Undergraduate Project

Efficient Ways to Process RDF graphs and SPARQL queries

Submitted By

Dhawal Upadhyay, Gurkirat Singh

Mentored By

Prof. Medha Atre

Department of Computer Science and Engineering
Indian Institute of Technology Kanpur

Table of Contents

1. Introduction

Resource Description Framework (RDF) is fast becoming the standard of data representation on the web, e.g. DBPedia and Yago ontologies used in training the IBM Watson computer for the *Jeopardy* challenge. The sizes of RDF graphs on the web are growing fast from several million to a couple of ***Billion*** edges. Due to this, a lot of focus has been put on efficient storage and querying over very large RDF graphs. SPARQL is the standard query language for RDF.

In the recent years commercial databases, like Oracle, IBM DB2, or Virtuoso, as well as specialized RDF engines, like BitMat, RDF-3X, TripleBit, gStore have geared up for processing RDF and SPARQL. Out of these systems, BitMat has shown to handle *low-selectivity* queries -- queries accessing a large amount of data -- much more efficiently. However current version of BitMat does not have (a) an efficient way of storing string/URI-to-ID and reverse dictionaries, (b) it lacks an interface to parse the SPARQL queries, and (c) is a single-threaded system.

On this background, our main contributions are listed below:

- A SPARQL query parser for BitMat, and additionally constructed a specialized query graph out of it, as expected by the BitMat system's SPARQL query processor (refer to BitMat-SIGMOD15[7]).
- BitMat system's query processor deals only with integer ID representation of RDF graph nodes and edges. Hence, built several B+ trees of forward (string → ID) and reverse (ID → string) dictionary for fast lookups.
- Parallelized BitMat's fold and unfold operations using multiple threads on the modern "multi-core" processors.

2. B+ Tree Dictionary

A B+ tree is a data structure often used in the implementation of database indexes because of their clever balancing techniques. We use B+ tree to create dictionary of string->id (forward) and id-> string (reverse) mapping. The B+ tree is later queried for these mappings while processing the SPARQL query. We do several optimizations to reduce precomputation and querying time and space

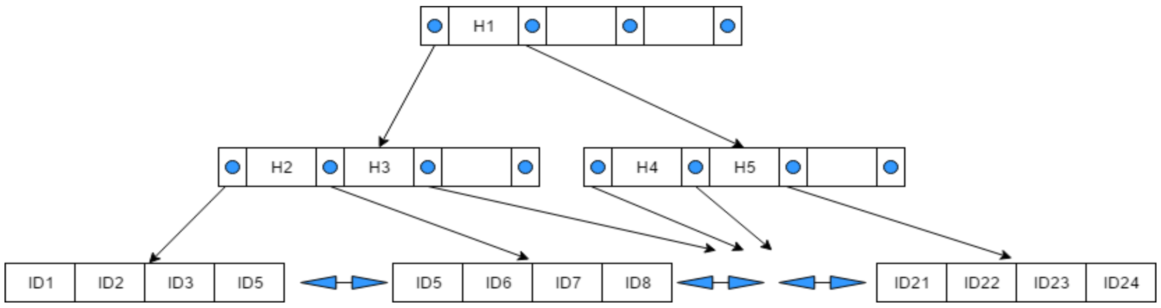

Fig 1: B+ Tree

2.1 Structure of B+ tree

• Forward tree keeps hash value of the string (pair as key and corresponding integer id as value.

• Reverse tree stores integer id as key, and a pointer to the corresponding string in file as value.

• The order of B+ tree (max number of keys in an internal node) is kept 24, after considering size of keys and pointers

2.2. Hashing of strings

We have used a method of using hash of string/URIs as the key for compact representation (two independent hashes of 8 bytes (unsigned long long) each having a range of 2^{64}). Such a large range and a good hash function practically preclude the possibility of any hash collision. Also, by having lesser key size, we can have more keys per page and thus lesser B+ tree height . This technique also reduces time and space complexities and insert/query times remain constant irrespective of string size

2.3. Decomposition into Smaller Trees

We decompose the forward and reverse B+ trees into smaller trees. The exact number of trees is decided by the files system in use. Typically this number is 1000. The decomposition reduces the average height of each B+ tree, thus reducing the number of IOs needed for each insertion and queries. Additionally, we use multi-threading while building these smaller trees. This reduces the construction time by ~75% (from 600s to 160s for 120 million mappings) using 8 cores.

No. of Threads	**Preprocessing Time (forward and reverse db)**
1	31 minutes
2	18 minutes
4	10 minutes
8	7 minutes

Fig 2: Results obtained on dbpedia file having ~200 million strings

2.4. Lookups on the Dictionary

Forward lookup- find which tree the string belongs to (since initially strings are sorted), compute it's hash, query the tree.

Reverse lookup- find which tree the id belongs to, query the tree to get pointer to data entry, and follow the pointer to get the string. Reverse lookups happen much more compared to forward lookups

2.5. Offline Processing for Reverse Lookups

Since the number of reverse lookups is much more than the number of forward lookups, we use an additional optimization to answer group of reverse lookups faster. The BitMat query processor first generates all the reverse lookups that need to be done in the integer ID format. The B+ tree processor takes these integer IDs, sorts them, and then does lookups on the tree and data file (sequential lookups are faster than random lookups). Additionally, it does these lookups parallel for bulk queries. This makes it more efficient to process reverse lookups than doing one random lookup at a time.

3. Lexer and Parser for SPARQL Queries

We have built a SPARQL query lexer and parser using techniques from TripleBit and RDF 3X system. Secondly, we construct a specialized query graph from the abstract syntax tree for an input query, because BitMat expects a special query graph which cannot be served by standard parsers. The specialized graph us constructed using depth-first traversal of Abstract Syntax Tree

4. Parallelization of BitMat

BitMat's fold and unfold operations work on each row in a mutually exclusive fashion. The current size of the bitmat for Dbpedia data is 20 GB, so processing each row of the bitmat sequentially takes huge time. We can reduce the time by dividing the bitmat's operations across different cores available across different CPU threads. Then combine the results of each thread after each thread is finished to give the final results.

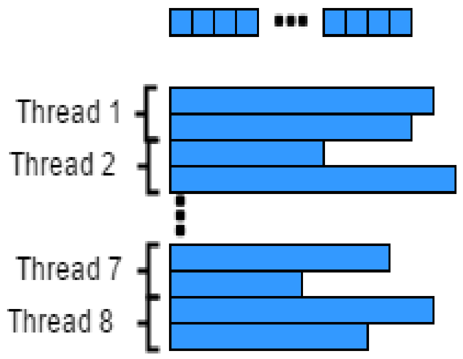

Fig 3. Dividing BitMat operations among threads). If BitMat has 100000000 rows and no of threads is 8, then thread 1 will operate on rows in range 1-12500000, thread 2 on 12500001-25000000 and so on

5. Future Work

As a part of the future work, we plan to change the native BitMat structure to save space when loaded in memory, and also use the techniques of bulk-loading and bulk-processing of the byte arrays in the BitMat rows instead of maintaining separate "character pointers" to them. We also aim to use bulk-loading in the pre-processing part of the database construction rather than inserting each element sequentially in the respective tree

6. Acknowledgements

We would like to thank Prof. Medha Atre for guiding us in understanding the BitMat system, data structures, and giving potential ideas of performance and space improvement.

7. References

1. https://www.w3.org/RDF/
2. https://www.w3.org/TR/rdf-sparql-query/
3. https://sourceforge.net/projects/bitmatrdf/
4. http://dl.acm.org/citation.cfm?id=1731354
5. http://grid.hust.edu.cn/triplebit/
6. http://www.icst.pku.edu.cn/intro/leizou/projects/gStore.htm
7. https://github.com/zcbenz/BPlusTree
8. http://www.cse.iitk.ac.in/users/atrem/papers/sigmod2015-atre.pdf
9. https://www.cse.iitk.ac.in/users/atrem/