MATRIX "BIT"LOADED: A SCALABLE LIGHTWEIGHT JOIN QUERY PROCESSOR FOR RDF DATA

MEDHA ANTRE, VINEET CHAOJI, MOHAMMED J. ZAKI, JAMES A. HENDLER

REPORT

HOT TOPICS IN INFORMATION RETRIEVAL SEMINAR

AMALIA RADOIU

OPPONENT: TEODOR STOEV

INTRODUCTION

With the expanding use of the web and the databases, there is also an increasing need to be able to search the data fast and efficiently.

The W3C standard used to represent information is RDF (Resource Description Framework), which consists of triples representing the relationships between its subjects (S), object (O) and the name of the relationships is given by the predicate (P) in increasingly gaining importance as more data is becoming available in the RDF format. It is used to represent data for various fields like social networks, Wikipedia and sciences. SPARQL is used as a query language for RDF data.

Although disk space is becoming cheaper and does not pose a problem, executing queries on large databases can be a challenge due to the limited processing power.

There are a number of query processing systems which have been developed. Some of the most notable ones are RDF-3X, Hexastore, Jena-TDB and MonetDB.

RDF-3X is a SPARQL query processor whose storing and indexing RDF triples completely eliminates the need for physical-design tuning. It performs fast merge joins to the largest possible extent and it has a query optimizer for choosing optimal join orders using a cost model based on statistical synopses for entire join paths.

The main features of the Hexastore query processor are the 6-way indexes and data compression. Together with RDF-3X it is one of the most powerful systems available today.

Jena-TDB provides a large scale storage and query of RDF data sets using a pure Java engine. It provides a programmatic environment for RDF and SPARQL and includes a rule-based inference engine.

MonetDB provides a storage model based on vertical fragmentation, a modern CPU-tuned query execution architecture, automatic and self-tuning indexes. Its internal data representation is memory-based, relying on the huge memory addressing ranges of contemporary CPUs.

The systems available today have their limitations with respect to query performance. Systems which generate indexes on the data perform well on queries with highly selective triple patterns. Queries with low-selectivity triple patterns, but highly selective join results have the lowest execution time at systems which use join selectivity estimation or per-computed join tables and indexes.

The category of queries which poses problems for most of the systems are the queries with low-selectivity triple patterns generating a large number of results.

The authors of the paper propose a new system for query execution, BitMat, which performs well on low-selectivity triple pattern queries for both low and high-selectivity join results. The main features of the proposed system are:

- compressed bit-matrix structure for storing huge RDF graphs
- a novel, lightweight, SPARQL query processing method
- the algorithm doesn't need intermediate join tables

In the following sections we are going to present the construction of the BitMat, describe the algorithm and discus the experimental results and conclude the report with the observed strengths and weaknesses of the BitMat algorithm as well as further work.

BitMat Construction

The data is represented as a 3D bit-cube, in which each dimension represents the subjects (S), predicates (P), objects (O). The sets VS, VP and VO to denote the sets of distinct subjects, predicates and objects in the RDF data. Each cell in the 3D cube represents a unique RDF triple. The 3D bit-cube is sliced along a dimension, which gives us two 2D matrices. By slicing along the P-dimension we get two matrices namely S-O and O-S as the inverted equivalent of the first matrix. The P-O matrix is obtained by slicing along the S-dimension and the S-P matrix is obtained by slicing along the O-dimension. We do not compute the inverted matrices for P-S and P-O since experience shows that the usage of those matrices is rare. There are |VS| x |VP| x |VO| possible triples but RDF data contains a considerably fewer number of triples. This property is used to reduce the storage place by applying a gap compression scheme. By gap compression we represent a bit-row of "0100111" by "[0]1123".

Next to the 3D bit-cube we also store the number of triples in each compressed BitMat and a row and a column bit-array which give a condensed representation of all the non-empty row and column values in the given BitMat.

Figure 1 illustrates the construction of the 3D bit-cube from RDF data in the first step.

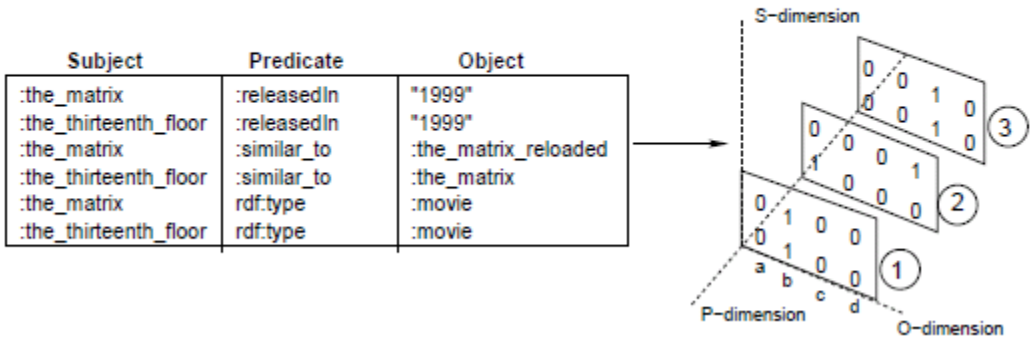

Figure 1. Bit-cube construction

The matrix that is obtained after the slicing along the P-dimension and its inverse are displayed in Figure 2.

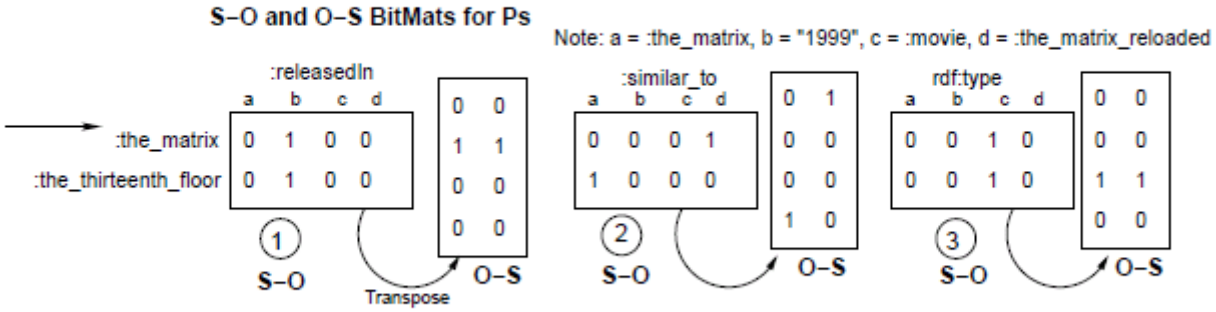

Figure 2. Matrices obtained after slicing

BitMat Algorithm

BitMat Operations

Before describing the actual BitMat algorithm two operations have to be introduced, namely Fold and Unfold. These operations will be later on used within the BitMat algorithm.

The fold operation represented by the method *fold(BitMat, RetainDimension) returns bit Array* folds the input BitMat by retaining the given dimension. The variable *RetainDimension* can have two values, "rows" and "column". If it takes the value "columns", in an S-O matrix all the subject bit-rows are OR-ed together to obtain an "object" bit-array.

The unfold operation, *unfold(BitMat, MaskbitArray, RetainDimension)* unfolds the *MaskBitArray* on the BitMat. Intuitively, it ANDs the MaskBitArray with each row of the BitMat.

Properties

There are three properties which describe the join process.

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 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.

CONSTRAINT GRAPH CONSTRUCTION

The constraint graph G is constructed as follows:

- Triple patterns are represented as tp-nodes and join variables as jvar-nodes.
- An undirected, unlabeled edge between a jvar-node and a tp-node marks that the join variable appears in the triple pattern corresponding to the tp-node.
- If two jvar-nodes appear in the same triple pattern there is an unlabeled, undirected edge between them.
- An undirected, labeled edge between two tp-nodes exists if they share a join variable between them. The label denotes the type of join between the triple patterns and an edge can also have multiple labels.

STEP 1 - PRUNING

In the pruning algorithm the constraints in the constraint graph G, are propagated from root to leaves.

The Gvar graph is a subgraph of G, which only contains jvar-nodes. If Gvar is cyclic, any cyclic edges are removed. The Gvar graph is walked from root to leaves in a breadth-first order and in order to assure complete propagation also the reverse is being done. In order to the children of a node only get processed after its ancestors, the algorithm uses a topological sort of the nodes. For each two adjacent tp-nodes the intersection of their bindings is generated. This is done in the second algorithm, lines 2-5, where the bitwise AND between the folded bit-arrays is computed. For the bindings which get dropped after the intersection, remove the corresponding triples from the BitMats' tp-nodes. In the second algorithm, this is performed in lines 6-9 using the unfold operation.

At the end of the reversed traversal in Algorithm 1, the set of remaining triples is minimal if Gvar is acyclic and it cannot be guaranteed to be minimal if Gvar is cyclic.

Algorithm 1 Pruning Step

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

Algorithm 2 prune_for_jvar(jvar-node J)

1: $MaskBitArr_J$ – a bit-arry 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($MaskBitArr_T$, *dim*)
5: **end for**
6: **for each** tp-node T adjacent to J do
7: *dim* = getDimension(J,T)
8: unfold($MaskBitArr_J$, $MaskBitArr_T$, *dim*)
9: **end for**

STEP 2 – GENERATING FINAL RESULTS

In order to avoid building intermediate join results, the algorithm builds a left-deep join tree. The results are output in a "streaming fashion. That is for k as the number of variables in the query, a map of the bindings which are generated for all the variables at a time is kept and one result is output when each of the k variables have been mapped.

Given we have n triple patterns in a query and each of the BitMats associated with the triple patterns has at most N triples. The BitMat which has the least number of triples is being processed first. Bindings for the variables in the tp-node corresponding to the BitMat1 are generated and stored in a map. The next tp-node is selected so that it shares a join variable with any of the previously selected tp-nodes. Locate the triples which can satisfy the bindings in BitMat2, the corresponds BitMat to tp-node2. If there aren't any bindings in the map to satisfy the bindings in BitMat2, the variable bindings stored in the map are discarded and another tp-node from BitMat1 is selected to generate new bindings. If there are common variable bindings between BitMat2 and BitMat2, move on and pic another tp-node. Repeat the above procedure until all triple patterns are processed and all variables have consistent bindings and output the result. This is done until all the triples in BitMat1 are exhausted.

EXPERIMENTS

EXPERIMENTAL SETTINGS

In the experiments the BitMat method was compared to RDF-3X and MonetDB, as two of the best systems available. The UniProt dataset with 845,075,855 triples was used and another dataset containing 1,334,081,176 triples was generated using LUMB. For the UniProt dataset 13 queries were executed in the tests and for the dataset generated by LUMB 6 queries were executed. The tests were performed on both cold and warm cache.

EXPERIMENT RESULTS

BitMat performed better than the other two systems on queries which have low-selectivity triple patterns and high-selectivity join results and also had a better performance on star-join queries.

On queries with low-selectivity triple patterns and low-selectivity join results BitMat had a better performance than the RDF-3X and MonetDB systems, meaning the aim to develop a system which has a better performance on low-selectivity tripe-patterns and low-selectivity join results was accomplished. On highly-selective triple patterns with highly-selective join results RDF-3X outperformed BitMat and MonetDB.

For the UniProt datasets the geometrical mean was better for BitMat on a cold cache while for a warm cache MonetDB had a better geometrical mean. In the LUMB generated datasets BitMat had outperformed the other two systems on both cold and warm cache with a considerably better geometrical mean.

If Q1 is not part of the geometrical mean computation BitMat, RDF-3X has a better performance on both warm and cold cache for the UniProt datasets and for the LUMB generated datasets it outperforms BitMat on a warm cache.

STRENGTHS, WEAKNESSES, FURTHER WORK

STRENGTHS

- The BitMat algorithm achieved its goal of performing better than the other available algorithms on low-selectivity triple patterns and low-selectivity join results queries.
- It works directly on compressed-data, without having to create intermediate join results.
- It has a compressed bit-matrix structure for storing huge RDF graphs.

WEAKNESSES

- The authors suggest that in order to benefit from the advantages of both BitMat and those of the other state-of-the-art systems, one idea would be to make a hybrid between BitMat and the other systems. One of the main questions when trying to develop such a hybrid system is whether the different query processing systems are compatible with each other and how difficult to implement the hybrid would be.
- For a query having n triple patterns, the size of the amount of memory necessary at the beginning is equivalent to the sum of the sizes of each of the BitMats, since the current implementation of the algorithm loads the BitMat associated with each triple pattern at the beginning of the query processing. Therefore, it is currently not feasible to handle queries which have triple patterns for all variable positions. An example for such a query is the following:

```
SELECT ?y WHERE {
        ?x similar_to ?y
        ?y ?m ?n
}
```

  The query would be equivalent to searching for a movie to which another movie is similar, and about which we have additional information.

This is a drawback of the algorithm because loading all of the BitMats could take a lot of time in the initialization phase. In the experimental UniProt queries Q11 – Q13 it has been proved that having to load entire BitMats slows down the initialization phase.

- According to the authors of the paper, joins across S-P and P-O dimensions are rare in the context of assertional RDF data, and therefore they are not handled by the algorithm. This basically means that queries of the following form cannot be handled:

```
SELECT ?x WHERE {
        :the_matrix ?x ?y
        ?x applies_to :movie
}
```

  The query can be expressed in simple words like this: get all relations that the matrix has to other movies. One could argue whether this is a disadvantage or not, since in real life such an information need might be indeed rare.

- According to the experiments, BitMat has a better performance than the other available systems for low-selectivity triple patterns with low-selectivity join results. Given the fact that in a real life situation, the posed queries also belong to the other two categories of queries, the overall performance is subjective. In the UniProt experiments, the geometrical mean of the performance of BitMat is only better than RDF-3X if Q1 is included. We consider that this argument is not strong enough to state that BitMat's overall performance is better than that of the other available systems.

- The paper mentions that for Uniprot query Q5, the initialization and the pruning phases were very fast, but 90% of the time was spent in the result construction phase. It doesn't mention the complexity of the construction phase, but it points out that the use of the "locality" in memory would reduce the time needed for the construction phase. It would have been useful in terms of assessing the algorithm to have a complexity estimation, so that we could deduce to what extent the use of the "locality" in memory would improve the performance.

  On the other hand, for Uniprot queries Q11-Q13, 90% of the query processing time was spent on initialization to load the BitMats associated with the triple pattern. The authors suggest solving this problem with the use of lazy loading, which would only load the BitMats associated with the tp-nodes instead of loading all the BitMats at the beginning, wait till the first join and then load only the required part of the BitMat necessary for the unfold operation.

- The queries used in the experiments were not organized for the three different query categories that were mentioned in the paper, namely:
  - highly-selective triple patterns queries
  - queries with low-selectivity triple patterns and highly-selective join results
  - queries with low-selectivity triple patterns and low-selectivity join results

Only the queries which belong to the third category were pointed out, but not those which belong to the first or the second category. Not having an exact classification of the queries used in the experiment makes it difficult to evaluate the performance of the BitMat query processor for each of the three query categories.

- The current BitMat query processor does not support FILTER conditions on literals, which are supported by the SPARQL query language. The paper does not address the issue of how such a functionality could be implemented in the future.

- Since the paper has been made some improvements to the RDF-3X system, some bugs were removed and it would be interesting to see a more up-to-date comparison of the performance of the two systems. This would probably also remove the cases when the RDF-3X system aborted and contribute to a more accurate evaluation.

FURTHER WORK

- BitMat doesn't support a formal SPARQL query parser interface and the interface used to output the results is still under development. The authors mention that this will be implemented but do not give details about how they are going to do it.

- In the future versions of the query processor, it should be enhanced to make use of the “locality” in memory.