uniprot(5)(22m)	106,524⇒261	340⇒261	108,989⇒261	106,524⇒261		
Barton(2)(51m)	1,547,202⇒31,194	34,868⇒31,194	51,598,374⇒862,293			

As pointed out in Section 4, the pruning step achieved a considerable reduction in the candidate RDF triples for *subgraph matching*. We briefly list out this effect in Table 3 for four queries. *TP* stands for triple pattern in the query. The number before ⇒ denotes number of triples left in BitMat after the *Initialization* step and the number after ⇒ shows the number of triples left after executing the pruning algorithm. A similar effect was observed in all the queries.

BitMat's load time can be classified into two main parts: (A) Mapping each subject, predicate, object to the integer IDs as elaborated in Section 3, generating a compressed BitMat for the first time in memory, and storing it on disk. This is equivalent to the data loading and index generation process of other triplestores. Figure 6 shows this comparison.

(B) Second part loads the compressed BitMat stored on disk previously in step (A), into memory. Since BitMat is an in-core RDF store, on every start of the BitMat process, we need to load the compressed BitMat in memory. Hence the load time is an important parameter which is given in Table 2. There is no equivalent comparison metric for other systems.

6. RELATED WORK

In addition to the previously mentioned systems such as C-store, Virtuoso, TDB, Hexastore, there is much existing work for processing RDF graphs. SwiftOWLIM [23] uses in-memory hash-tables to load and process large data, but their focus is to build *OWL inferences* in memory. Other in-memory RDF graph stores, such as BRAHMS [11] and GRIN [24], have tried to focus on variable-length-path queries over RDF graphs. BRAHMS has evaluated their system for up to 6 million LUBM triples whereas GRIN was executed with up to 17,000 triples. Work by Oren et al. [19] fits large RDF graphs in memory using Bloom filters, but focuses on *approximate* query answering. The RDF-3X system [16] is based on the same six indexes as Hexastore. It incorporates both index compression and a join index to facilitate BGP query processing. They build B^+-trees and employ traditional database techniques to process BGP queries.

The RDFCube [15] system is conceptually closest to BitMat. RDFCube also builds a 3D cube of subject, predicate, and object dimensions. However, RDFCube's design approximates the mapping of a triple to a cell by treating each cell as a hash bucket containing multiple triples. They primarily used this as a distributed structure in a peer-to-peer setup (RDFPeers [7]) to reduce the network traffic for processing join queries in a conventional manner. In contrast, BitMat's compressed structure maintains unique mapping of a triple to a single bit, and also employs a different query processing algorithm. RDFCube has demonstrated a bitcube of only up to 100,000 triples.

7. CONCLUSION AND FUTURE WORK

To conclude this paper, we would like to emphasize that BitMat's structure and algorithms provide a novel way of processing very large RDF graphs efficiently and entirely in memory. It differs from the approach followed by the conventional systems which mainly focus on using more indexes and hence memory to achieve query performance. BitMat demonstrates an acceptable query performance while keeping the memory footprint of the system very small – even when persistent stores like TDB failed to operate. This makes it usable under multiple scenarios – embedded systems, mobile devices, and cluster machines with small main memory per node.

BitMat is a prototype implementation of a novel technique of representation and querying of large RDF graphs in memory. It is built as an independent engine, as opposed to many other RDF triplestores which are built on top of existing relational databases or which straightaway follow relational querying techniques. Hence currently BitMat does not support all types of SPARQL queries and other operations (e.g. addition/deletion of triples) supported by the conventional database systems. These enhancements are a part of our future work. BGP queries being the fundamental building blocks of a SPARQL query, presently we have focused only on those. BGP queries with *Cartesian joins*[10] are rare in the context of RDF data, hence they are not handled right now as well. In the current implementation, we need to load the entire BitMat in memory before processing the queries. In future, this idea can be extended to process RDF graphs of much larger orders of magnitude by strategically loading parts of the BitMat and processing them. This is analogous to processing parts of a large database table by spooling to the disk. The notable point is that BitMat would be able to load and process much larger amounts of data in the same amount of space than conventional systems.

BitMat presents a new paradigm of representing and processing large amounts of RDF data, taking the scale of RDF data that can be handled to the next level.

8. REFERENCES

[1] RDFa - Primer. http://www.w3.org/TR/rdfa-syntax/.

[2] Redland - RDF Libraries. http://librdf.org/.

[3] TDB - A SPARQL Database for Jena. http://jena.sourceforge.net/TDB/.

[4] Reference hidden for anonymity.

[5] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management using Vertical Partitioning. In *VLDB*, 2007.

[6] Barton RDF Dataset. http://simile.mit.edu/wiki/Dataset:_Barton.

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

[8] D-gap Compression Scheme. http://bmagic.sourceforge.net/dGap.html.

[9] O. Erling. Advances in Virtuoso RDF Triple Storage (Bitmap Indexing), 2006. http://virtuoso.openlinksw.com/-wiki/main/Main/VOSBitmapIndexing.

[10] A. Gara, M. A. Blumrich, D. Chen, et al. Overview of the Blue Gene/L System Architecture. *IBM Journal of Research Development*, 2005.

[11] M. Janik and K. Kochut. BRAHMS: A WorkBench RDF Store and High Performance Memory System for Semantic Association Discovery. In *ISWC*, 2005.

[12] Jena - A Semantic Web Framework for Java. http://jena.sourceforge.net/.

[13] T. Johnson. Performance Measurements of Compressed Bitmap Indices. In *VLDB*, 1999.

[14] Lehigh University Benchmark (LUBM). http://swat.cse.lehigh.edu/projects/lubm/.

[15] A. Matono, S. M. Pahlevi, and I. Kojima. RDFCube: A P2P-based Three-dimensional Index for Structural Joins on Distributed Triple Stores. In *DBISP2P at VLDB 2006*, September 2006.

[16] T. Neumann and G. Weikum. RDF-3X: A RISC-style Engine for RDF. In *VLDB*, 2008.

[17] P. O'Neil and G. Graefe. Multi-Table Joins Through Bitmapped Join Indices. In *SIGMOD Record*, volume 24, September 1995.

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

[19] E. Oren, C. Gueret, and S. Schlobach. Anytime Query Answering in RDF through Evolutionary Algorithms. In *ISWC*, 2008.

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

[21] S. Sarawagi. Indexing OLAP data. *Data Engineering Bulletin*, 20(1), 1997.

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

[22] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2005. pg 484-487.
[23] SwiftOWLIM Semantic Repository. http://www.ontotext.com/owlim/index.html.
[24] O. Udrea, A. Pugliese, and V. Subrahmanian. GRIN: A Graph Based RDF Index. In *AAAI*, 2007.
[25] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of ACM*, 23(1), 1976.
[26] UniProt RDF. http://dev.isb-sib.ch/projects/uniprot-rdf/.
[27] C. Weiss, A. Bernstein, and S. Boccuzzo. i-MoCo: Mobile Conference Guide - Storing and querying huge amounts of Semantic Web data on the iPhone/iPod Touch. In *ISWC Billion Triple Challenge*, 2008.
[28] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In *VLDB*, 2008.
[29] Wikipedia RDF Dataset. http://labs.systemone.at/wikipedia3.

APPENDIX

A. PROOF OF MINIMAL TRIPLE SET GENERATION

Assumption 1: Induced graph $\mathcal{G}'$ over constraint graph $\mathcal{G}$ is acyclic.
Assumption 2: No two triple patterns in a BGP query share more than one join variables among them.

PROPERTY 4. *At each jvar-node j, in Algorithm 3, the sequence of* ***fold*** $\rightarrow$ ***bitwise AND*** $\rightarrow$ ***unfold*** *operations ensure that all the BitMats associated with respective triple patterns have the same reduced set of bindings for j. Properties 1 and 2 hold too.*

PROPERTY 5. ***Assumption 2*** *enforces additional restriction on Property 3 that two adjacent jvar-nodes share* ***exactly*** *one tp-node (and hence one BitMat) among them.*

In the first phase, when we traverse the nodes in the decreasing order of node-IDs, we traverse the tree *bottom-up* from leaves to root. *prune_for_jvar* method monotonically reduces the set of bindings for each jvar-node j and in turn the triples in the BitMats associated with the respective tp-nodes. By Property 5, since any two adjacent jvar-nodes j and k share an BitMat between them, this can further reduce the bindings of the adjacent jvar-node k. In the context of Figure 5, elimination of triples from $BitMat_2$ as a result of *prune_for_jvar(?m)*, affects the bindings generated for *?n* by $BitMat_2$. Thus in the first phase, constraints get propagated from leaf nodes to the root. There can be one or more tp-nodes $\mathcal{T}$ in the query which have only one jvar-node j. An BitMat associated with such $\mathcal{T}$ is not shared between two jvar-nodes. An example of such BitMat is BM_2^p in Figure 8. It is due to the existence of such BitMats, that a second pass over $\mathcal{G}'$ is needed. Suppose, at the end of the first bottom-up traversal of graph $\mathcal{G}'$, BitMat BM_1^p has bindings {1, 2, 3} for *?p*. But BM_2^p, which is not shared with any other jvar-node generates bindings {1, 2} for *?p*. After calling *prune_for_jvar(?p)*, some triples in BM_1^p will get eliminated to remove binding *'3'* of *?p*. This in turn can affect the bindings of *?p*'s children and their children in turn (*?m, ?n, ?c, ?d*). Hence a second pass, in *top-down* manner is needed on graph $\mathcal{G}'$.

LEMMA 1. *At the end of the second top-down traversal over graph $\mathcal{G}'$, no more triples can be eliminated from any of the BitMats.*

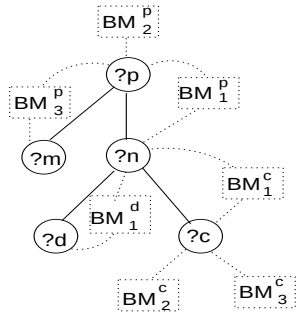

Figure 8: Example of graph $\mathcal{G}'$ (Note: BitMats connected with dotted edges are shown for the sake of understanding, they are not part of graph $\mathcal{G}'$)

PROOF. We prove our lemma by contradiction. Consider any internal (non-leaf or non-root) jvar-node *?n*. Let *?p* be its parent and *?c* be one of the children. Let us assume that the second top-down pass over *?n*, did *not* generate minimal set of triples in the BitMats associated with *?n* (BM_1^c, BM_1^p, BM_1^d)[11]. This can happen only if the triples in those BitMats can be eliminated further while traversing over other jvar-nodes. As per Property 5, any two adjacent jvar-nodes share *exactly* one BitMat among them.

Let us first consider the child node *?c*'s case. After the second top-down pass over *?n*, let B_1^c be the set of bindings generated by BM_1^c for *?c*, $B_2^c \rightarrow$ set of bindings generated by BM_2^c, and $B_3^c \rightarrow$ set of bindings generated by BM_3^c. Now, triples in BM_1^c can get further eliminated *only if* $|B_1^c| > |B_2^c|$ or $|B_3^c|$. But in the first phase, during the bottom-up pass over *?c*, *prune_for_jvar(?c)*'s **fold** $\rightarrow$ **bitwise AND** ensured that $B_1^c = B_2^c = B_3^c$. Since the triples always get eliminated monotonically, after the second pass over *?n*, $|B_1^c| \leq |B_2^c|$ or $|B_3^c|$. This is a contradiction to our assumption of $|B_1^c| > |B_2^c|$ or $|B_3^c|$. Since $\mathcal{G}'$ is acyclic, there is only one path reaching *?c* through *?n* from the root node. Hence *?c*'s bindings and triples in the associated BitMats remain unchanged until the second pass over *?n*. Thus it can be proved that no more triples in BM_1^c can be eliminated after calling *prune_for_jvar(?c)*, hence BM_1^c has minimal set of triples. Without loss of generalization, same analysis can be applied to all other children of *?n*.

Now consider parent node *?p*'s case. Each node has exactly one parent. *?n* and *?p* share exactly one BitMat (BM_1^p) among them. As stated earlier, *?n*'s bindings can reduce further only if triples in BM_1^p can be eliminated further. To prove that after the second top-down pass over *?p*, BM_1^p contains minimal set of triples, we simply denote *?p* $\Rightarrow$ *?n* and *?n* $\Rightarrow$ *?c*, and use the same analysis as given above to prove that BM_1^p has minimal set of triples after second pass over *?p*. Without loss of generalization, same analysis can be applied to all the nodes in $\mathcal{G}'$. $\square$

[11] BitMats associated with *?n* are nothing but BitMats of tp-nodes connected to *?n*. For simplicity we call those BitMats associated with *?n* as well.