AWETO: Efficient Incremental Update and Querying in RDF Storage System

Xu Pu
Department of Computer Science and Technology, Tsinghua University
bertpu@gmail.com

Jianyong Wang
Department of Computer Science and Technology, Tsinghua University
jianyong@tsinghua.edu.cn

Ping Luo
HP Labs China
ping.luo@hp.com

Min Wang
HP Labs China
min.wang6@hp.com

ABSTRACT

With the fast growth of the knowledge bases built over the Internet, storing and querying millions or billions of RDF triples in a knowledge base have attracted increasing research interests. Although the latest RDF storage systems achieve good querying performance, few of them pay much attention to the characteristic of dynamic growth of the knowledge base. In this paper, to consider the efficiency of both querying and incremental update in RDF data, we propose a hAsh-based tWo-tiEr rdf sTORage system (abbr. to AWETO) with new index architecture and query execution engine. The performance of our system is systematically measured over two large-scale datesets. Compared with the other three state-of-the-art RDF storage systems, our system achieves the best incremental update efficiency, meanwhile, the query efficiency is competitive.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*Access methods*; H.2.4 [**Database Management**]: Systems—*Query processing*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms

Algorithms, Design, Performance

Keywords

AWETO, RDF, Index, Incremental Update, Query

1. INTRODUCTION

The RDF (Resource Description Framework) [5] data model, recommended by W3C, is widely adopted to represent the relationships in a knowledge base. With the much efforts in extracting relationships over the Internet in recent years, storing and querying these relationships, represented by the RDF triples, have attracted increasing research interests. Current research on large-scale RDF storage system mainly focuses on query efficiency of the system but considers little on incremental update, which is essentially required by the increasing size of knowledge bases. Therefore, considering the dynamic nature of the built knowledge base, a well-designed RDF storage system is extremely needed for fast incremental update and querying of RDF triples. In this paper, we propose a new RDF storage system AWETO (abbr. for a hAsh-based tWo-tiEr rdf sTORage system) with new index architecture and query execution engine which considers the efficiency of both querying and incremental update. In AWETO, a hash-based string-ID mapping strategy is firstly developed which maps the string representation of triples to their ID representation in a hash manner. Secondly, instead of creating big clustered B+ tree indices for all triples, we group the triples according to different atoms and different roles (subject, predicate or object) of atoms to create a two-tier index. The above two designs of the index architecture benefits the incremental update procedure, in the meantime the query performance is also competitive. Due to our new index architecture, new query execution engine is developed for fast access of the index. Note that incremental update only refers to incremental bulk load of the RDF triples in this paper, and other update operations such as triple deletion are for future work.

2. STRING-ID MAPPING APPROACH

Before the creation of the ID-based triple index, we convert the triples to their ID representation. Most of the RDF storage systems do the transformation because it decreases the index size and improves query efficiency.

Typically, while performing incremental update, the RDF storage system will look up each string in the input data to generate ID-based triples. This approach will cause vast disk look up because each string in the input data file need to be looked up in the in-disk string-ID mapping table, which causes performance degradation. We adopt a hash-based approach in our system, however it is different from those in previous work. We use an in-disk ID-to-string mapping table and an in-memory *conflict map* rather than only the in-disk

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

string-ID mapping table to efficiently resolve the string-ID mapping. With the in-memory *conflict map*, our approach can efficiently resolve the string-ID mapping in a fast and bulk manner, which cannot be accomplished in previous work.

During initial bulk load, we generate a temp file which contains all different strings with their IDs and the file is sorted by IDs. The IDs in the file are generated by the hash lookup algorithm [1]. Then, for each (*id*, *string*) pair in the temp file, we determine whether *id* equals to the ID in the previous pair in the file. If it does not, we add the pair into an in-disk ID-to-string mapping table (denoted as $itsTable$). Otherwise, we know that the *id* has already been assigned to another string. On this occasion, we use a *conflict map* (denoted as $conflictMap$) to record all such strings, which is called *conflict string*. Another ID $newID$ is assigned and the ($newID$, *string*) pair is added both to the $itsTable$ and $conflictMap$.

When an ID-to-string look up is needed, we use $itsTable$ to look up strings. In our current implementation, we adopt the Tokyo Cabinet [4] B+ tree as the ID-to-string mapping table. To improve the performance of $itsTable$, a *Bloom filter* [9] is added above $itsTable$. For *conflict map*, the number of *conflict strings* is small when a good hash function, is adopted even for large RDF data, thus, the *conflict map* can be located in memory for fast access.

For the sequential string-ID mapping approach, a string-to-ID mapping table must exist in disk to get the ID of a specific string. However, in our approach, the in-disk string-to-ID mapping is not essential which saves a lot of disk space. When we need to convert a string s into its ID representation, we first look up s in $conflictMap$. If $conflictMap$ contains s, we naturally get the ID, otherwise, $h(s)$ is computed. If we are sure that s exists in the string-ID mapping, $h(s)$ is naturally the ID representation for s. It can be used for generating ID-based triples after the string-ID mapping. If we are not sure the existence of s, $h(s)$ is looked up in $itsTable$. If we get exactly the string s by the look up, we can conclude that $h(s)$ is the ID representation of s, otherwise, we know that s does not exist in the string-ID mapping.

Differently from initial bulk load, during incremental update, the *conflict strings* cannot be retrieved by the temp file mentioned above because there have already existed many strings in the string-ID mapping. In this case, we generate a temp file which only contains all different strings of the new triples to be inserted, sorted by $h(s)$. Then, for each string s in the temp file, we look up if s is contained by string-ID mapping. If it is, nothing will be done, otherwise, we need to check if s is a new *conflict string*. If it is, we assign a new unused ID $newID$ for it and add $(newID, s)$ into both $itsTable$ and $conflictMap$, otherwise, $(h(s), s)$ is added into $itsTable$.

Compared with the traditional ID assignment approaches, which looks up each string in the new triples in string-ID mapping, we propose a two-step string-ID mapping approach. The first step resolves all the strings to their ID representation which is depicted above; the second step performs a sequential scan of the input data and generates ID-based triples. Our approach achieves high efficiency for the following two reasons. Firstly, in the first step, the number of different strings is much smaller than that of all the strings in the source file, less I/O operation is required to look up the strings. Bloom filter is adopted which also decreases the number of I/O operations. Secondly, in the second step, all the strings in the new triples have been already resolved and added to our string-ID mapping. On this occasion, each string is looked up extremely fast because no I/O operation for looking up string-ID mapping is needed. We only need to perform look ups in the in-memory *conflict map* and calculate the hash values of the strings.

3. TRIPLE INDEX

Traditionally, several big clustered B+ trees are used to store different orders of the ID-based triples. During the incremental update procedure, large number of triples with their ID form will be inserted into the B+ trees which causes heavy maintenance burden. If the number of insertions could be decreased, it could improve the performance of incremental update. Based on this consideration, differently from previous work, we adopt a new two-tier index architecture. The upper tier is called *atom position index* (AP *index*) which is actually a small B+ tree index, and the lower tier is called *binary tuple index* (BT *index*) which adopts our own index strategy that can be efficiently maintained.

Our triple index consists of four different index orders: S-PO, P-SO, P-OS, and O-PS. The basic idea is to separate a triple into a single atom and a two-atom tuple, which is called *binary tuple*. The separated atom is stored in AP *index* with some information related to it. All the *binary tuples* associated with each separated atom are sorted and stored in BT *index*.

The AP *index* is a key-value store in disk which is implemented by B+ tree. For each atom that appears in each role of a triple, we add a data item into AP *index*. The key is a 9-byte (*flag, atom*) pair (*FA pair*). The first byte is a flag byte, which indicates not only the role of the atom, but also the order of the associated *binary tuples* in BT *index*. The following 8 bytes after the flag byte contain the ID representation of the atom. The value associated with the key contains all the (*position, length*) pairs which indicate where the *binary tuples* associated with the *FA pair* are stored in BT *index*.

The BT *index* is a file located in disk. We divide the disk space into *blocks*, which is the basic unit for allocating disk space in BT *index*. The length of a *block* should be small, especially for the orders of S-PO and O-PS, because *binary tuples* associated with different *FA pairs* use different *blocks*, if *block* size is big, too much *internal fragmentation* will be made. Note that, due to the small size of *block*, it is only used for allocating disk space, not used for reading and writing data in disk.

Although *blocks* may be incontinuous in disk, for each *FA pair*, the *blocks* associated with it can be seen as *logically continuous*. We group each n *logically continuous blocks* into a *segment*, where n is called *segment size*. To support fast query execution, for each *FA pair*, except storing (*position*, *length*) pair, we also store the range information of the *binary tuples* for each *segment* in the value field of the *FA pair*, which is used by our query execution engine. *Segment* is treated as the basic unit of operating on the indexed data in BT *index*, including data compression/decompression, atom filtering (discussed in Section 4) and U-SIP [13] pruning.

Compression is needed for both the AP *index* and BT *index*. For AP *index*, we adopt the famous *variable-byte coding* [17] for the value field of AP *index*. For BT *index*,

instead of storing the original *binary tuples*, we adopt our own compression scheme which is similar to [12]. We omit the details for space limitation.

When performing incremental update, we sort the new ID-based RDF triples into four different orders, i.e., SPO, PSO, POS and OPS to get the *FA pairs* and sorted *binary tuples* associated with them. For each of the *FA pairs* and the associated *binary tuples* bt_{new}, we get the *binary tuples* bt_{old} which have the same role and atom with bt_{new} in triple index. Then we perform a merge union operation to merge the bt_{old} and bt_{new}. After that, we write the merged data into disk using the space that has already been allocated to bt_{old}. If the space is not sufficient, new space will be allocated. If the old *AP index* does not contain the new *FA pair*, we directly allocate new space and write bt_{new} into disk. Finally, we update the value field of *AP index*.

4. QUERY EXECUTION

Our query execution engine executes queries in a pipelining way; query is executed by an *operator tree*. Each operator gets part of data from its children and achieves its own logic. Due to our new index architecture, the *index scan operator* is different from that in previous work. In our system, the *index scan operator* reads and decompresses data from disk by *segment*. We take the S-PO order for example. There are four kinds of *simple access patterns* (SAPs) related to S-PO order: $(s, ?p, ?o)$, $(s, p, ?o)$, (s, p, o), and $(?s, ?p, ?o)$.

For SAPs like $(s, ?p, ?o)$, we firstly look up the *AP index* using atom s, then according to the positions and lengths information got from *AP index*, the bindings for $?p$ and $?o$ can be naturally retrieved. For $(s, p, ?o)$ and (s, p, o), we also firstly look up *AP index* using s. However, among all the bindings of $(s, ?p, ?o)$, we must get the bindings that satisfy $?p = p$ for $(s, p, ?o)$ or $?p = p, ?o = o$ for (s, p, o). Here, we adopt an efficient *atom filter* to accomplish this task. It first filters in the *segment* level and skips useless *segments* according to the range information before reading the *segments*, then only the data in the first and last read *segments* need to be filtered after the skips. At last, for the special case $(?s, ?p, ?o)$, we perform a full scan on the *AP index* to get all the bindings of $?s$. Then for each binding of $?s$, the operation is the same as $(s, ?p, ?o)$.

5. EXPERIMENTAL EVALUATION

In order to evaluate the performance of our system, we compared both the query runtime and incremental update efficiency to other systems. All the experiments were done on an IBM System x3650 server (eight 1.66GHz CPU cores, 20GB memory) with 64-bit Linux. Two datasets are used, YAGO2 [6] (the core version) and LUBM [11]. YAGO2 contains 15,820,985 different strings and 32,393,226 different triples. For LUBM, we generated a dataset which consists of 500 universities by the UBA 1.7 data generator with $index = 0$ and $seed = 0$. The dataset contains totally 16,439,335 different strings and 66,751,196 different triples. Because LUBM is a synthetic dataset, the triple order in it has fixed and regular pattern. Thus, we disorganized the triple order in the dataset to make it more natural. In all the tests, the *block* size is set to 32 bytes. *Segment size* is set to 32 for YAGO2 and 512 for LUBM.

The RDF-3X system introduced in [12, 13, 14, 15] is the primary competitor of our system because it has achieved

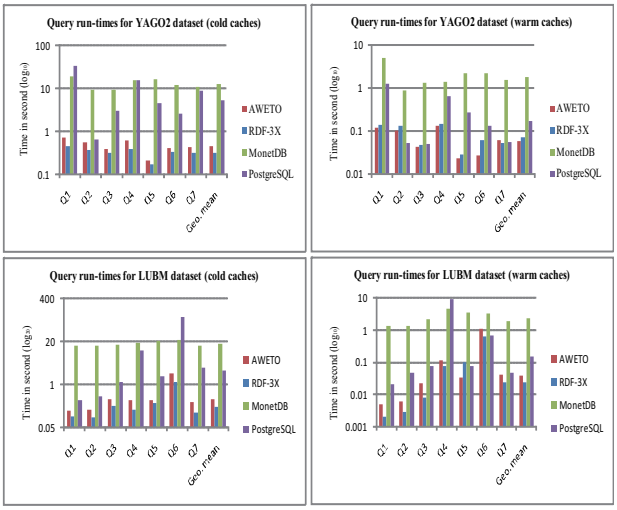

Figure 1: Query run-times evaluation

the best performance among all the current RDF storage systems. The second baseline system is column-store-based vertical partitioning approach introduced in [8, 7], which has gained the best performance among all other approaches based on DBMS. Differently from vertical partitioning approach, we used MonetDB [2] instead of C-Store [16] as the underlying column store suggested by the authors of C-Store. The third baseline is the PostgreSQL [3] database system acting as *triple store*. The indices were built with the orders SPO, PSO and POS which belong to the Sesame-style storage system [10].

The query evaluation for YAGO2 and LUBM are shown in Figure 1. For YAGO2, RDF-3X storage system achieves the best performance in cold caches and our system performs the best in warm caches. In cold caches, RDF-3X outperforms our system by an average factor of about 1.4, while in warm caches, our system performs better than RDF-3X by an average factor of about 1.2. The main reason for the slowness of our system in cold caches is that our system takes both querying and incremental update into consideration. The big ID range of hash-based string-ID mapping influences the compression efficiency of *binary tuples*, thus slows down the I/O read time. Our system performing well in warm caches benefits from our well-designed query execution engine. For MonetDB and PostgreSQL, our system achieves much higher efficiency than the two systems. Our system outperforms PostgreSQL by an average factor of 11.8, sometimes by more than 47.1 in cold caches and an average factor of 2.9, sometimes by more than 12.2 in warm caches. Differently from performance reported in [8], MonetDB performs the worst because we have included the time for converting the string representation of atoms in the SPARQL query into their ID representation. MonetDB consumes much of the time for this conversion which has dominated the query time.

LUBM has similar result. RDF-3X storage system achieves the best performance and our system is the runner-up. RDF-3X outperforms our system by an average factor of about 1.7 in cold caches and about 1.5 in warm caches. For Post-

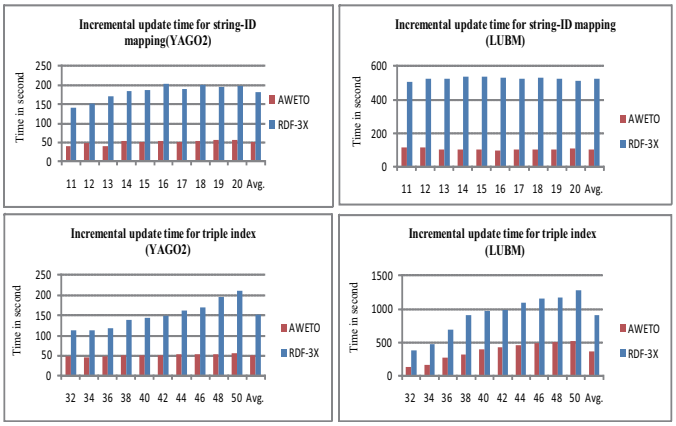

Figure 2: Incremental update evaluation

greSQL, our system outperforms PostgreSQL by an average factor of 7.7, sometimes by more than 49 in cold caches and an average factor of 4, sometimes by more than 78 in warm caches. For MonetDB, string-to-ID mapping also dominates the whole query time.

To evaluate incremental update efficiency, for all the systems, we take the N3 file as input. Our system has full ability to accomplish incremental update. For RDF-3X system, although the authors have implemented incremental update in their system, it cannot work well according to our experiment. Furthermore, our system and RDF-3X uses different programming language and B+ tree implementation. To make a fair comparison, we implemented the basic idea of incremental update procedure of RDF-3X in Java and used Tokyo Cabinet which is the same as AWETO and tried our best to make the code the most efficient. We reduced the number of triple indices in RDF-3X from fifteen to four, which makes RDF-3X have the same number of indices with our system. For MonetDB and PostgreSQL, we found the incremental update operation is much slower than that of our system and RDF-3X, thus we only report the incremental update performance of our system and RDF-3X.

We assume the following scenario of incremental update. For YAGO2 dataset, the initial size of the knowledge base is set to 10 million triples. We repeat 10 batches of incremental update, 1 million triples in each batch. For LUBM we set the initial size to 30 million triples and repeat 10 batches of incremental update, 2 million triples per batch.

The experimental results of string-ID mapping strategy and triple index for our system and RDF-3X are shown in Figure 2. The X-axis indicates the number of triples (in million) in the knowledge base after the incremental update. For both the string-ID mapping and triple index, our system achieves the best incremental update efficiency and outperforms RDF-3X a lot with both of the two datasets. For string-ID mapping, our system outperforms RDF-3X by an average factor of 3.638/4.962 by adopting our hash-based String-ID mapping approach using YAGO2/LUBM dataset. Also, our system outperforms RDF-3X by a factor of 2.964/2.453 by adopting our triple index using YAGO2/LUBM. Therefore, both our hash-based String-ID mapping approach and the triple index are optimized for incremental update and more time-efficient compared with previous work.

6. CONCLUSION

In this paper, we propose a new RDF storage system AWETO which considers both the performance of querying and incremental update. For string-ID mapping, we adopt a hash-based approach with in-memory *conflict map* which achieves high performance in incremental update. For triple index, a new two-tier index approach is proposed which optimizes the incremental update efficiency. For query execution, our highly-efficient operators achieve high efficiency while performing queries. Experimental results show that our system is competitive in querying and outperforms all other three state-of-the-art storage systems when performing incremental update.

7. ACKNOWLEDGMENT

This work was supported in part by the National Basic Research Program of China (973 Program) under Grant No. 2011CB302206, National Natural Science Foundation of China under grant No. 60833003 and 60873171, an HP Labs Innovation Research Program award, and the Program for New Century Excellent Talents in University under Grant No. NCET-07-0491, State Education Ministry of China.

8. REFERENCES

[1] A hash function for hash table lookup. http://burtleburtle.net/bob/hash/doobs.html.
[2] MonetDB. http://monetdb.cwi.nl.
[3] PostgreSQL: the world's most advanced open source database. http://www.postgresql.org.
[4] Tokyo cabinet: a modern implementation of DBM. http://fallabs.com/tokyocabinet.
[5] W3C: Resource description framework(RDF). http://www.w3.org/RDF.
[6] YAGO2. http://www.mpi-inf.mpg.de/yago-naga/yago.
[7] D. J. Abadi, A. Marcus, S. Madden, and K. Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *VLDB J.*, 18(2):385–406, 2009.
[8] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422, 2007.
[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
[10] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying rdf data and schema information. In *Spinning the Semantic Web*, pages 197–222, 2003.
[11] Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.
[12] T. Neumann and G. Weikum. Rdf-3x: a risc-style engine for rdf. *PVLDB*, 1(1):647–659, 2008.
[13] T. Neumann and G. Weikum. Scalable join processing on very large rdf graphs. In *SIGMOD Conference*, pages 627–640, 2009.
[14] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.
[15] T. Neumann and G. Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *PVLDB*, 3(1):256–263, 2010.
[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.
[17] H. E. Williams and J. Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999.