**Efficient Ways to Process RDF graphs and SPARQL queries**

Dhawal Upadhyay, Gurkirat Singh

(third year BTech)

Dept of Computer Science and Engineering

Indian Institute of Technology Kanpur

{dhawal, gurkirat}@iitk.ac.in

[Resource Description Framework](https://www.w3.org/RDF/) (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](https://www.w3.org/TR/rdf-sparql-query/) 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](https://sourceforge.net/projects/bitmatrdf/), [RDF-3X](http://dl.acm.org/citation.cfm?id=1731354), [TripleBit](http://grid.hust.edu.cn/triplebit/), [gStore](http://www.icst.pku.edu.cn/intro/leizou/projects/gStore.htm) 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, the main contributions of our work are listed below:

1. We have built 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)](http://www.cse.iitk.ac.in/users/atrem/papers/sigmod2015-atre.pdf).
2. BitMat system’s query processor deals only with integer ID representation of RDF graph nodes and edges. Hence we have built a B+ tree of forward (string → ID) and reverse (ID → string) dictionary.
   1. In this B+ tree, we have used a novel method of using *hash* of the string/URIs as the ‘key’ to make compact and efficient storage. We use two independent hashes of 8 bytes each having a range of 2128. This practically precludes the possibility of any hash collision.
   2. We decompose the forward and reverse B+ trees into √n smaller trees where ‘n’ is the total number of records/keys to be inserted in the dictionary. We use multi-threading while building these smaller trees to reduce the construction time by ~75% (from 600s to 160s for 120 million mappings) using 8 cores.
   3. For reverse lookups, we use *offline* processing method: the BitMat core query processor first generates entire query results in the integer ID format alone. The B+ tree processor takes these results, sorts them, and then does bulk lookups on the tree and data file (parallelly). This makes it more efficient to process reverse lookups than doing one random lookup at a time.
3. Since BitMat’s *fold* and *unfold* operations do mutually exclusive operations on each row of the bit-matrix, we have parallelized these operations using multiple threads on the modern “multi-core” processors.

As a part of the future work, we plan to change the native BitMat structure to save space, and also use the techniques of bulk-loading and bulk-processing of the byte arrays in the BitMat rows.

**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.