BitMat: A Main-memory Bit Matrix of RDF Triples for Conjunctive Triple Pattern Queries

Medha Atre
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, USA
atrem@cs.rpi.edu

Jagannathan Srinivasan
Oracle
1 Oracle Drive
Nashua, NH, USA
jagannathan.srinivasan@oracle.com

James Hendler
Dept. of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, USA
hendler@cs.rpi.edu

ABSTRACT

This poster proposes BitMat, a bit matrix structure for representing a large number of RDF triples in memory and processing conjunctive triple pattern (multi-join) queries using it. The compact in-memory storage and use of bitwise operations, can lead to a faster processing of join queries when compared to the conventional RDF triple stores. Unlike conventional RDF triple stores, where the size of the intermediate join results can grow very large, our BitMat based multi-join algorithm ensures that the intermediate result set remains small across any number of join operations (provided there are no Cartesian joins). We present the key concepts of BitMat structure, its use in processing join queries, describe the preliminary experimental results with UniProt and LUBM datasets, and discuss the possible use case scenarios.

1. KEY CONCEPTS

BitMat is a 3-dimensional (subject, predicate, object) bit matrix flattened in 2-dimensions for representing RDF triples. Each element of the matrix is a bit denoting the presence or absence of that triple (subject predicate object) by the bit value 1/0. Thus, very large RDF triple-sets can be represented compactly in memory as BitMats. Bitwise AND/OR operators are used to process join queries expressed as conjunctive triple patterns. The BitMat representation and its use in processing joins allows i) fast identification of candidate result triples, and ii) a compact representation of the intermediate results for multi-joins. BitMat is similar to RDFCube [1], which builds a 3D cube of subject, predicate, and object dimensions. However, RDFCube's design approximates the mapping of a triple to a cell by treating each cell as a hash bucket containing multiple triples. It is primarily used to reduce the network traffic for processing join queries over a distributed RDF store (RDFPeers) by narrowing down the candidate triples. In contrast, BitMat structure maintains the unique mapping of a triple to a single bit element. This does make the BitMat size larger compared to RDFCube for the same dataset. However, the use of run length encoding (RLE) on the bit sequences within a BitMat, reduces the storage overhead (see Section 3). Also, here the join results are represented by a *result BitMat*. The goal here is to speed up multi-join queries, especially for the queries where individual triple patterns in the join are not selective but the combined join result is selective.

BitMat Creation and Auxiliary Tables: We leverage the fact that the number of distinct predicates is typically small (< 100) in a RDF dataset. The conceptual 3-dimensional bit matrix is represented as a concatenation of (S,O) or (O,S) matrices for all the distinct predicates (see Figure 1), which together form a bit-matrix as well as a *mat* of *bits* (and hence the name *BitMat*). The concatenation can be done along the subject dimension ((S,O) matrices), referred to as subject BitMat or the object dimension, referred to as object BitMat.

BitMat for a RDF triple-set is built as follows: 1) Create three auxiliary tables to maintain mappings of distinct subjects, predicates, and objects to the sequence-based identifiers, 2) Group the RDF triples by predicates, 3) To build a subject BitMat, for each predicate group, build a (S,O) matrix by transforming the (s,o) pairs to the equivalent sequence-id based representation and setting the corresponding bit in (S,O) matrix to 1, and 4) Concatenate (S,O) matrices together to get the two-dimensional BitMat 5) Apply RLE on each subject row in the concatenated BitMat. 5) We maintain three more tables that contain position mappings of URIs that appear in two dimensions in different triples (subject-object, subject-predicate, predicate-object). These tables help in evaluating cross-dimension join queries. E.g. (?y :p2 ?s . ?s :p1 ?x) makes use of (subject-object) mapping table. Our join procedure can be carried out using the following set of primitives.

BitMat Primitives: (1) *filter(BitMat, TriplePattern) returns BitMat*: Identifies a subset of triples that satisfy the triple pattern, and clears bits of all other triples. E.g. i) a triple pattern with only bound subject value, clears all the bits in the rows other than the row corresponding to the bound subject value in the BitMat, a bound predicate retains bits in a single (S,O) matrix, a bound object retains bits in a list of (S,P) columns; ii) A triple pattern with two bound values retains a subset of the bits retained in the one bound value case, e.g. (:s1 :p1 ?x) retains a part (corresponding to predicate :p1) of single :s1 horizontal row of BitMat. **(2)** *fold(BitMat, retainDimension) returns a bit-array*: Bitwise OR is performed on the two dimensions other than *retainDimension* which results in a bit-array. E.g. *fold(filter(BM, '?s ?p ?x'), 'predicate')* folds all the object and subject bits and reduces the BitMat to a bit-array. *fold(filter(BM, ':s1 ?p ?x'), 'predicate')* folds all the object bits in ':s1' row to reduce it to a bit-array. Length of this bit array is the number of distinct predicates. Intuitively, a bit set to 1 corresponding to ':p1' in the bit-array indicates the presence of at least one triple of the form (:s1 :p1 ?x). **(3)** *unfold(BitMat, Mask, retainDimension) returns BitMat*: For the bit set to 0 in the *mask*, all the corresponding bits in the BitMat for the specified dimension are cleared. E.g. *unfold(filter(BM, ':s1, ?p, ?x'), '101', 'predicate')* would result in clearing all the bits in the input BitMat corresponding to predicate ':p2' (see Figure 1). Although conceptually the return type is shown as BitMat for these operations, internally they are managed as a collection of compressed bit sequences.

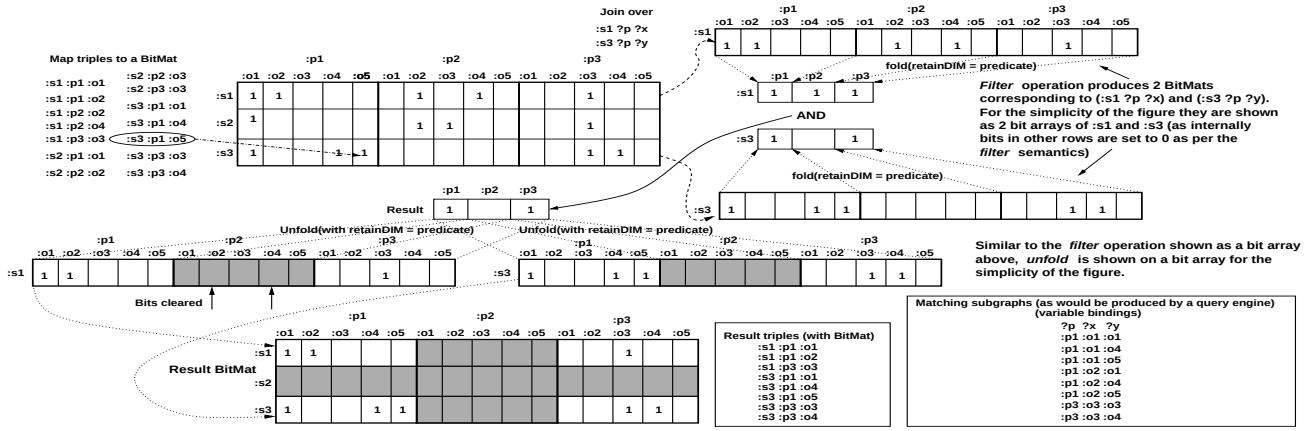

Figure 1: BitMat and sample join

2. JOIN PROCESSING

A conventional RDF query processing engine produces zero or more rows (matching subgraphs) having variable bindings (see Figure 1). Formally, a BitMat join takes the original RDF graph G represented by BitMat and produces another graph G' represented by the result BitMat such that G' is a subset of G in which: (1) A triple being absent indicates that it is *not* part of any matching subgraph. (2) A triple being present indicates that it must be part of at least one matching subgraph. Our join algorithm is given below.

Let BM be the BitMat of the original triple-set
Let tp_1 and tp_2 be the join triple patterns
/* filter and fold */
BM_{tp1} = filter(BM, tp_1); BM_{tp2} = filter(BM, tp_2)
f_1 = fold(BM_{tp1}, $retainDimension_{tp1}$)
f_2 = fold(BM_{tp2}, $retainDimension_{tp2}$)
j_{res} = AND(f_1, f_2)
r_1=unfold(BM_{tp1}, j_{res}, $retainDimension_{tp1}$)
r_2=unfold(BM_{tp2}, j_{res}, $retainDimension_{tp2}$)
/* Produce final result BitMat */
Let B' be an empty BitMat
B' = OR(OR(B', r_1), r_2)

Figure 1 shows the join of (:s1 ?p ?x . :s3 ?p ?y) using the above join algorithm. In both the triple patterns, subject is bound and they are joined over the predicate. Thus, we fold both the filtered BitMats by retaining 'predicate' dimension to get two bit sequences which are ANDed. The result is used as the *mask* in *unfold* operations to set the bits corresponding to :p2 to 0.

Multi-Joins: In multi-joins two or more triple patterns join over two or more variables (e.g. (:s1 ?p ?x . :s2 ?p ?y . ?z :p3 ?x)). In multi-joins the result of a later join can change the variable bindings generated by a previous join. E.g. consider (:s1 ?p ?x . :s3 ?p ?x) join over the BitMat in Figure 1, after obtaining the result BitMat (B') for predicate join (?p), the join on object (?x) requires filter and fold of B' for (:s1 ?p ?x) and (:s3 ?p ?x) by retaining the 'object' dimension, ANDing them, and unfolding. This clears all the bits in :o2, :o4, and :o5 columns thereby returning two distinct matching subgraphs (p,x)={(:p1,:o1),(:p3,:o3)}. Our algorithm to build and resolve multi-join dependencies is omitted due to lack of space. Also, currently we assume that the conjunctive triple patterns do not have Cartesian products.

Join Reduction Fraction (JRF): is defined to characterize the effectiveness of the BitMat based join processing.
JRF = (#Triples in the result BitMat)/(#Total triples).
JRF for the example in Figure 1 is 8/14 = 0.57.

3. RESULTS AND USE CASES

We experimented with 200K UniProt and 1million LUBM RDF triple-sets. Since BitMats will be sparsely populated (10% or lower), using run length encoding (RLE) significantly reduces the compressed BitMat size (Table 1).

Table 1: BitMat size

Dataset (#triples)	BitMat size (# of cells)	BitMat size (uncompr / compr)
UniProt (200k)	75,511,715,290	9GB / 1.4MB
LUBM (1million)	1004,445,322,713	116GB / 11.5MB

The storage overhead of auxiliary tables is ≈10MB and ≈20MB for UniProt and LUBM datasets respectively. Thus, overall memory requirement of our scheme is quite small (11.4MB and 31.5MB respectively). We executed the following query on UniProt 200K dataset, which involved a join on subject dimension:

(?s <urn:lsid:uniprot.org:ontology:author> ?x . ?s <rdf:type> ?y)

The above query (#resulting triples=31,044 JRF=0.16) took 0.02sec. A similar two pattern query on LUBM (#resulting triples=386,519 JRF=0.29) took 0.28sec. These preliminary results demonstrate the effectiveness of joins using BitMat data structure.

Although currently we do not have an algorithm to enumerate matching subgraphs from the result BitMat, BitMat join procedure can be used as a precursor to a conventional in-memory query processing engine (e.g. Jena-ARQ) for fast identification of the result triples from a large triple-set. Let T_{BitMat} be the time taken by our join algorithm to produce the join result BitMat, $T_{QoBitMat}$ be the time taken by a query engine to produce the matching subgraphs by using the BitMat results, and T_{QOrig} be the time taken by a query engine to produce the results without BitMat interception. It is our hypothesis that $T_{BitMat} + T_{QoBitMat} \ll T_{QOrig}$ for a useful class of queries (e.g. JRF < 0.5). Also, *'EXISTS'* or *'ASK'* queries for an arbitrarily large number of joins can be performed faster using BitMat (existence of one or more 1 bits in the result BitMat indicates the existence of one or more matching subgraphs). Lastly, as a part of ongoing work, we are designing an efficient algorithm to produce the matching subgraphs using the result BitMat so that BitMat based join can be used as an independent query processor.

4. REFERENCES

[1] A.Matono, S.Mirza, and I.Kojima. RDFCube: A P2P-based Three-dimensional Index for Structural Joins on Distributed Triple Stores. In *DBISP2P, In conjunction with VLDB'06.*