Data & Knowledge Engineering 89 (2014) 55–75

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Editorial

Efficient incremental update and querying in AWETO RDF storage system

Xu Pu [a,*], Jianyong Wang [a], Zhenhua Song [a], Ping Luo [b], Min Wang [c]

[a] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[b] Institute of Computing Technology, Chinese Academy of Sciences, No. 6 Kexueyuan South Road, Zhongguancun, Haidian District, Beijing 100190, China
[c] HP Labs China, SP Tower A505, Tshinghua Science Park, Building 8, Beijing 100084, China

ARTICLE INFO

Article history:
Received 14 March 2012
Received in revised form 13 November 2013
Accepted 13 November 2013
Available online 25 December 2013

Keywords:
Indexing methods
AWETO
RDF
Incremental update
Query

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. Since the building of the knowledge base is usually a continuous process, incremental update over the RDF storage system is in great need. 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 datasets. Compared with the other three state-of-the-art open source RDF storage systems, our system achieves the best incremental update efficiency meanwhile, the query efficiency is competitive.

1. Introduction

The RDF (Resource Description Framework) [1] data model, recommended by the W3C, is widely adopted to represent the relationships in a knowledge base. For the past few years, a large amount of relationships (facts) have been extracted and transformed into RDF triples for further use. For example, millions of relationships were extracted mainly from Wikipedia [2] to form DBPedia [3]; relationships derived from Wikipedia, WordNet [4] and GeoNames [5] form the knowledge base YAGO2 [6,7]. Additionally, RDF enables merging and sharing across different applications. The Linking Open Data Project [8] collects various open datasets and establishes interlink among them, which enables users to easily navigate among the datasets. With the many efforts conducted in extracting relationships over the Internet, 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. Usually, building a modern knowledge base is a continuous process. Suppose an international enterprise establishes its own knowledge base according to the current information about employees, products, etc. The information will change everyday due to new employees on board, product release, and so on. These changes require efficient incremental update strategy of the knowledge base. Moreover, take the YAGO2 project for example. Currently, YAGO2 includes millions of entities and relationships extracted from Wikipedia. As time goes on, many new pages will be added into Wikipedia and new RDF triples will be extracted from the new pages. Then, an incremental update operation is required to load the newly extracted triples into the existing knowledge base.

Currently the existing RDF data storage systems can be divided into two types. The first type relies on the underlying relational database system and stores all triples in it with specific form. When executing an RDF query, the query is first converted to SQL

* Corresponding author.
E-mail addresses: bertpu@gmail.com (X. Pu), jianyong@tsinghua.edu.cn (J. Wang), songzh11@mails.tsinghua.edu.cn (Z. Song), ping.luoul@gmail.com (P. Luo), min.wang6@hp.com (M. Wang).

0169-023X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.datak.2013.11.003

automatically or manually, then the SQL query is used for querying the results from DBMS. This kind of RDF storage system naturally achieves the ability of incremental update by the underlying database system. However, the efficiency of incremental update is not good for most of the systems. The other type of RDF storage systems establishes their own storage and index architecture and directly queries data using RDF query languages. Most of the previous work takes query efficiency as the most important criteria for optimization and does not take much consideration on incremental update. Recently, SW-Store [9] supports incremental update in their system but performance is not reported. gStore [10] and RDF-3X [11–14] take incremental update as a feature in their system and their incremental update performance has also been evaluated.

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. Differently from previous work, we recently proposed 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 and optimizes the architecture for incremental update. A poster paper has depicted its basic ideas [15]. Now, we give a complete and detailed description of AWETO. 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 our index.

We summarize the contributions of our work as follows:

- To our best knowledge this is the first work to emphasize and optimize the incremental update feature of RDF storage systems to address the dynamic nature of knowledge bases.
- A hash-based string-ID mapping strategy and a two-tier triple index architecture are designed and developed in our system, which help achieve high incremental update rate. Incorporating with our well-designed query execution engine, our system also achieves high query efficiency.
- Systematic experimental study over two large-scale datasets was conducted to compare our system with other three open source state-of-the-art RDF storage systems. Our system achieves the best incremental update efficiency meanwhile, the query efficiency is very competitive.

The remainder of this paper is organized as follows. In Section 2, we introduce RDF and SPARQL. Our hash-based string-ID mapping strategy is introduced in Section 3. Triple index is discussed in Section 4. Then we introduce the query execution engine of our system in Section 5. The overall system is evaluated in Section 6. Related work will be discussed in Section 7, and we conclude the paper in Section 8.

2. Preliminaries

In this section, we briefly introduce the RDF data model and SPARQL query language which are widely used in Semantic Web.

RDF data model is used for representing *resources* and relationships between resources in *Semantic Web*. Its intention is to provide a mechanism to describe the resources in the Internet, which makes the resources understood by computer in a semantic manner. RDF provides a simple data model to represent resources and relationships between pairs of resources. When all the people follow such model, it is much easier for sharing information among multiple applications.

In RDF data model, we call an entity a resource. In order to describe relationships between pairs of resources, RDF introduces the concept of *property*. A resource can have multiple properties, and properties have values. Take the fact "Alan Turing was born on 23th June, 1912" for example. In order to represent the relationship "born on" between "Alan Turing" and "23th June, 1912", we can define a property "<*bornOnDate*>" and such relationship can be represented by (<*Alan_Turing*>, <*bornOnDate*>, "*1912-06-23*").

We call such *triple* a *statement*. The resource to be stated is called *subject*, the property in the relationship is called *predicate* and the value of the property is called *object*. Thus, differently from the traditional entity-relationship model, in RDF data model, data are represented as (*subject*, *predicate*, *object*) triples. We call each role in a triple an *atom*. Each triple represents a relationship between subject and object, and the type of the relationship is represented by predicate. In most cases, subject is a resource. In RDF, we use *URI reference* to uniquely represent a resource. Predicate is also a resource. Generally, object can be either a *literal* or resource. Literal is not a resource; it is a string value (e.g., "*1912-06-23*").

Following are some triples related to Alan Turing[1]:

(<*Alan_Turing*>, <*bornInLocation*>, <*London*>)
(<*Alan_Turing*>, <*bornOnDate*>, "*1912-06-23*")
(<*Alan_Turing*>, <*diedInLocation*>, <*Wilmslow*>)
(<*Alan_Turing*>, <*diedOnDate*>, "*1954-06-07*")
(<*Alan_Turing*>, <*hasWebsite*>, <http://www.alanturing.net>)
(<*Alan_Turing*>, <*isCalled*>, "*Alan M. Turing*").

[1] The full URI is omitted for simple presentation.

There is also a special kind of subject (object) in RDF, which is called *blank node*. Blank node is an anonymous resource. In general, we define a resource by its URI reference; however some subjects (objects) are only used to link up two relationships and we do not care about their URI references. In such situation, blank nodes are used to represent the anonymous subjects (objects).

RDF data can be seen as a large directed graph where subjects and objects are represented as nodes and predicates as directed edges. The SPARQL (*SPARQL Protocol and RDF Query Language*) Query Language [16] is used for expressing queries of RDF data. Its basic idea is graph pattern match. Now we briefly introduce some important concepts in SPARQL. A string beginning with a question mark is called a *variable*. A *Simple Access Pattern* (SAP) is a triple where zero or more atoms are replaced by variables. For instance, the following SAP can be used to query all the resources who were born in London: (*?p*, <*bornInLocation*>, <*London*>).

Here *?p* is a variable. For each triple that satisfies the pattern, its value of the subject is called a *binding* of the variable *?p*. SAP can only handle simple queries, for complicated ones, *Basic Graph Pattern* (*BGP*) is introduced. A basic graph pattern is a conjunction of one or more SAPs. For example:

?p <*bornInLocation*> <*London*>.
?p <*diedInLocation*> <*Wilmslow*>.

In the above example, variable *?p* appears in two SAPs, which means the two SAPs are conjunct via variable *?p*. Therefore, the above BGP queries resources who were born in London and died in Wilmslow. SPARQL query is based on BGP. In SPARQL, the SELECT clause identifies the variables that appear in the query results, and the WHERE clause provides the BGP to match against the data graph. If we rewrite the above query with SPARQL, it will be:

SELECT *?p* WHERE {
?p <*bornInLocation*> <*London*>.
?p <*diedInLocation*> <*Wilmslow*>.}

3. 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. This section describes the proposed String-ID mapping approach of AWETO.

String-ID mapping of an RDF storage system provides the following two functionalities:

- *String-to-ID mapping*: Given a SPARQL query, it transforms the string representation of atoms in the SPARQL query into their ID representation. Furthermore, while performing incremental update, each string in the triples will be looked up in the string-ID mapping to test if the string has already been assigned an ID.[2]
- *ID-to-string mapping*: Given the result of a query generated by the query execution engine, it transforms the ID representation of atoms into their string representation.

Typically, two kinds of string-ID mapping approaches are adopted in the current RDF storage systems.

The first approach is a simple sequential approach which is adopted in many RDF storage systems. The system sequentially assigns a new ID when each new string appears. Two tables are established in disk, namely, string-to-ID mapping table and ID-to-string mapping table (or database table with indices built on both the ID column and string column), to provide the above two functionalities. In the incremental update phase, the system will look up each string in the input data to generate ID-based triples. This approach will cause vast disk access because each string in the input data needs to be looked up in the string-to-ID mapping table, which causes performance degradation.

The second approach is the hash-based approach, adopted by Oracle [17] and 3store [18]. In this approach, each string is hashed by a specific hash function and a database table is used for storing the mapping. We also adopt hash-based approach in our system; however it is different from that introduced in [17] and [18]. We use an in-disk ID-to-string mapping table and an in-memory *conflict map* rather than single relational database 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.

Before we introduce our string-ID mapping approach, we first define *conflict string*:

Definition 1. Conflict string

Given a string s and a hash function $h(s)$ which maps a string to an ID, we call string s a conflict string if there exists another string $s' \neq s$ that has already been assigned the ID $h(s)$.

We establish an in-memory hash table which maps a string to an integer to store all the *conflict strings* with their ID representation. The table is called *conflict map*. Next, we will introduce how the *conflict map* is used for efficient string-ID mapping in both initial bulk load and incremental update.

[2] For convenient presentation, we use the word "string" to indicate the string representation of an atom and "ID" to indicate the ID representation of an atom in the following sections.

3.1. String-ID mapping in initial bulk load

During initial bulk load, we generate a temporary 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 [19]. It is reported in [17] that the hash lookup algorithm performs the best among several state-of-the-art hash algorithms considering both hashing speed and hashing quality. Our proposed algorithm for resolving string-ID mapping in initial bulk load is depicted in Algorithm 1.

Algorithm 1: Resolving string-ID mapping in initial bulk load

Input: The temporary file $file$ that contains all different strings sorted by IDs;
The ID-to-string mapping table $itsTable$;
The conflict map $conflictMap$.

```
1  initialize itsTable;
2  initialize conflictMap;
3  lastID ← −1;
4  foreach (id, string) pair isPair ∈ file do
5      if isPair.id ≠ lastID then
6          insert isPair into itsTable;
7          lastID ← isPair.id;
8      else
9          insert isPair into conflictMap;
10 foreach (id, string) pair isPair ∈ conflictMap do
11     repeat
12         generate a new ID newID for string;
13     until itsTable does not contain newID ;
14     insert (newID, string) into itsTable;
15     replace the ID field of isPair in conflictMap to newID;
```

For each (*id*, *string*) pair in the temporary file, we determine whether *string* is a *conflict string* by judging if *id* in the previous (*id*, *string*) pair equals to that in this pair (line 5). If *string* is not a conflict one, we add the pair into *itsTable* (line 6), otherwise, the pair is added to *conflictMap* for further operations (line 9). After the scan of the file, for each (*id*, *string*) pair in *conflictMap*, we generate a new ID for each pair and then modify the *conflictMap* to ensure there does not exist any different two strings with the same ID (lines 10–15).

Notice that, in line 13, we need to look up the *itsTable* given a specific ID which may cause I/O operation. We add a *Bloom filter* [20] above the *itsTable*. Bloom filter is a space-efficient data structure and is used to test whether an element is contained in a set with a probabilistic way. It allows false positives but does not allow false negatives, i.e., in our scenario, if the containment test of the bloom filter for a specific ID returns "contain", we must double-check if the ID really exists, however if the test returns "not contain", we can safely conclude that the ID is not contained by *itsTable* which saves much I/O time. In the current implementation of AWETO, we use one hash function $h(x) = x \bmod m$, where m is the length of the bit array in the *Bloom filter*.

Now we give an example to illustrate our string-ID mapping approach. Assume we need to insert the following 15 triples into AWETO:

(<*Alan_Turing*>, <*type*>, <*Person*>)
(<*Alan_Turing*>, <*isCalled*>, "*Alan M. Turing*")
(<*Alan_Turing*>, <*isCalled*>, "*Alanus Mathison Turing*")
(<*Alan_Turing*>, <*isCalled*>, "*Alan Turing*")
(<*Alan_Turing*>, <*hasGivenName*>, "*Alan*")
(<*Alan_Turing*>, <*hasFamilyName*>, "*Turing*")
(<*Alan_Turing*>, <*graduatedFrom*>, <*Princeton_University*>)
(<*Alan_Turing*>, <*hasAcademicAdvisor*>, <*Alonzo_Church*>)
(<*Princeton_University*>, <*type*>, <*Universitiy*)
(<*Princeton_University*>, <*isCalled*>, "*Princeton-universiteit*")
(<*Princeton_University*>, <*isCalled*>, "*Universitas Princeton*")
(<*Alonzo_Church*>, <*type*>, <*Person*>)
(<*Alonzo_Church*>, <*hasGivenName*>, "*Alonzo*")
(<*Alonzo_Church*>, <*hasFamilyName*>, "*Church*")
(<*Alonzo_Church*>, <*isCalled*>, "*Alonzo Church*")

Then, we generate a temporary file containing the following information:

325139335 Princeton-universiteit
620828401 Alanus Mathison Turing

795117782 type
922484582 Church
922484582 Alonzo_Church
975184447 graduatedFrom
1019901428 Princeton_University
1065738113 Universitas Princeton
1124849173 hasAcademicAdvisor
1273271076 Alan Turing
1330155065 Universitiy
1402263080 Alonzo Church
1439869607 Alan
1439869607 Alonzo
1801982372 Alan_Turing
1887385039 hasGivenName
1918674024 Person
1959756650 hasFamilyName
2040294948 Alan M. Turing
2108943609 isCalled
2138869059 Turing

In each line shown above, the integer is computed by hash lookup algorithm. We can easily find that *Alonzo_Church* is a *conflict string* because another string *Church* has the same hash value with *Alonzo_Church* and the hash value *922484582* has already been assigned to *Church* as its ID. Thus we generate a new ID *953038171* which has not been assigned to any string. For the same reason, *conflict string Alonzo* is assigned a new ID *1651651253* as well. After scanning the temporary file, the ID-to-string mapping table and the *conflict map* are shown in Fig. 1.

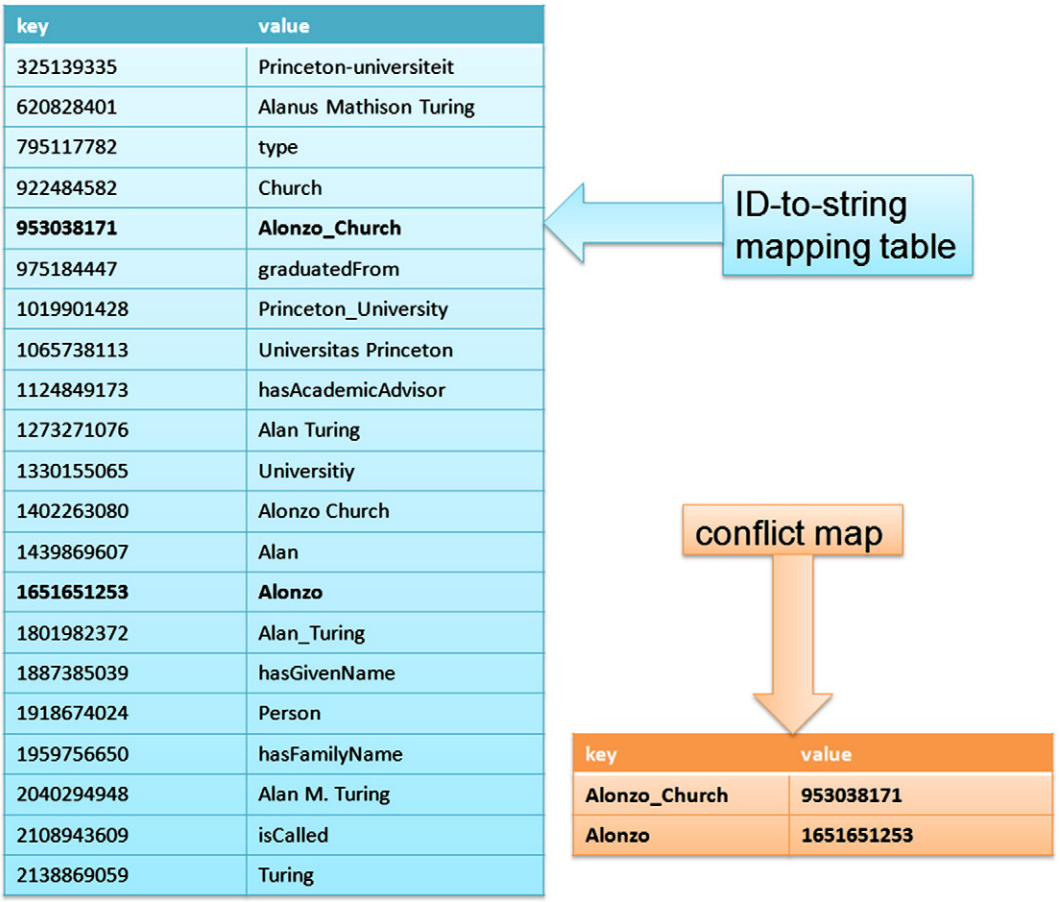

Fig. 1. Structure of AWETO string-ID mapping.

In conclusion, two tasks are done during Algorithm 1:

1. All the ID-to-string mappings are written to the ID-to-string mapping table in disk.
2. All the *conflict strings* are assigned new IDs and inserted into both the ID-to-string mapping table and the *conflict map*.

The first task is to construct an in-disk ID-to-string mapping table. When an ID-to-string look up is needed (e.g., convert the query results generated by the query execution engine into their string representation), we use this table to look up strings. In our current implementation, we adopt the Tokyo Cabinet [21] B+ tree as the ID-to-string mapping table. The second task constructs an in-memory *conflict map* which contains all the strings and their corresponding IDs that are not generated normally by the hash function $h(s)$. Even for large RDF data, the number of *conflict strings* is small when a good hash function is adopted. According to [17], for UniProt [22] dataset with more than 33 million different strings, the authors adopted a 63-bit hash lookup algorithm and got no collisions. In our implementation, we adopt a 31-bit hash lookup algorithm. The reason to use shorter bits to represent IDs is that we consider both the conflict number and the compression rate. Our compression (detailed in Section 4.1) is based on the difference values between adjacent triples. Too large ID range will cause degradation of compression rate because it will enlarge the difference values between adjacent ID values. With 31-bit length, the number of *conflict strings* is 58,219 for YAGO2 and the total number of different strings in YAGO2 is 15,820,985. For LUBM, the number of *conflict strings* is 62,485 while the total number is 16,439,335.[3] Thus, the number of *conflict strings* in these large-scale datasets is small enough to locate in memory.

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 table 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 the *conflict map*. If the *conflict map* 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 from the source RDF data file after the string-ID mapping. If we are not sure of the existence of *s*, $h(s)$ is looked up in the ID-to-string mapping table. 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.

3.2. String-ID mapping in incremental update

Firstly, we discuss incremental insertion. Differently from initial bulk load, during incremental insertion, the *conflict strings* cannot be retrieved by the temporary file mentioned in the preceding section because there have already existed many strings in the string-ID mapping. Thus, different strategies are adopted. We generate a temporary file which only contains all different strings of the new triples to be inserted, sorted by $h(s)$. Then we perform a single scan of the file to finish the resolution of string-ID mapping. Our algorithm for resolving string-ID mapping in incremental insertion is depicted in Algorithm 2.

Algorithm 2: Resolving string-ID mapping in incremental insertion

Input: The temporary file $file$ that contains all different strings, sorted by $h(s)$;
The ID-to-string mapping table $itsTable$;
The conflict map $conflictMap$.

```
1  foreach string ∈ file do
2      if conflictMap does not contains string then
3          stringFromMap ← itsTable.get(h(string));
4          if stringFromMap = null then
5              insert (h(string), string) into itsTable;
6          else if string ≠ stringFromMap then
7              repeat
8                  generate a new ID newID for string;
9              until itsTable does not contain newID ;
10             insert (newID, string) into itsTable;
11             insert (newID, string) into conflictMap;
```

Here, $h(s)$ is the same hash function as in initial bulk load phase. For each *string* in the file, the algorithm first checks if the string is contained by the *conflict map* (line 2). If it is, nothing will be done for the string because it has already been assigned an ID. Otherwise, as described in line 3, we get the *stringFromMap* from *itsTable* given the hash value of the *string* $h(string)$. If $h(string)$ cannot be found in *itsTable*, which means it is a new string but not a *conflict string* (line 4), we add it into *itsTable*(line 5). If $h(string)$ is found from *itsTable*, a double-check is needed because a different string may be assigned the ID $h(string)$ before, which indicates *string* is a *conflict string*. Here in line 6, we check if *string* equals to *stringFromMap*. If they do, nothing will be done because the *string* has already been assigned $h(string)$ as ID, otherwise, in lines 7–9, a new ID is assigned and the (*newID*, *string*) mapping will be added into both *itsTable* and *conflictMap* (line 10 and 11). Note that, in lines 3 and 9, we need to look up the *itsTable* given a specific ID. The same as initial bulk load, we also adopt the Bloom filter to reduce I/O operations.

[3] For detailed description of the two datasets, see Section 6.2.

After resolving the string-ID mapping using Algorithms 1 or 2, we need to transform the triples in the input data from string representation to ID representation. After that, we can use the ID-based triples to generate the triple index. We need to rescan the input data to generate the ID-based triples. Because all the strings in the input data are contained by the string-ID mapping, which is accomplished by Algorithms 1 or 2, no I/O operation for looking up the string-ID mapping is needed. Thus, the generation of the ID-based triples can be very fast.

Compared with the sequential ID assignment approach, which looks up each string in the new triples in string-ID mapping, our approach achieves high efficiency for the following two reasons. Firstly, when resolving string-ID mapping, by generating the temporary file, our approach looks up each different string in the string-ID mapping rather than all the strings in the input data, and the number of different strings is much smaller than that of all the strings, less I/O operation is required to look up the strings. In YAGO2 dataset, there are 32,393,226 different triples, thus it contains 97,179,678 strings. However, there are only 15,820,985 different strings in the dataset, which is about 16.3% of all the strings. Similarly, in LUBM dataset, there are 66,751,196 different triples, thus it contains 200,253,588 strings. However, there are only 16,439,335 different strings in the dataset, which is about 8.2% of all the strings. Furthermore, Bloom filter is adopted which also decreases the number of I/O operations. Secondly, when generating ID-based triples, 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.

For incremental deletion of the triples, currently, state-of-the-art RDF storage systems do not consider the deletion of strings in the string-ID mapping. RDF storage system is used as the serialization layer to store knowledge in semantic web. In most cases, knowledge frequently increases but rarely decreases. For instance, new knowledge will be added to Wikipedia everyday, but existing knowledge in Wikipedia is rarely removed. Thus, the strings in the string-ID mapping in an RDF storage system are rarely deleted. It will cause heavy maintenance burden if deletion is considered in string-ID mapping.

Thus, like other RDF storage systems, we do not consider the deletion of strings in the string-ID mapping either. When performing incremental deletion, for each string s in a triple to be deleted, if we cannot find it in the string-ID mapping, it indicates that the triple is not in the knowledge base. Therefore, we can ignore the triple and continue to the next one.

4. Triple index

After mapping the strings to their ID representation, we use ID-based triples to build our triple index. In this section, we discuss our two-tier triple index architecture. Overview of the triple index is introduced in Section 4.1, then we introduce the incremental update of triple index in Section 4.2.

4.1. Overview of triple index

Traditionally, several big clustered B+ trees are used to store different orders of all triples (SPO, PSO, POS, OPS, etc.). During the incremental update procedure, large number of triples with their ID form will be inserted into (deleted from) the B+ trees which causes heavy maintenance burden of B+ trees. If the number of insertions (deletions) 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*. Take P-OS order for example, the triple (*subject*, *predicate*, *object*) is separated into an atom *predicate* and a *binary tuple* (*object*, *subject*). The separated atom is stored in *AP index* with some information related to it. The *binary tuples* associated with each separated atom are sorted and stored in *BT index*. Next we will introduce *AP index* and *BT index* in detail.

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 of the data item is a 9-byte (*flag*, *atom*) pair (for simplicity, we call it *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*. In the current implementation, we define 0 for P-SO order (the role is predicate, and the order of *binary tuples* is sorting by subject first, then object), 1 for P-OS order, 2 for O-PS order, and 3 for S-PO order. The following 8 bytes after the flag byte contain the ID representation of the atom. The value field of the data item 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*. *Block* is the basic unit for allocating disk space in *BT index*. Recall that position and length information of a specific *FA pair* are stored in *AP index*. Here, *position* is the offset in *BT index* file, and *length* is the length of the indexed data measured in *block*.

Here, we use an example to demonstrate our two-tier triple index. Recall the example in Section 3.1. After string-ID mapping, the triples are converted into ID representation and we sort the triples by SPO order:

(*953038171*, *795117782*, *1918674024*)
(*953038171*, *1887385039*, *1651651253*)
(*953038171*, *1959756650*, *922484582*)
(*953038171*, *2108943609*, *1402263080*)
(*1019901428*, *795117782*, *1330155065*)
(*1019901428*, *2108943609*, *325139335*)

(*1019901428*, *2108943609*, *1065738113*)
(*1801982372*, *795117782*, *1918674024*)
(*1801982372*, *975184447*, *1019901428*)
(*1801982372*, *1124849173*, *953038171*)
(*1801982372*, *1887385039*, *1439869607*)
(*1801982372*, *1959756650*, *2138869059*)
(*1801982372*, *2108943609*, *620828401*)
(*1801982372*, *2108943609*, *1273271076*)
(*1801982372*, *2108943609*, *2040294948*).

To simplify the demonstration, we suppose one *block* contains two triples (actually, there are more triples in one *block*). For the first subject *953038171*, we have 4 triples, thus, 2 *blocks* need to be allocated to store them. Suppose we allocate *block* 0 and *block* 1. We firstly add a data item with key (3, 953038171) and value (0, 2) into *AP index*, then write (795117782, 1918674024), (1887385039, 1651651253), (1959756650, 922484582), (2108943609, 1402263080) into *BT index* beginning with offset 0. Similarly, triples with subject *1019901428* and *1801982372* will be also written to *AP index* and *BT index* in the same way. After this, the structure of the two-tier triple index is shown in Fig. 2.

The establishment of the two-tier triple index for the other three orders (P-SO, P-OS, O-PS) is similar to that for S-PO order. In this example, 15 triples are inserted into the triple index. For the traditional B+ tree-based triple index, 15 keys will be inserted into the B+ tree. However, in our implementation, only 3 AP index insertion operations are needed, which decreases the maintenance burden of the B+ tree. Furthermore, *BT index* can be written directly which is also efficient. Thus, compared with the traditional triple index structure, our two-tier triple index gains better performance.

Now, we have described the basic structure of the two-tier triple index. Next, we will look into the triple index deeply.

For a specific *FA pair p*, the *binary tuples* associated with it will be stored in several *blocks*. The *blocks* can be either continuous or incontinuous. When initial bulk load, the system will allocate continuous *blocks* for the *binary tuples* associated with *p* to ensure that the *binary tuples* associated with the same *FA pair* being stored in continuous disk space. After deletion operations are performed, there will exist several unused *blocks* in the middle of *BT index*. At this time, the space allocation algorithm has to give considerations to both *data locality* and *external fragmentation* while inserting new *binary tuples*. It should make the choice between allocating continuous blocks (which benefits *data locality* but causes *external fragmentation*) and allocating several incontinuous blocks (which breaks the *data locality* but does not cause *external fragmentation*). When the rate of fragmentation in *BT index* exceeds a threshold, a reorganization operation is required to reorganize the *BT index* to locate *blocks* associated with the same *FA pair* in a continuous disk space.

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*. In general, the number of different subjects and objects in an RDF dataset is large and the number of triples with the same subject or object is usually small, therefore, if *block* size is big, too much *internal fragmentation* will be made which causes the waste of disk space. 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. In our system implementation, the *memory mapped file* technique is used, which accesses disk with operating system page as the basic unit.

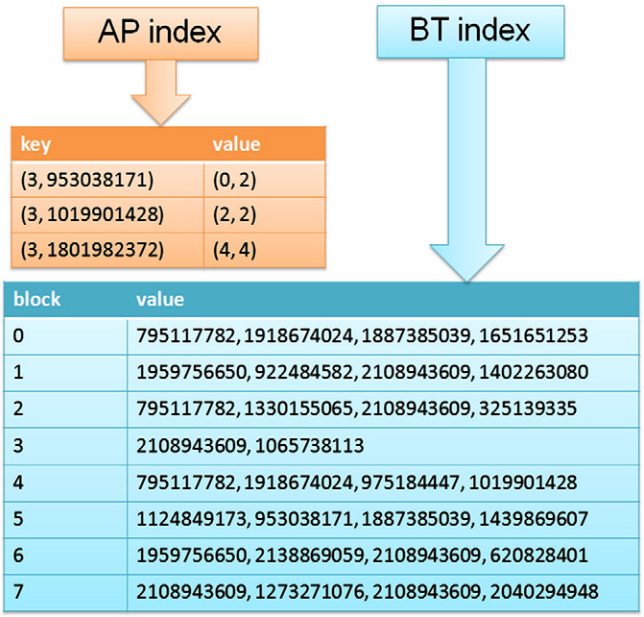

Fig. 2. Structure of AWETO triple index.

Although *blocks* associated with a specific *FA pair* may be incontinuous in disk, for each *FA pair*, the *blocks* associated with it can be seen as *logically continuous*, that is, for a given *FA pair p*, although the *blocks* associated with *p* may be incontinuous in disk, when we put the *blocks* together, they represent all the associated *binary tuples* for *p*.

Because the size of block is small, 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 (see Section 5). *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 5) and U-SIP pruning[4] [12]. For compression and decompression, we adopt *segment* as the basic unit because the compression rate will drop if too small compression unit such as *block* is used. For U-SIP pruning, if *block* is used as basic unit, significant CPU overhead will be caused due to the check for pruning is too often. Thus, introducing the concept of *segment* benefits both the data compression and U-SIP pruning.

Compression is needed for both the *AP index* and *BT index*. There is a tradeoff between the space savings and CPU consumption for decompression of compressed items. Too aggressive compression will degrade the query performance since decompression will dominate most of the time during reading and decompressing [11,23]. For *AP index*, we adopt the famous *variable-byte coding* [24] for the value field of the data items in *AP index*. For *BT index*, instead of storing the original *binary tuples*, for each *segment*, we compute the difference values between adjacent *binary tuples* and store the difference values. We use 32-bit integer for all predicates and 64-bit integer for subjects and objects and treat a 64-bit integer as two 32-bit integers in our current implementation for further use. For a (*predicate, object*) or (*predicate, subject*) *binary tuple*, there are three 32-bit integers for each *binary tuple*. We can adopt the compression scheme of [11] which is also a three-32-bit-integer compression scheme. For a (*subject, object*) or (*object, subject*) *binary tuple*, we adopt our own differential compression scheme. Also note that, we store the length of the compressed data at the beginning of the *segment* such that the decompressor can know how many bytes to be decompressed.

Our compression scheme is similar with [11]. In a (*subject*, *object*) or (*object*, *subject*) *binary tuple*, four 32-bit integers represent a *binary tuple*. Thus, we slightly modify the compression scheme of [11] to support four-32-bit-integer compression. We also add header bytes before each *binary tuple* and write the non-zero tails of the integers in the tuple. Our *binary tuple* consists of four integers; one header byte is not sufficient to represent different compression conditions (totally 340 different conditions). And we think two header bytes for each *binary tuple* is a waste of space. Alternatively, we add a *binary tuple header* for each *binary tuple* and group four *binary tuples* into a group to generate a *group header*. The *group header* and the four *binary tuple headers* guide the decompressor on how to decompress the four *binary tuples*.

According to our test, for AP index, 20.7% of disk space can be reduced when adopting the above compression scheme for both of the datasets YAGO2 and LUBM, which means 20.7% of I/O operation can be reduced in average while performing querying and incremental update in YAGO2 and LUBM . For BT index, 42.8% and 51.3% of disk space can be reduced for the datasets YAGO2 and LUBM respectively. Thus, by introducing compression into AWETO, lots of I/O time can be saved.

4.2. Incremental update of triple index

In each of the ID-based RDF triples to be inserted or deleted, we add a flag f. f equaling to '*i*'/'*d*' indicates the triple should be inserted into/deleted from the triple index. We sort the new triples into four different orders, i.e., SPO, PSO, POS and OPS to get the *FA pairs* and the 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 the triple index. Then we perform a merge operation to merge bt_{old} and bt_{new}. While performing the merge operation, all the triples with $f =$ '*i*' will be inserted into bt_{old}, and all the triples with $f =$ '*d*' will be removed from bt_{old}. 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 the *FA pair* in *AP index*.

Now we analyze the cost of the incremental update operation. Suppose the numbers of triples in bt_{old} and bt_{new} are m and n, respectively. Then, the cost for each sort operation is $O(nlog(n))$. We have four sort operations, thus the total cost for the sort operations is $O(4nlog(n))$. The cost for the merge operation is $O(m + n)$. Therefore, the overall cost of the incremental update operation is $O(4nlog(n) + m + n)$.

Note that, in the scenario of new triples arriving frequently and in small chunks, we will temporarily locate the new triples in memory. When the number of triples in memory meets a certain threshold, the triples will be inserted into/deleted from the triple index in a batch manner.

Now, we use an example to demonstrate the incremental insertion of our two-tier triple index. Recall the example in Section 4.1 and we also take SPO order for example. Assume the following two triples need to be inserted:

(1019901428, 854315478, 315487547)
(1019901428, 2108943609, 457865481).

Firstly, we look up the *AP index* using the key (3, 1019901428). Then we get its value (2, 2), which means the *binary tuples* associated with the subject 1019901428 are stored in *blocks* 2 and 3. Then, we get the *binary tuples* from *blocks* 2 and 3, namely, 795117782, 1330155065, 2108943609, 325139335, 2108943609, 1065738113, and merge the *binary tuples* with the new ones,

[4] U-SIP is a light-weight run-time method that enables index scan operator to skip reading some useless part of the index during query execution, which improves the query execution efficiency.

namely 854315478, 315487547, 2108943609, 457865481. After the merging, we get 795117782, 1330155065, 854315478, 315487547, 2108943609, 325139335, 2108943609, 457865481, 2108943609, 1065738113. Two *blocks* are not enough to store these tuples, so we allocate the third *block*, *block* 8, then these tuples are written to *blocks* 2, 3 and 8. Fig. 3 shows the structure of the triple index after the insertion.

5. 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*. As our index contains four different orders and index scan for the four orders is similar, we take the S-PO order as an example to illustrate the operator. There are four kinds of 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 the value field of the AP index, the bindings for $?p$ and $?o$ can be naturally retrieved.

For $(s, p, ?o)$, we also look up *AP index* using *s*. Compared with $(s, ?p, ?o)$, the difference is, among all the bindings of $(s, ?p, ?o)$, we must get the bindings that satisfy $?p = p$. Here, we adopt an *atom filter* to accomplish this task. Before a new *segment* is read from disk, the *atom filter* will judge if *p* belongs to the range of the *segment*. (The range information is located in the *AP index* and has already been read to memory together with the (*position*, *length*) pairs when looking up s in *AP index*.) If *p* does not belong to the *segment*, we skip reading the *segment* and continue to the next *segment*. In this way, useless *segments* will not be read thus eliminating redundant I/O operations. We should note that, after we read the first and last *segment*, the *binary tuples* in these two segments may not contain only the binding *p* for $?p$, therefore, the *binary tuples* within these two *segments* still need to be filtered internally the *segment*.

Here we demonstrate an example for the *atom filter*. Suppose the following *binary tuples* are associated with subject 10 with the order S-PO and one segment contains four tuples: {(1, 15), (1, 16), (1, 17), (1, 18)}, {(1, 19), (2, 20), (2, 21), (2, 22)}, {(2, 23), (2, 24), (2, 25), (2, 26)}, {(2, 27), (2, 28), (2, 29), (3, 9)}. Now we want to query the SAP $(10, 2, ?o)$. Firstly, we look up the AP index with key 0x03000000000000000A and find only the second, third and fourth *segments* can potentially contain *binary tuples* with $?p = 2$ by checking the range information. Thus all other *segments* can be skipped to reduce I/O operations (In this example, the first *segment* is skipped). The second and fourth *segments* still need to be filtered because they may contain *binary tuples* not satisfying $?p = 2$, such as tuples (1, 19) and (3, 9) in this example. The *segments* in the middle (the *third segment* in this example) do not need to be filtered because we know all the *binary tuples* must satisfy $?p = 2$.

Similarly to $(s, p, ?o)$, for (s, p, o), the atom filter filters not only the $?p$ value but also the $?o$ value.

At last, for the special case $(?s, ?p, ?o)$, i.e., get all the triples from the RDF storage system, we get all the bindings of $?s$ by performing a full scan on the keys that represent S-PO order in the *AP index*. Because we use B+ tree to maintain the *AP index*, the keys that represent S-PO order are sorted, thus the retrieved bindings of $?s$ are naturally sorted, which enables merge join on $?s$. Then for each binding of $?s$, the operation is the same as $(s, ?p, ?o)$.

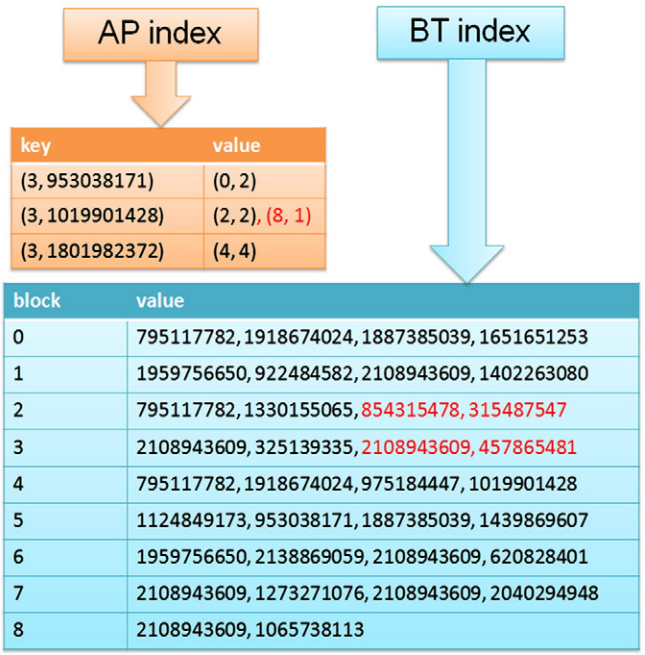

Fig. 3. Structure of AWETO triple index after insertion.

The U-SIP technique introduced in [12] is also adopted in our query execution engine. For merge join, our execution engine gets the *next* value of a *segment* by getting the maximum value of all current variable bindings that exist in the same *equivalent class*. If the *next* value is larger than the maximum value of the *segment*, which indicates the *segment* can be safely pruned, the *index scan operator* will skip the *segment* and go ahead to the next one. For hash join, we build Bloom filter for each *equivalent class* containing a hash join, then detect skips according to the Bloom filter. By U-SIP technique, some useless *segments* for joins can be safely pruned thus improving the performance by decreasing disk I/O time, decompression time and join time.

In order to clearly describe how the *atom filter* and *equivalent class* work when our *index scan operator* reads a segment, we give a formal description in Algorithm 3. There are two input arguments in the algorithm: the *atomFilter* object and the *equivalentClass* object. *atomFilter* object deals with the SAPs $(a_1, a_2, ?v_1)$ and (a_1, a_2, a_3). There are two methods in *atomFilter* object. One is *prune*, which returns if the current *segment* can be safely pruned. It is done by checking the range information of the current *segment*. The other is *needInternalFiltering* which returns if we need to internally filter the current *segment* while reading the *segment*. It is done by checking if the minimum value of the data in the *segment* equaling to the maximum value in the *segment*. The *equivalentClass* object is responsible for getting the *next* value described above. For SAPs $(a_1, ?v_1, ?v_2)$ and $(a_1, a_2, ?v_1)$, *equivalentClass* contains the *equivalent class* information on $?v_1$ if $?v_1$ is contained by an *equivalent class*. Otherwise, *equivalentClass* is set to *null*.

In line 1 to line 19, we try to skip useless *segments* which is done by *atom filtering* and *equivalent class*. In line 2, we get the information of the next *segment* to be read (*segmentInformation*). *segmentInformation* contains the positions, lengths and range information of the *segment*. If there is no longer the next *segment*, we have finished reading all the *segments* (lines 3 and 4). Otherwise, we first check if *atom filtering* is needed (line 5). If it is, the *prune* method is invoked to judge if the current *segment* can be filtered (line 6). If it can, we continue to the next *segment* (lines 7 and 8). Then we check if *equivalent class* can be used (line 9). If it can, there are two conditions. One condition is for the SAP $(a_1, ?v_1, ?v_2)$. In this condition, *atomFilter* is *null* (line 10). We get the *next* value by the minimum value of $?v_1$ in the *segment* (line 12). If the *next* value is larger than the maximum value of $?v_1$ in the *segment*, we can conclude that the current *segment* is useless for join thus we continue to the next *segment* (lines 13 and 14). The other condition is for the SAP $(a_1, a_2, ?v_1)$. In this condition, we need to check if the current *segment* needs to be filtered internally (line 15). If it does not, we know that all the *binary tuples* in the *segment* match the SAP $(a_1, a_2, ?v_1)$. In this way, *equivalent class* can be used similarly with the previous condition (lines 17–19). We should note that, if the current *segment* needs to be filtered internally, which indicates not all the triples in the *segment* match the SAP $(a_1, a_2, ?v_1)$, we cannot know the minimum and maximum value of $?v_1$ because we only know the range information of the *segment*. In this way, *equivalent class* cannot be used for the *segment*. After the filtering and pruning of useless *segment*, in lines 20–23, we read the *binary tuples* in the *segment*.

Algorithm 3: Algorithm when the *index scan operator* reads a *segment*

```
   Input: The atomFilter object
          The equivalentClass object.
 1 while true do
 2     segmentInformation ← getNextSegmentInformation();
 3     if segmentInformation = null then
 4         return false;
 5     if atomFilter ≠ null then
 6         prune = atomFilter.prune(segmentInformation);
 7         if prune = true then
 8             continue;
 9     if equivalentClass ≠ null then
10         if atomFilter = null then
11             /*Here minValue1 and maxValue1 represent the minimum
                value and maximum value of the first values in the binary
                tuples in the segment.*/
12             next =
                equivalentClass.next(segmentInformation.minValue1);
13             if next > segmentInformation.maxValue1 then
14                 continue;
15         else if not(atomFilter.needInternalFiltering()) then
16             /*Here minValue2 and maxValue2 represent the minimum
                value and maximum value of the second values in the binary
                tuples in the segment.*/
17             next =
                equivalentClass.next(segmentInformation.minValue2);
18             if next > segmentInformation.maxValue2 then
19                 continue;
20 if atomFilter = null or not(atomFilter.needInternalFiltering()) then
21     read the data in the segment;
22 else
23     read the data in the segment and do the internal filtering;
24 return true;
```

Table 1
Runtime and knowledge base size for initial bulk load.

	YAGO2		LUBM	
	Time	Size	Time	Size
AWETO	20 min and 24 s	2.9 GB	42 min and 31 s	4.1 GB
RDF-3X	23 min and 25 s	2.3 GB	74 min and 9 s	4.6 GB
MonetDB	21 min and 48 s	1.4 GB	64 min and 12 s	2.6 GB
PostgreSQL	44 min and 15 s	6.4 GB	107 min and 25 s	11.9 GB
OWLIM-SE	57 min and 2 s	3.3 GB	530 min and 23 s	4.9 GB

The B+ tree look up operation in *AP index* is an efficient operation. The *atom filter* is also highly efficient because it filters in the *segment* level and skips many useless *segments*. Thus, for all the four patterns, our *index scan operator* can achieve high efficiency.

6. Experimental evaluation

6.1. General setup

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 with eight 1.66 GHz CPU cores and 20 GB memory. In the server, we ran a 64-bit Linux with the kernel version of 2.6.18. 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 [11–14] is the primary competitor of our system. We used RDF-3X 0.3.6 to run the tests. Although the authors of gStore reported the performance of gStore is better than RDF-3X in [10], we cannot get the code or executable file of gStore. We contacted the authors of gStore, but did not get any response. Furthermore, all the systems in our evaluation are disk-based. However, gStore relies on both the tree index which is located in memory and the adjacency list in disk. The tree index will occupy large amount of memory when the dataset is large. Thus, gStore is not our competitor. The second baseline system is column-store-based vertical partitioning approach introduced in [25,9], which has gained the best performance among all other approaches based on database. Differently from [25,9], we used MonetDB [26] instead of C-Store [27] as the underlying column store. Because C-Store was no longer maintained and the query execution engine was not fully implemented, the authors of C-Store suggested using MonetDB instead. The third baseline is the PostgreSQL [28] database system acting as *triple store*. For the string-ID mapping, we built indices on both IDs and strings, which made it support both ID-to-string query and string-to-ID query. And for the triple table, the indices were built with the orders SPO, PSO and POS which belong to the Sesame-style storage system [29]. Other systems like Jena2 [30,31], YARS2 [32,33] and Sesame [29,34] were evaluated by [11] and suffered scalability issues with large datasets. Thus we have omitted these systems for comparison. In addition, although BitMat [35] is a recent work, it optimizes for queries with low selectivity and sacrifices much on the performance of highly-selective queries, which is different from the purpose of our work. Thus, we exclude BitMat for comparison. Furthermore, we also compared our system with a commercial RDF storage system, OWLIM-SE [36].

6.2. Query evaluation

For query evaluation, we performed both the cold-cache test and warm-cache test. In cold-cache test, all the file system caches were dropped by the **/proc/sys/vm/drop_caches** kernel interface before the start of each run. All the queries were ran ten times

Table 2
Query run-times (in seconds) for YAGO2 dataset.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geom. mean
Cold caches								
AWETO	0.709	0.568	0.402	0.624	0.217	0.416	0.426	0.453
RDF-3X	0.464	0.370	0.316	0.396	0.171	0.341	0.322	0.327
MonetDB	18.963	9.567	9.325	15.711	16.486	12.115	10.986	12.865
PostgreSQL	33.368	0.660	3.062	15.854	4.650	2.660	9.167	5.324
OWLIM-SE	2.052	0.353	0.448	4.768	3.896	0.901	0.236	1.036
Warm caches								
AWETO	0.118	0.104	0.043	0.132	0.023	0.028	0.061	0.060
RDF-3X	0.141	0.131	0.048	0.149	0.029	0.062	0.053	0.074
MonetDB	5.048	0.877	1.338	1.389	2.266	2.251	1.552	1.816
PostgreSQL	1.266	0.052	0.051	0.659	0.281	0.136	0.056	0.174
OWLIM-SE	0.425	0.023	0.035	2.780	1.938	0.065	0.008	0.138

Table 3
Query run-times (in seconds) for LUBM dataset.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Geom. mean
Cold caches								
AWETO	0.163	0.171	0.367	0.330	0.331	2.227	0.296	0.357
RDF-3X	0.107	0.101	0.229	0.171	0.281	1.221	0.144	0.215
MonetDB	14.920	15.040	16.532	17.936	21.717	23.019	15.299	17.531
PostgreSQL	0.335	0.431	1.197	11.117	1.747	109.434	3.307	2.758
OWLIM-SE	0.065	0.065	0.101	556.708	0.111	0.837	0.119	0.428
Warm caches								
AWETO	0.005	0.006	0.022	0.120	0.033	1.108	0.042	0.038
RDF-3X	0.002	0.003	0.008	0.080	0.103	0.665	0.025	0.025
MonetDB	1.377	1.364	2.257	4.596	3.514	3.440	1.964	2.403
PostgreSQL	0.021	0.048	0.078	9.404	0.077	0.678	0.049	0.152
OWLIM-SE	0.006	0.006	0.018	553.988	0.010	0.237	0.004	0.062

to avoid the influence of other OS activities and we reported the best result. Before the query test, we load all the triples in YAGO2 and LUBM (Notation 3 (N3) [37] format) into the five systems. The load time and database size are illustrated in Table 1.

For the first experiment we used the YAGO2 [7] dataset (the core version, N3 format), which is a huge semantic knowledge base, derived from Wikipedia, WordNet and GeoNames. It contains 15,820,985 different strings and 32,393,226 different triples. The previous work RDF-3X [11] uses the YAGO [6] dataset. However, the latest version of RDF-3X system cannot work well with YAGO dataset. In addition, YAGO2 dataset is different from the original YAGO, some of the queries for YAGO will generate empty result in YAGO2. For the above reasons, we modified and rewrote seven queries to test the query performance of all systems with YAGO2 dataset. All the queries are shown in Appendix A.1.

The experimental results for YAGO2 are shown in Table 2. The same as previous work, we also use the geometric mean of query time to measure the performance of each system. Geometric mean is often used as a workload-average measure in benchmarks [11]. Firstly, we compare our system with RDF-3X. The 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 and hash-based string-ID mapping strategy is adopted. The mapped IDs uniformly distribute in the range from 1 to 2^{31}–1. However, in the sequential strategy, the IDs will be in the range from 1 to *number of different strings*. The big range of hash-based string-ID mapping influences the compression efficiency of *binary tuples* because the compression scheme of *binary tuples* is based on difference values between adjacent *binary tuples*. 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 [25], 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. In our experiment, MonetDB consumes much of the time for this conversion so that the conversion has dominated the query time. For this dataset, MonetDB takes about 8 s to convert a string into its ID representation in cold caches, and about 0.22 s in warm caches. While for other three systems, they only takes a little time for the conversion compared with the execution time of the operator tree. Thus executing the operator tree dominates the query time in the three other systems. The reason for slow conversion from string to ID is caused by the index strategy of MonetDB. MonetDB creates and maintains indices relying on its own decision [38]. If we eliminate the time for conversion, AWETO, RDF-3X, MonetDB and PostgreSQL achieves geometric means of 0.404, 0.262, 2.158 and 5.106 respectively for cold caches and 0.060, 0.074, 0.099 and 0.150 for warm caches. (Our system and

Table 4
Incremental insertion time (in seconds) for string-ID mapping (YAGO2).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
11 million	40.481	141.969	3.507
12 million	48.048	151.219	3.147
13 million	39.012	170.536	4.371
14 million	53.358	183.288	3.435
15 million	50.271	188.293	3.746
16 million	53.110	203.237	3.827
17 million	52.023	189.881	3.650
18 million	53.513	200.136	3.740
19 million	55.553	194.822	3.507
20 million	55.215	197.927	3.585
Average	50.058	182.131	3.638

Table 5
Incremental insertion time (in seconds) for string-ID mapping (LUBM).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
32 million	116.963	509.112	4.353
34 million	116.852	527.611	4.515
36 million	104.073	526.203	5.056
38 million	104.944	536.388	5.111
40 million	106.161	535.824	5.047
42 million	95.473	531.897	5.571
44 million	103.760	527.151	5.080
46 million	101.237	532.368	5.259
48 million	103.767	528.203	5.090
50 million	108.775	514.674	4.732
Average	106.201	526.943	4.962

RDF-3X consumes less than 1 millisecond to do the string-to-ID conversion in warm caches.) In our experiments, for the above four systems, we do not consider the time for converting the ID representation of atoms in the query results into their string representation because these systems all convert the query results by B + trees and it is not related to the query execution engine. For the commercial system OWLIM-SE, because it is not open source and we cannot get the implementation details of OWLIM-SE, we only show the experimental results and the time for query results conversion is not excluded.

For the second experiment, we used the LUBM [39] dataset. The Lehigh University Benchmark (LUBM) consists of a university domain ontology with synthetic data. 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. Then, we use the Raptor RDF parser utility [40] to convert the triples to N3 format. Because LUBM is a synthetic dataset, the triple order in it has fixed and regular pattern, which is not like real datasets. When generating the N3 file for the dataset, we disorganized the triple order in the dataset to make it more natural. For test queries, we selected seven representative queries from the 14 test queries provided by LUBM dataset. Because the tested RDF storage systems do not support reasoning, we slightly modified the queries. All the queries are shown in Appendix A.2.

The experimental results for LUBM are shown in Table 3. RDF-3X storage system achieves the best performance and our system is the runner-up. Compared with RDF-3X, RDF-3X outperforms our system by an average factor of about 1.7 in cold caches and about 1.5 in warm caches. For PostgreSQL, 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, the same problem appears. MonetDB takes about 12 s to convert a string into its ID representation in cold caches, and about 0.3 s in warm caches in LUBM dataset. If we eliminate the time for the string-to-ID conversion, AWETO, RDF-3X, MonetDB and PostgreSQL achieves geometric means of 0.265, 0.145, 3.404 and 2.400 respectively for cold caches and 0.038, 0.025, 0.361, 0.096 for warm caches.

6.3. Incremental update evaluation

The performance of incremental update is also an important factor for RDF storage systems. In this section, we present the performance of incremental update. To test the incremental update performance of AWETO systematically, we test both the incremental insertion and deletion efficiency. In each evaluation, we divide it into two parts, which evaluates the performance of string-ID mapping and the triple index, respectively.

Table 6
Incremental insertion time (in seconds) for triple index (YAGO2).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
11 million	47.441	112.486	2.371
12 million	45.033	114.144	2.535
13 million	49.190	117.999	2.399
14 million	51.119	139.517	2.729
15 million	50.159	143.653	2.864
16 million	51.053	150.152	2.941
17 million	53.217	162.221	3.048
18 million	52.731	170.572	3.235
19 million	54.365	195.359	3.593
20 million	57.235	209.920	3.668
Average	51.154	151.602	2.964

Table 7
Incremental insertion time (in seconds) for triple index (LUBM).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
32 million	130.797	378.559	2.894
34 million	171.933	476.523	2.772
36 million	273.941	693.151	2.530
38 million	328.183	912.901	2.782
40 million	391.176	969.036	2.477
42 million	435.164	987.772	2.270
44 million	462.275	1099.538	2.379
46 million	488.471	1165.601	2.386
48 million	510.270	1169.956	2.293
50 million	530.156	1278.030	2.411
Average	372.237	913.107	2.453

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. The knowledge base after performing incremental update cannot be used for both querying and incremental update again. Furthermore, our system is implemented in Java and takes Tokyo Cabinet [21] as the B+ tree implementation. RDF-3X is written in C++ and uses their own 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 as well and tried our best to make the code the most efficient. In our RDF-3X implementation, we reduced the number of triple indices from fifteen in the original RDF-3X implementation, to four in our implementation. This 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. For MonetDB, due to its slow querying time on string-to-ID mapping reported in Section 6.2, incremental update for MonetDB is a disaster. We tested the performance of incremental insertion using YAGO2 dataset with initial size 5 million triples and incremental size only 1000 triples. It takes 121.814 s to incrementally insert the strings in the triples which concludes the rate of incremental insertion of String-ID mapping for MonetDB is 8.21 triples/s. For PostgreSQL, only two small incremental sizes were tested because PostgreSQL can only achieve an incremental insertion efficiency of several hundreds of triples per second. We tested PostgreSQL on YAGO2 and LUBM respectively with 5 million triples for both initial and incremental sizes, and we got an incremental insertion rate of 304 triples/s for YAGO2 and 695 triples/s for LUBM, which is also much slower than our system and RDF-3X. For the above reason, we only report the incremental update performance of our system and RDF-3X for incremental update with large amount of triples.

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 execute 10 batches of incremental insertion, 1 million triples in each batch, thus after the incremental insertion, the knowledge base will contain 20 million triples. The number of triples in LUBM is about two times than that in YAGO2. Thus, for LUBM we set the initial knowledge base size to 30 million triples and repeat 10 batches of incremental insertion, 2 million triples per batch. After the incremental insertion, the knowledge base will contain 50 million triples in total. For incremental deletion, the initial knowledge base size is set to 30 million triples for YAGO2 and 50 million triples for LUBM and we delete 1 million triples per batch for YAGO2, and 2 million triples for LUBM.

The experimental results of incremental insertion of our system and RDF-3X are shown in Tables 4, 5, 6, and 7. The column of "knowledge base size" indicates the number of triples in the knowledge base after the incremental update. The incremental update time is measured in seconds. The last columns in the tables show the runtime ratio which is defined as the quotient of the RDF-3X's runtime and AWETO's runtime. For both the string-ID mapping and triple index, our system achieves the best incremental

Table 8
Incremental deletion time (in seconds) for string-ID mapping (YAGO2).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
19 million	27.245	126.003	4.625
18 million	25.599	125.543	4.904
17 million	25.576	124.011	4.849
16 million	25.615	122.932	4.799
15 million	26.326	120.364	4.572
14 million	26.023	124.551	4.786
13 million	25.771	123.703	4.800
12 million	26.085	122.775	4.707
11 million	25.946	119.397	4.602
10 million	25.752	113.614	4.412
Average	25.994	122.289	4.705

Table 9
Incremental deletion time (in seconds) for string-ID mapping (LUBM).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
48 million	59.088	473.220	8.009
46 million	56.816	480.841	8.463
44 million	57.304	481.777	8.407
42 million	56.687	475.629	8.390
40 million	56.548	479.174	8.474
38 million	57.265	480.562	8.392
36 million	55.949	478.778	8.557
34 million	56.426	483.762	8.573
32 million	55.227	478.327	8.661
30 million	56.080	481.679	8.589
Average	56.739	479.375	8.449

insertion 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. The experimental results of incremental deletion are shown in Tables 8, 9, 10, and 11, which also conclude both of our hash-based String-ID mapping approach and the triple index are optimized for incremental update and more time-efficient than RDF-3X.

The experimental results of incremental insertion of our system and OWLIM-SE are shown in Tables 12 and 13. Our system outperforms OWLIM-SE by an average factor of 1.099 using YAGO2 dataset, however, OWLIM-SE gains better performance than our system using LUBM dataset.

To conclude the above experiments, by adopting both the hash-based string-ID mapping approach and the two-tier triple index, the incremental update efficiency can be improved substantially and our system achieves the highest incremental update efficiency compared with the other three state-of-the-art open source RDF storage systems.

7. Related work

The storage and querying of RDF data have been studied for over ten years. Existing RDF storage systems can be divided into two categories. The first category relies on an underlying DBMS system [30,41,31,42,43,29,34,25,44,9,45,17,46,18,47]. In these approaches, triples are stored into a relational database. Since the lengths of atom strings are usually long, many systems convert the triples with their string representation into their ID representation. Then the string-ID mapping table and ID-based triples are stored into the database. Many of the systems use a sequential-based approach to map the strings, Oracle [17], 3store [18] and our system adopt a hash-based ID mapping strategy. ID-based triples can be stored in the database in different ways. All the triples can be stored in a "giant triple table" with subject, predicate, object as three columns of the table [30,31,29,34,45,46,18]. Alternatively, *property table* [30,41,31] can be used which put some entities with their properties into property tables and put the left-over triples in a triple table. What is more, along with the design of column-based store, *vertical partitioning* approach [25,44,9] has been adopted in column-based stores. In this approach, triples with the same predicate are stored in the same table. This approach gains good query performance when the number of total predicates is not too large. Bornea et al. [47] introduces an *entity-oriented* approach which stores relationships associated with the same entity into one or more rows in the relational database.

The second category of RDF storage systems relies on their own index architectures [48,49,35,32,33,50–52,11,13,12,14,10,53]. Most of the systems utilize B+ trees as their index structure and store the ID-based triples in different orders [32,33,50,51,11,13,12,14]. gStore [10] utilizes an in-memory vertex signature tree and an in-disk adjacency list to store and index all the triples. Hexastore [48] proposes a sextuple index architecture, however the experiments were only done based on an in-memory prototype of Hexastore.

Table 10
Incremental deletion time (in seconds) for triple index (YAGO2).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
19 million	41.354	62.016	1.500
18 million	41.300	60.359	1.461
17 million	39.767	57.199	1.438
16 million	39.712	55.753	1.404
15 million	39.384	54.603	1.386
14 million	38.548	52.616	1.365
13 million	37.488	50.038	1.335
12 million	37.006	49.359	1.334
11 million	36.951	46.079	1.247
10 million	35.459	45.104	1.272
Average	38.697	53.313	1.378

Table 11
Incremental deletion time (in seconds) for triple index (LUBM).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	RDF-3X	
48 million	135.950	158.022	1.162
46 million	119.920	157.924	1.317
44 million	116.775	153.728	1.316
42 million	118.790	159.973	1.347
40 million	114.620	144.872	1.264
38 million	111.399	141.615	1.271
36 million	128.383	157.715	1.228
34 million	121.460	149.528	1.231
32 million	123.666	142.366	1.151
30 million	119.672	150.155	1.255
Average	121.064	151.590	1.252

GRIN [52] introduces an index format based on binary tree and hash tables by grouping information around selected "center" nodes, however GRIN can only work well with small or medium datasets. BitMat [49,35] builds a compressed bit-matrix structure and applies bit operations to execute queries which is optimized for low-selectivity queries. TripleBit [53] introduces a compact RDF storage system, which uses a compressed triple matrix storage structure and two auxiliary indexing structures, ID-Chunk and ID-Predicate bit matrix to support efficient querying of RDF data.

For incremental update, to the best of our knowledge, the SW-Store [9] establishes an *overflow table* and an RDF *batch writer* is implemented to convert RDF triples in the *overflow table* into its main index (the vertical-partitioned tables). However, the update efficiency was not reported. The RDF-3X engine [11,13,12,14] establishes *differential index* in memory and merges the differential index with its main index (clustered B+ tree index) when the number of triples existing in memory exceed a threshold. The performance of incremental update is reported in [13]. For gStore [10], because the vertex signature tree is in memory, its maintenance overhead is low. The strategy to maintain the in-disk adjacency list is not reported in their paper.

8. 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 on both reading data from triple index and performing join operations. Experimental results show that our system is competitive in querying and outperforms the other three state-of-the-art open source RDF storage systems when performing incremental update. With the fast growth of RDF data, storing RDF data in a single node cannot satisfy the demand of high performance. For future work, we will investigate the distributed RDF storage system based on AWETO, which partitions the RDF data into multiple nodes and performs querying and incremental update in a parallel way.

Acknowledgment

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61272088, the National Key Basic Research Program of China (973 Program) under Grant No. 2014CB340505, and an HP Labs Innovation Research Program award.

Table 12
Incremental insertion time (in seconds) comparison with OWLIM-SE (YAGO).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	OWLIM-SE	
11 million	87.922	56.850	0.647
12 million	93.081	85.885	0.923
13 million	88.202	74.555	0.845
14 million	104.477	88.573	0.848
15 million	100.430	155.761	1.551
16 million	104.163	125.988	1.210
17 million	105.240	124.999	1.188
18 million	106.244	140.878	1.326
19 million	109.918	132.204	1.203
20 million	112.450	126.313	1.123
Average	101.213	111.201	1.099

Table 13
Incremental insertion time (in seconds) comparison with OWLIM-SE (LUBM).

Knowledge base size	Incremental update time		Runtime ratio
	AWETO	OWLIM-Lite	
32 million	247.760	292.449	1.180
34 million	288.785	304.703	1.055
36 million	378.014	348.900	0.923
38 million	433.127	415.202	0.959
40 million	497.337	416.696	0.838
42 million	530.637	470.561	0.887
44 million	566.035	487.803	0.862
46 million	589.708	499.811	0.848
48 million	614.037	565.809	0.921
50 million	638.931	475.256	0.744
Average	478.437	427.719	0.894

Appendix A. Queries for evaluation

A.1. YAGO2

Q1: select ?name ?state ?m1 ?m2 where {
?a <isCalled> ?name.
?a <type> <wordnet_actor_109765278>.
?a <livesIn> ?city.
?city <isLocatedIn> ?state.
?a <actedIn> ?m1.
?m1 <type> <wordnet_movie_106613686>.
?a <directed> ?m2.
?m2 <type> <wordnet_movie_106613686>.
filter (?m1 !=?m2)}

Q2: select ?name ?b ?c where {
?a <isCalled> ?name.
?a <livesIn> ?b.
?b <isLocatedIn> ?c.
?c <isLocatedIn> <United_States> }

Q3: select ?name ?m1 ?i1 ?m2 ?i2 where {
?a <isCalled> ?name.
?a <actedIn> ?m1.
?m1 <hasImdb> ?i1.
?a <directed> ?m2.
?m2 <hasImdb> ?i2.
?a <isLocatedIn> ?b.
?b <isLocatedIn> <New_Jersey>}

Q4: select ?name ?city ?l where {
?p <isCalled> ?name.
?p <type> <wordnet_actor_109765278>.
?p <wasBornIn> ?city.
?city <isLocatedIn> ?l }

Q5: select distinct ?name1 ?name2 where {
?p1 <hasFamilyName> ?name1.
?p2 <hasFamilyName> ?name2.
?p1 <type> <wordnet_scientist_110560637>.
?p2 <type> <wordnet_scientist_110560637>.
?p1 <hasWonPrize> ?award.
?p2 <hasWonPrize> ?award.
?p1 <wasBornIn> ?city.
?p2 <wasBornIn> ?city.
filter (?p1 !=?p2) }

Q6: select ?p ?GivenName ?FamilyName ?GivenName2 ?FamilyName2 where {
?p <hasGivenName> ?GivenName.
?p <hasFamilyName> ?FamilyName.
?p <wasBornIn> ?city.
?p <type> <wordnet_scientist_110560637>.
?city <isLocatedIn> <Switzerland>.
?p <hasAcademicAdvisor> ?a.
?a <isCitizenOf> <Germany>.
?a <hasGivenName> ?GivenName2.
?a <hasFamilyName> ?FamilyName2 }
Q7: select ?g1 ?f1 where {
?a <hasGivenName> ?g1.
?a <hasFamilyName> ?f1.
?a <wasBornIn> ?c.
?c <isLocatedIn> <Ohio>.
?a <livesIn> ?b.
?b <isLocatedIn> <Utah>}

A.2. LUBM

Q1: select ?X where {
?X <type> <GraduateStudent>.
?X <takesCourse>
<http://www.Department0.University0.edu/GraduateCourse0>
}
Q2: select ?X where {
?X <type> <Publication>.
?X <publicationAuthor>
<http://www.Department0.University0.edu/AssistantProfessor0>
}
Q3: select ?X ?Y1 ?Y2 ?Y3 where {
?X <type> <Full Professor>.
?X <worksFor> <http://www.Department0.University0.edu>.
?X <name> ?Y1.
?X <emailAddress> ?Y2.
?X <telephone> ?Y3 }
Q4: select ?X where {
?X <type> <UndergraduateStudent> }
Q5: select ?X ?Y where {
?X <type> <UndergraduateStudent>.
?Y <type> <Course>.
?X <takesCourse> ?Y.
<http://www.Department0.University0.edu/AssociateProfessor0> <teacherOf> ?Y }
Q6: select ?X ?Y ?Z where {
?X <type> <UndergraduateStudent>.
?Y <type> <Department>.
?X <memberOf> ?Y.
?Y <subOrganizationOf> <http://www.University0.edu>.
?X <emailAddress> ?Z }
Q7: select ?X ?Y where {
?X <type> <FullProfessor>.
?Y <type> <Department>.
?X <worksFor>?Y.
?Y <subOrganizationOf> <http://www.University0.edu>
}

References

[1] W3C: Resource Description Framework (RDF), http://www.w3.org/RDF 2012.
[2] Wikipedia, http://www.wikipedia.org 2012.
[3] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, Z.G. Ives, DBpedia: A Nucleus for a Web of Open Data, ISWC/ASWC, 2007. 722–735.
[4] WordNet, http://wordnet.princeton.edu 2012.
[5] GeoNames, http://www.geonames.org 2012.
[6] F.M. Suchanek, G. Kasneci, G. Weikum, YAGO: A Core of Semantic Knowledge, 2007. 697–706 www.
[7] YAGO2, http://www.mpi-inf.mpg.de/yago-naga/yago 2012.
[8] Linking open data, http://esw.w3.org/SweoIG/TaskForces/CommunityProjects/LinkingOpenData 2012.
[9] D.J. Abadi, A. Marcus, S. Madden, K. Hollenbach, SW-Store: a vertically partitioned DBMS for semantic web data management, VLDB J. 18 (2009) 385–406.
[10] L. Zou, J. Mo, L. C., M.T. Özsu, D. Zhao, gStore: answering SPARQL queries via subgraph matching, PVLDB 4 (2011) 482–493.
[11] T. Neumann, G. Weikum, RDF-3X: a RISC-style engine for RDF, PVLDB 1 (2008) 647–659.
[12] T. Neumann, G. Weikum, Scalable join processing on very large RDF graphs, SIGMOD Conference, 2009, pp. 627–640.
[13] T. Neumann, G. Weikum, The RDF-3X engine for scalable management of RDF data, VLDB J. 19 (2010) 91–113.
[14] T. Neumann, G. Weikum, x-RDF-3X: fast querying, high update rates, and consistency for RDF databases, PVLDB 3 (2010) 256–263.
[15] X. Pu, J. Wang, P. Luo, M. Wang, AWETO: efficient incremental update and querying in RDF storage system, CIKM, 2011, pp. 2445–2448.
[16] W3C: SPARQL query language for RDF, http://www.w3.org/TR/rdf-sparql-query 2012.
[17] S. Das, E.I. Chong, Z. Wu, M. Annamalai, J. Srinivasan, A scalable scheme for bulk loading large RDF graphs into oracle, ICDE, 2008, pp. 1297–1306.
[18] S. Harris, N. Gibbins, 3store: efficient bulk RDF storage, PSSS, 2003.
[19] A hash function for hash table lookup, http://burtleburtle.net/bob/hash/doobs.html 2012.
[20] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (1970) 422–426.
[21] Tokyo cabinet: a modern implementation of DBM, http://fallabs.com/tokyocabinet 2012.
[22] UniProt RDF, http://dev.isb-sib.ch/projects/uniprot-rdf 2012.
[23] T. Westmann, D. Kossmann, S. Helmer, G. Moerkotte, The implementation and performance of compressed databases, SIGMOD Rec. 29 (2000) 55–67.
[24] H.E. Williams, J. Zobel, Compressing integers for fast file access, Comput. J. 42 (1999) 193–201.
[25] D.J. Abadi, A. Marcus, S. Madden, K.J. Hollenbach, Scalable semantic web data management using vertical partitioning, VLDB, 2007, pp. 411–422.
[26] MonetDB, http://monetdb.cwi.nl 2012.
[27] 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, S.B. Zdonik, C-store: a column-oriented DBMS, VLDB, 2005, pp. 553–564.
[28] PostgreSQL: the world's most advanced open source database, http://www.postgresql.org 2012.
[29] J. Broekstra, A. Kampman, F. van Harmelen, Sesame: an architecture for storing and querying RDF data and schema information, Spinning the Semantic Web, 2003, pp. 197–222.
[30] K. Wilkinson, C. Sayers, H.A. Kuno, D. Reynolds, Efficient RDF storage and retrieval in Jena2, SWDB, 2003, pp. 131–150.
[31] Jena semantic web framework, http://jena.sourceforge.net 2012.
[32] A. Harth, J. Umbrich, A. Hogan, S. Decker, YARS2: A Federated Repository for Querying Graph Structured Data from the Web, ISWC/ASWC, 2007. 211–224.
[33] YARS: Yet Another RDF Store, http://sw.deri.org/2004/06/yars 2012.
[34] OpenRDF, http://www.openrdf.org 2012.
[35] M. Atre, V. Chaoji, M.J. Zaki, J.A. Hendler, Matrix "bit" loaded: a scalable lightweight join query processor for RDF data, Proceedings of the 19th International Conference on World Wide Web, WWW '10, ACM, New York, NY, USA, 2010, pp. 41–50.
[36] OWLIM, http://www.ontotext.com/owlim 2013.
[37] Notation3 (N3): a readable RDF syntax, http://www.w3.org/TeamSubmission/n3 2012.
[38] MonetDB documentation 3.1.8: indexes, http://monetdb.cwi.nl/SQL/Documentation/Indexes.html 2012.
[39] Y. Guo, Z. Pan, J. Heflin, LUBM: a benchmark for OWL knowledge base systems, J. Web Sem. 3 (2005) 158–182.
[40] Raptor RDF parser utility, http://librdf.org/raptor/rapper.html 2012.
[41] K. Wilkinson, K. Wilkinson, Jena Property Table Implementation, SSWS2006.
[42] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, K. Tolle, The ICS-FORTH RDF Suite: Managing Voluminous RDF Description Bases, SemWeb2001.
[43] The ICS-FORTH RDF suite, http://athena.ics.forth.gr:9090/RDF 2012.
[44] L. Sidirourgos, R. Goncalves, M.L. Kersten, N. Nes, S. Manegold, Column-store support for RDF data management: not all swans are white, PVLDB 1 (2008) 1553–1563.
[45] E.I. Chong, S. Das, G. Eadon, J. Srinivasan, An efficient SQL-based RDF querying scheme, VLDB, 2005, pp. 1216–1227.
[46] L. Ma, C. Wang, J. Lu, F. Cao, Y. Pan, Y. Yu, Effective and efficient semantic web data management over DB2, SIGMOD Conference, 2008, pp. 1183–1194.
[47] M.A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, B. Bhattacharjee, Building an efficient RDF store over a relational database, SIGMOD Conference, 2013, pp. 121–132.
[48] C. Weiss, P. Karras, A. Bernstein, Hexastore: sextuple indexing for semantic web data management, PVLDB 1 (2008) 1008–1019.
[49] M. Atre, J. Srinivasan, J.A. Hendler, BitMat: a main-memory bit matrix of RDF triples for conjunctive triple pattern queries, International Semantic Web Conference, 2008, (Posters & Demos).
[50] G.H.L. Fletcher, P.W. Beck, Scalable indexing of RDF graphs for efficient join processing, CIKM, 2009, pp. 1513–1516.
[51] B. Liu, B. Hu, HPRD: a high performance RDF database, NPC, 2007, pp. 364–374.
[52] O. Udrea, A. Pugliese, V.S. Subrahmanian, GRIN: a graph based RDF index, AAAI, 2007, pp. 1465–1470.
[53] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, L. Liu, TripleBit: a fast and compact system for large scale RDF data, PVLDB 6 (2013) 517–528.

Xu Pu received his bachelor's degree at Beijing University of Posts and Telecommunications. Now he is a Master's student at Tsinghua University. His main research topics are RDF data indexing and querying, and data mining.

Jianyong Wang is currently a professor in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He received his PhD degree in Computer Science in 1999 from the Institute of Computing Technology, Chinese Academy of Sciences. He was an assistant professor at Peking University, and visited Simon Fraser University, University of Illinois at Urbana-Champaign, and University of Minnesota at Twin Cities before joining Tsinghua University in December 2004. His research interests mainly include data mining and Web information management. He has co-authored over 60 papers in some leading international conferences and some top international journals. He is serving or ever served as a PC member for some leading international conferences, such as SIGKDD, VLDB, ICDE, WWW, and an associate editor of IEEE TKDE. He is a senior member of the IEEE, a member of the ACM, a recipient of the 2009 and 2010 HP Labs Innovation Research award, the 2009 Okawa Foundation Research Grant (Japan), WWW'08 best posters award, and the Year 2007 Program for New Century Excellent Talents in University, State Education Ministry of China.

Zhenhua Song received his bachelor's degree at Beihang University. Now he is a Master's student at Tsinghua University. His main research topics are RDF data storage and approximate querying, and data mining.

Ping Luo received his PhD degree in Computer Science from the Institute of Computing Technology, Chinese Academy of Sciences. He is currently an associate professor in the Institute of Computing Technology, CAS. He has published several papers in some prestigious refereed journals and conference proceedings, such as the IEEE Transactions on Information Theory, IEEE Transactions on Knowledge and Data Engineering, Journal of Parallel and Distributed Computing, ACM SIGKDD, ACM CIKM, and IJCAI. His research interests include knowledge discovery and machine learning. He is the recipient of the Doctoral Dissertation Award, China Computer Federation, 2009. He is a member of the IEEE Computer Society and the ACM.

Dr. Min Wang is a Distinguished Technologist and the Director of HP Labs China. Prior to joining HP, Dr. Wang led a 10-year career at IBM's Thomas J. Watson Research Center, where she pursued her research interests in database systems and information management.

Dr. Wang has published broadly in the areas of database systems and information management. In 2009, she received the ACM SIGMOD 2009 Test of Time Award for her SIGMOD 1999 paper.

She received her Ph.D. degree in Computer Science from Duke University and the B.S. and M.S. degrees, both in Computer Science, from Tsinghua University, Beijing, China.