Efficient processing of large RDF streams using memory management algorithms

Vaibhav Khadilkar, Murat Kantarcioglu, Latifur Khan, and Bhavani Thuraisingham

The University of Texas at Dallas

Abstract. As more RDF streaming applications are being developed, there is a growing need for an efficient mechanism for storing and performing inference over these streams. In this poster, we present a tool that stores these streams in a unified model by combining memory and disk based mechanisms. We explore various memory management algorithms and disk-persistence strategies to optimize query performance. Our unified model produces an optimized query execution and inference performance for RDF streams that benefit from the advantages of using both, memory and disk.

1 Introduction

An application that processes RDF streams does not know *apriori* the size of the stream. This makes it difficult to store these streams in memory as only a limited amount of data can be stored, and on the disk which requires a longer query processing time for small streams. Research has thus far focused on storing RDF data in relational databases [1] or in non-database approaches [2]. We have a developed a tool[1] that presents a unified model based on an optimal combination of memory and disk based solutions for storing RDF streams. This tool allows users to pose any SPARQL query (non-inference or inference) to an application. Our tool is implemented as a Jena graph which is the basic building block of the Jena framework. Our graph moves from Jena's in-memory graph to a Lucene[2] graph when we begin to run out memory. The Lucene graph mirrors the in-memory graph and queries to the unified model are rewritten to query the Lucene indices. We use different memory management algorithms to select nodes from the RDF stream to be left in memory based on the frequency of access patterns and the centrality of nodes in the stream. We have also tested two Lucene index creation strategies to optimize query performance. Finally, we switch to a Jena RDB graph when a threshold limit is reached, beyond which the RDB graph is better than Lucene for non-inference queries. Reference [3] presents an excellent survey on using memory management algorithms in relational databases. We have selected Lucene only as a temporary storage mechanism because we will switch to the RDB graph if more data is streamed. Our proposed model works very well for query execution and inference tasks with RDF streams.

2 Proposed Architecture

Figure 1 shows a flow of control to store RDF streams using the unified approach. We begin by storing the RDF stream in Jena's in-memory triple store.

[1] http://jena.sourceforge.net/contrib/contributions.html,http://cs.utdallas.edu/semanticweb
[2] http://lucene.apache.org/java/docs/index.html

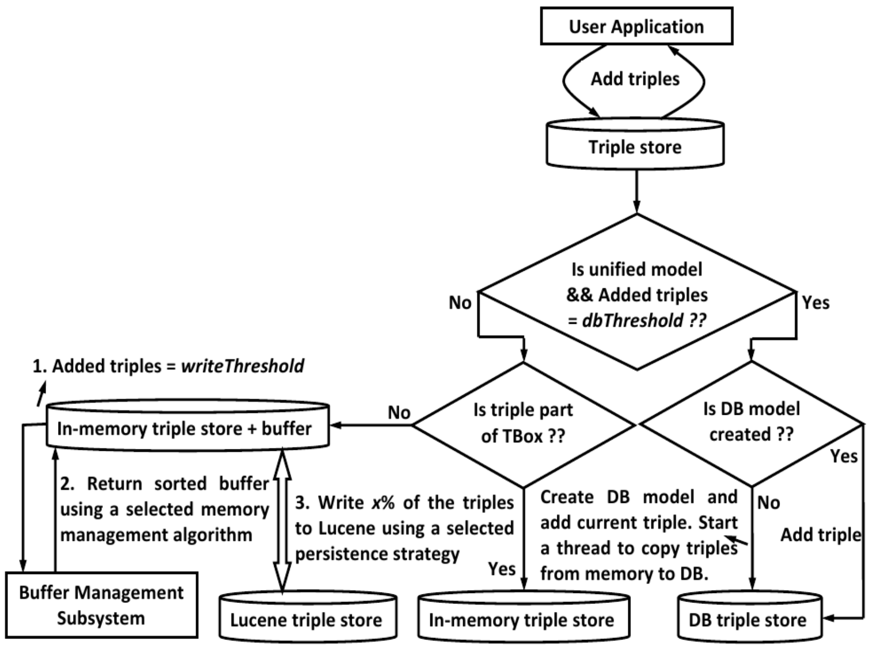
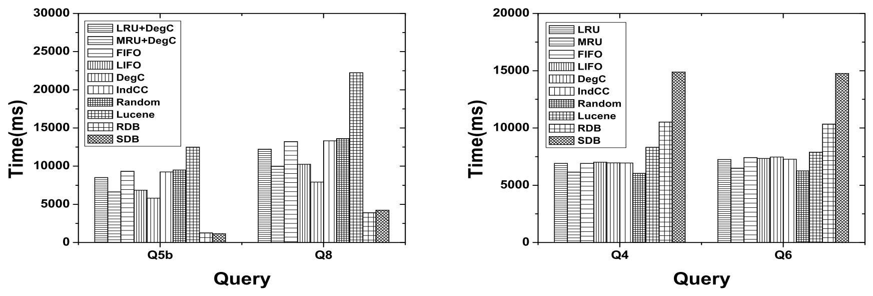

User Application
Add triples
Triple store
Is unified model && Added triples = *dbThreshold* ??
No
Yes
1. Added triples = *writeThreshold*
In-memory triple store + buffer
No
Is triple part of TBox ??
Is DB model created ??
Yes
2. Return sorted buffer using a selected memory management algorithm
3. Write *x*% of the triples to Lucene using a selected persistence strategy
Create DB model and add current triple. Start a thread to copy triples from memory to DB.
No
Add triple
Yes
Buffer Management Subsystem
Lucene triple store
In-memory triple store
DB triple store

Fig. 1. Unified Approach Architecture - Creating a model

If the triple is an ABox triple we also store or update for every *subject* of each incoming triple, it's *degree*, a *timestamp* of when it was last accessed and a *pointer* to the triples belonging to it, in a memory buffer. The TBox triples are first read into memory without maintaining any statistics for them in the buffer. This helps to distinguish TBox triples from ABox triples. Since no statistics are maintained these TBox triples are never written to disk. When the *writeThreshold* is reached the buffer management subsystem returns a sorted buffer based on the selected memory management algorithm such as FIFO, LIFO, LRU, MRU and RANDOM. We also adapted social network centrality measures such as degree centrality (DC) and clustering coefficient (CC) [4] into memory management algorithms. The *writeThreshold* is defined as, $writeThreshold = initThreshold \times totalMem$. The equation takes *initThreshold* as a number of triples and the memory size (specified in gigabytes) given to the current run from the user. Triples from the in-memory graph are moved to the Lucene graph using the *pointer* of every *subject* node (from the sorted buffer) and the selected persistence strategy. This process of moving triples continues as long as $x\%$ of the *writeThreshold* is not reached (x is also user configurable). Finally, when the *dbThreshold* is reached we move all triples to Jena's RDB graph. From this point onwards all incoming triples are directly stored in the RDB graph. We use a combination of in-memory, Lucene and RDB graphs for non-inference models and a combination of in-memory and Lucene graphs for inference models. For query execution, the input query is submitted to the graph that is currently being used. A non-inference query is run on either the in-memory and Lucene graphs or the RDB graph and a complete result is returned to the user. For an inference query, Pellet infers additional triples by reasoning over the result from the in-memory and Lucene graphs, using the TBox triples that are always in memory. The resulting triples are then returned to the user.

3 Experimental Results

We performed benchmark experiments to compare the performance of the unified model to both, the Jena database backends and a purely Lucene triple store. We used the Sp^{2}Bench [5] benchmark to check non-inference query execution and the LUBM [6] benchmark to test inference. Although we have tested all queries of both benchmarks on our system, in this section we show results only for Q5b and Q8 from Sp^{2}Bench and for Q4 and Q6 from LUBM as they are representative of the overall trend. The graphs below show only query time and they do not include loading times. We have also performed scalability tests with varying graph sizes, but in this poster we only show graph sizes of 50168 triples for Sp^{2}Bench and 1 university ($\approx$ 103000 triples) for LUBM. We use these small sizes since we only want to determine the best algorithm and Lucene persistence strategy in this paper. Further, we set $writeThreshold = (3/4) \times$ no. of triples in the graph, $totalMem = 1$ and $x =$ no. of triples in mem/90. We chose these values for the parameters so that we always have a good balance of triples between memory and Lucene giving us a good indication of the overall performance of various queries of both benchmarks.

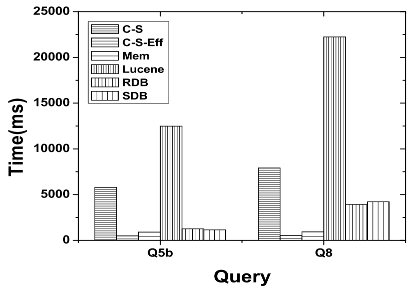


(a) Algorithms - Sp^{2}Bench (b) Algorithms - LUBM

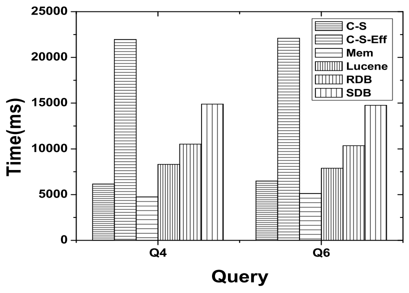
Fig. 2. Comparison of all algorithms and persistence strategies

We have used the *degree* and *timestamp* values to implement the memory management algorithms. For example, if we use LRU, we sort the buffer in the increasing order of *timestamp* values while for degree centrality we sort the buffer in increasing order of DC values. The reader should note that the DC and CC values are recomputed for every node each time the buffer is sorted. We then move triples to the Lucene graph for every node starting from the top of the buffer until $x\%$ of the triples are moved. We have also combined LRU and MRU with both DC and CC by first using the *timestamp* to sort the buffer and if there is a tie we use DC or CC to break the tie. Figure 2 shows a comparison of all memory management algorithms that we have tested for Sp^{2}Bench and LUBM.

For Sp^{2}Bench, we see that *DC* gives us the best result because it keeps nodes that are relevant to the Sp^{2}Bench queries in memory. In comparison, for LUBM, we see that MRU performs the best. This is due to the fact that MRU leaves the least recently used nodes in memory that are used by the Pellet reasoner for inference and query execution.



(a) Persistence strategies - Sp^{2}Bench



(b) Persistence strategies - LUBM

Fig. 3. Comparison of persistence strategies

We have also tested two Lucene persistence strategies, the first creates all indices at the same time (C-S) while the second creates each index as needed starting with the predicate, then the object and finally the subject (C-S-Eff). In the unified model we do not create the subject Lucene index, instead we set *dbThreshold* to be the number of triples at this point. The C-S strategy works well with LUBM queries but does not work for the Sp^{2}Bench queries as shown in figure 3. With C-S-Eff we get a query time comparable to in-memory storage for Sp^{2}Bench, but a much higher query time for LUBM. For LUBM, the Pellet reasoner needs to query the larger predicate Lucene index multiple times, making it slower than the C-S approach where the reasoner needs to query the smaller subject and object Lucene indices. The Sp^{2}Bench queries use the in-memory subject and object structures, and hence perform as well as the in-memory model.

4 Conclusion

In this paper we show that creating a unified model by combining the in-memory, Lucene and relational database models gives us excellent query execution and inference time with an enhanced scalability for RDF streams.

References

1. Dave Beckett. Scalability and Storage: Survey of Free Software/Open Source RDF storage systems. `http://www.w3.org/2001/sw/Europe/reports/rdf_scalable_storage_report/`, July 2002.
2. AllegroGraph RDFStore. `http://www.franz.com/agraph/allegrograph/`, 2005.
3. Hong-Tai Chou and David J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB*, pages 127–141, 1985.
4. M. O. Jackson. *Social and Economic Networks*. Princeton University Press, 2008.
5. Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^{2}Bench: A SPARQL Performance Benchmark. In *ICDE*, pages 222–233, 2009.
6. Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.