Efficient Index Maintenance for Frequently Updated Semantic Data

Yan Liang[1], Haofen Wang[1], Qiaoling Liu[1], Thanh Tran[2], Thomas Penin[1], and Yong Yu[1]

[1] Department of Computer Science & Engineering
Shanghai Jiao Tong University, Shanghai, China
{yliang,whfcarter,lql,tpenin,yyu}@apex.sjtu.edu.cn
[2] Institute AIFB, Universität Karlsruhe, Germany
{dtr}@aifb.uni-karlsruhe.de

Abstract. Nowadays, the demand on querying and searching the Semantic Web is increasing. Some systems have adopted IR (Information Retrieval) approaches to index and search the Semantic Web data due to its capability to handle the web-scale data and efficiency on query answering. Additionally, the huge volumes of data on the Semantic Web are frequently updated. Thus, it further requires effective update mechanisms for these systems to handle the data change. However, the existing update approaches only focus on document. It still remains a big challenge to update IR index specially designed for semantic data in the form of triples, which are finer grained structured objects rather than unstructured documents. In this paper, we present a well-designed update mechanism on the IR index for triples. Our approach provides flexible and effective update mechanism by dividing the index into blocks. It reduces the number of update operations during the insertion of triples. At the same time, it preserves the efficiency on query processing and the capability to handle large scale semantic data. Experimental results show that the index update time is a fraction of that by complete reconstruction w.r.t the portion of the inserted triples. Moreover, the query response time is not notably affected. Thus, it is capable to make newly arrived semantic data immediately searchable for users.

1 Introduction

Nowadays, indexing and retrieving the Semantic Web data is drawing an increasing attention. Some systems such as [1, 2] have adopted IR (Information Retrieval) approaches for indexing these data. In particular, the semantic search engine [1] has indexed over 1.4 million Semantic Web documents and began to provide search services in the Semantic Web community similar to Google. The success is due to the fact that IR is proved to handle web-scale data and be efficient on query answering. Moreover, these systems are benefited from the IR approaches to exploit huge amounts of textual information on the Semantic Web by users keyword queries.

Additionally, the huge volumes of semantic web data are frequently updated. Thus it requires semantic search engines not only to be scalable but also have flexible update mechanisms to make the newly arrived data immediate searchable for users. For example, the search engine in [1] indexes over 1.4 million Semantic Web documents and it has to update its index on a regular basis.

However, to keep the Semantic Web data up-to-date in an IR index is a difficult task. The IR-based approaches index the Semantic Web data by reusing the existing structure of inverted index. Although there are many discussions on the index update for traditional IR search engines [3–7], current update approaches are just suitable for semantic documents. But for an IR index which is designed as a repository of triples, index updating is more difficult. Since during the update it should also keep the original relations between existing individuals. Although some methods (such as [8]) have presented to speed up the index construction, frequently rebuilding the index is still costly.

In this paper, we present an efficient updating mechanism based on [9], which is the state-of-art of the current IR approaches to index and retrieve the large scaled semantic instances (RDF triples). It reuses the IR engine's index structure and functions to provide efficient query processing. Moreover, it supports both the structured queries for semantic web data and the keyword queries for textual information.

Our approach is based on the idea of dividing the index into blocks, which reduce the number of update operations during the insertion of triples. Our index mechanism can also be used for the index update based on an incremental crawler. Experimental results show that the index update time is a fraction of complete reconstruction due to the portion of the inserted triples. Thus it's capable to make the newly arrived semantic data immediately searchable by users. At the same time it preserves both the efficiency on query processing and the scalability to handle the large-scaled semantic data. Moreover,the reuse of IR search engine not only can index the structural Semantic Web data but also the textual information. Thus it supports the hybrid query capability for both structured queries and keyword queries.

The paper is organized as follows. Section 2 introduces the related work. Section 3 describes the basic index structure we are based on. Section 4 discusses the extended block index structure, along with an update mechanism. Moreover, a comprehensive analysis on the performance of our update mechanism is presented in Section 5. Section 6 shows the experimental results and we will give a conclusion in Section 7.

2 Related Work

The update mechanisms for inverted index is a well studied field. Work in [3] proposed a dual inverted list which stores short lists in the memory and long lists on the disk. When the area for the short lists is full, the longest short list will be merged into a long list. Work in [5] presents a hybrid approach in which long posting lists are updated in-place, while short lists are updated using a

merge strategy. [6] improves the in-place update by saving the short posting lists within the vocabulary and over-allocating the long lists. [4] uses overflow 'buckets' to handle the new arriving postings. Work in [7] presents a method to update previously indexed documents whose content have changed. The idea is based on blocking together with the diff algorithm. [10] presents a just-in-time indexing component which invests less in the preprocessing of arriving data, at the expense of a tolerable latency in query response time. Index update can also be achieved by reconstruction. Method in [8] is presented to speed up the index construction.

However, there are few work on the index update for Semantic Web data. [11] enables incremental update of index for XML documents part of which are changed.

Querying and searching semantic web data using IR-based approaches are emerging areas. Work in [1] presents a crawler based indexing and retrieval system for semantic web data. It uses an IR engine to index the crawled semantic web documents by using the n-gram and taking URIrefs as terms. [2] provides an interface for searching ontologies and semantic documents using keywords. However, these works are designed to index semantic documents and they do not index triples. [9] is designed as a repository of RDF triples based on the IR engine's index structure. It supports both structured query and keyword query. [12] uses keyword search results to do spread activation on semantic networks. But it does not support structure queries for semantic web data. Work in [13] discusses the result ranking for combining keyword search and structured queries. [14] borrows XML Fragment query language to search semantically annotated text corpora but not for semantic web data such as RDF triples. Moreover, the inverted index also can be used in DBMSs to support containment queries in XML documents [15, 16].

3 Overview of Semplore

Our work is based on [9], which indexes and retrieves RDF triples using the existing index structure and functions of current IR engines. It provides the hybrid query capability by combining both structured queries and keyword queries. In this section, we will give a brief introduction to its query capability, index structure and query evaluation algorithm.

3.1 Hybrid Query Capability

Here the hybrid query is an extension of the DL-based conjunctive query which was introduced in [17] and can be presented by the SPARQL query language. To support the keyword queries, an extension of the ordinary conjunctive query is made by taking keyword as a virtual concept. An individual is an instance of a certain virtual concept if the textual content of its properties contain the corresponding keyword. Then the users can input a conjunctive query containing

Table 1. Translation from semantic web data to fields, documents, and terms

Document	**Field**	**Term**
concept C	subConOf	super-concepts of C
	superConOf	sub-concepts of C
	text	tokens in textual properties of C
relation R	subRelOf	super-relations of R
	superRelOf	sub-relations of R
	text	tokens in textual properties of R
individual i	type	all concepts that i belongs to
	subjOf	all relations R that $(i, R, ?)$ is a triple in data
	objOf	all relations R that $(?, R, i)$ is a triple in data
	text	tokens in textual properties of i

keyword constraints. For example, to find all films which are about "romantic" and directed by some Chinese director, the query is:

$\{f \mid \texttt{"romantic"}(f) \wedge \texttt{directs}(d, f) \wedge \texttt{ChineseDirector}(d)\}$

Here the queries are restricted as tree-shaped unary queries, whose query graphs are trees. The detailed definition can be found in [9].

3.2 Index Structure

The index structure of traditional IR search engine is the inverted index which based on fields, documents and terms. Work in [9] uses the inverted index structure to index triples and provides searching and querying based on the functions of IR search engine. Its main idea is to translate semantic web data into documents, fields and terms which can be indexed and retrieved by traditional IR engine. The translation is shown in Table 1.

After the translation, semantic web data can then be indexed by the IR engine. The IR engine's retrieval functions can also be used over these indexed data. For example, for each relation, the IR engine can find all its super relations by inputting the relation name and the field "superRelOf" as a query. For each concept, the IR engine can also return all its individuals by taking the concept name and the field "type" as a query.

For relation triples, the index saves relation names as terms and the subject individuals as the documents. As it is shown in Fig.1, for each subject in a certain relation's posting list, its position list stores all its corresponding objects in this relation. As an example in Fig.1, i_2 and i_3 are corresponding objects of i_1, then i_2 and i_3 are stored in i_1's position list in relation R_1's posting list. Thus the IR engine can find all the objects of a certain relation with a given set of subjects by return the union of the corresponding position lists. The index structure is symmetric, for the objects of a relation can be taken as the subjects of the inverse relation. So in the similar way, objects of triples are stored in the posting list of inverse relations and they also have position lists to stored the corresponding subjects.

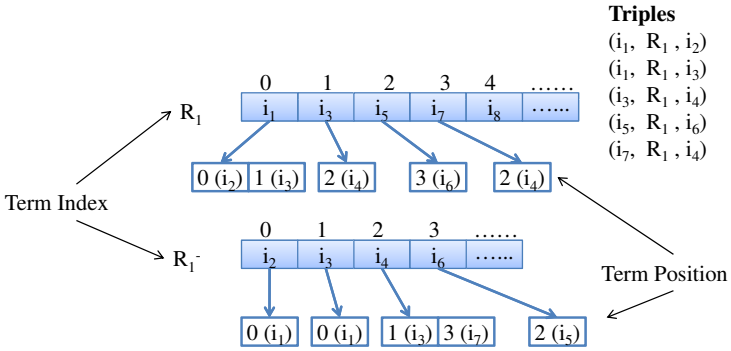

Fig. 1. Triples stored in the inverted index

To force the search engine to save the object individuals as position information, the actual contents stored in the position lists are the object's local positions in the inverse relation's posting list. For example in Fig.1, in i_1's position list stores 0 and 1, which are i_2 and i_3's local position in the posting list of R^-. By reading the subject's position list the search engine can quickly skip to the corresponding objects in the inverse position list. Based on this index structure, the search engine can provide efficient query evaluation algorithm which will be discussed in Section 3.3.

IDs are used throughout the indexing process to uniquely represent a resource (individual). Individuals in the posting lists or position lists are sorted in ascending order according to their IDs in order to provide fast query evaluation.

3.3 Query Evaluation

Basic Operations In modern IR engines, two basic operations can be efficiently achieved, which are the Basic Retrieval and the Merge Sort. Given a field f and a term t, Basic Retrieval (f, t) returns the corresponding posting list from inverted index. The result is sorted by individual IDs in ascending order. The input of Merge Sort are two sorted lists of individual IDs S_1 and S_2 and a binary operator op which can be $\cap$, $\cup$ or $-$. The Merge Sort operation $m(S_1, op, S_2)$ computes $S_1\, op\, S_2$ by merging the lists S_1 and S_2 and returns the result as a new sorted list of individual IDs. According to the index structure mentioned in Section 3.2, works in [9] reuses and extends these basic functions of IR engine to support its own query evaluation algorithm.

(1) Concept Constraints

The input of this operation is a boolean combination of concepts and keyword concepts. It's output is a sorted list of individual IDs which match the constraints. This operation can be implemented using basic retrieval and merge-sort operation mentioned above. For example, for the input `Film` $\sqcap$ "romantic",

the Concept Constraints can be achieved through two Basic Retrievals and one Merge Sort: $m(\,(\texttt{type}, \texttt{Film}), \cap, (\texttt{text}, \text{“romantic”})\,)$.

(2) Relation Expansion

The input of this operation is a relation R and two sets S_1 and S_2 of individual IDs. The operation computes the set $\{y \mid \exists x : x \in S_1 \wedge (x, R, y) \wedge y \in S_2\}$ and returns it as a sorted list of individual IDs. The Relation Expansion is not directly supported by traditional IR engines. This operation needs to find all the objects of a certain relation with a given set of subjects. According to the index structure in 3.2, these objects can be obtained by computing the union of the subjects' position lists. Since in these position lists it stores the objects' local position in the inverse position list, the union can be computed based on a bit vector which has the same length as the inverse relation's posting list.

Query Evaluation Algorithm From these basic operations, a tree-shaped hybrid query can be evaluated using a bottom-up method. At first for each leaf nodes in the query graph, it uses the Concept Constraints operation to obtain the satisfied individuals. When all of the children nodes are evaluated, it moves forward to the parent node using the Relation Expansion to filter the results. Then these children nodes are removed. Doing this procedure iteratively then the final result is obtained when finish visiting all the edges in the query graph.

4 Index Update Mechanisms

Based on the index structure in Section 3.2, for concept names, relation names or concept individuals which are indexed without using position lists, we can update the index by adopting the optimizations of traditional index maintenance([3, 6]). However, for relation triples, updating the index is time consuming. During the index update, newly arrived triples need to be added into the index. Their subjects and objects are inserted into the posting lists of corresponding relations and inverse relations. However, inserting new individuals into a posting list would make some of the original individuals' local position moved behind. These affected local positions are stored in the position lists in the inverse relation's posting lists. As a result, these position lists which store the updated local positions should be updated. It is certainly a heavy cost, since inserting one individual may sometimes leads to the reconstruction of the whole posting list. In this section, we present a Block Index structure based on Section 3.2. It reduces the cost of inserting new relation triples into the index.

4.1 Block Index Structure

The purpose of our Block Index structure is to minimize the changes in the position lists when inserting relation triples. The basic idea is to split posting lists into blocks. The first individual of each block is taking as landmark. All

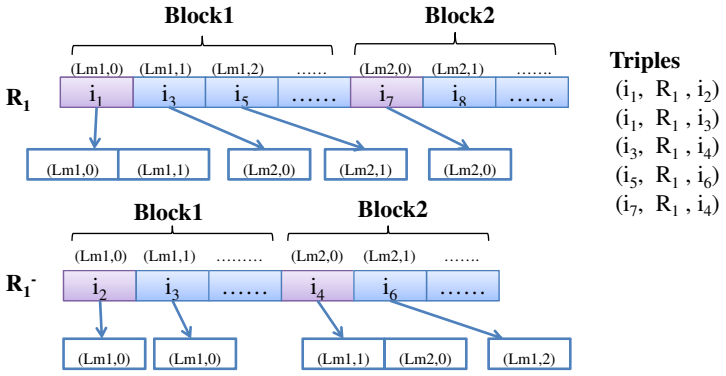

Fig. 2. Structure of block based index

Algorithm 1: Single Update Algorithm

Input: An inserted triple (s, R, o)
1 **if** $s \notin Posting(R)$ **then** $Insert(s, R)$;
2 **if** $o \notin Posting(R^-)$ **then** $Insert(o, R^-)$;
3 Add $LocalPosition(R, s)$ to $PositionList(o, R^-)$;
4 Add $LocalPosition(R^-, o)$ to $PositionList(s, R)$;

individuals in the same block have their offsets comparing their local position to that of the block's landmark. Then the local position of each individual in the posting list can be presented as a $< Landmark\_ID, offset >$ pair. Take Fig.2 as an example, in the posting list of relation R_1, individuals i_1 and i_7 are landmarks of $Block_1$ and $Block_2$ respectively. Then the local position of i_5 can be represented by the pair $< Lm1, 2 >$ where $Lm1$ is the landmark_ID of $Block_1$ and 2 is the offset. An auxiliary landmark table is needed to store all the landmarks and their real position in the posting lists. Thus real positions of individuals in a posting list can be obtained by getting the landmark's real position from landmark table and adding the offset value.

Note that the Block Index structure is only for the storage of relation triples. For concept names, relation names or concept individuals, which are only stored in posting lists, the index structure is the same as it defines in Section 3.2.

4.2 Single Update Operation

In this section, we present the algorithm for inserting a single triple into the index, which is shown as Algorithm 1. First we insert the subject into the relation's posting list if necessary. Second is to insert the object into the inverse relation's posting list in a similar way if necessary. After the insertion, we get the local positions of the subject and the object in corresponding posting lists. Then we can add their local positions into each other's position list.

Procedure `Insert(`s`,`R`)`

1 Find block B that s should inserted into;
2 **foreach** *instance* i *that* $i \in B \wedge i > s$ **do**
3 **foreach** $< p_l, p_o > \in PositionList(i, R)$ **do**
4 $o = Skip\_To(< p_l, p_o >, Posting(R^-))$;
5 Find $< n_l, n_o >$ in $PositionList(o, R^-)$ that $Skip\_To(< n_l, n_o >, Posting(R)) == i$;
6 $n_o = n_o + 1$;
7 Add s to Posting(R);
8 Update the Landmark Table;

The procedure of $Insert(s, R)$ is to insert an individual s into the posting list of a relation R. For each subsequence individual i in the Block, we read every local position ($< p_l, p_o >$) in it's position list to find the corresponding object o in the inverse-relation list. Then we update the old local position of i which is stored in o's position list by increasing the offset value by one. We should also maintain the landmark table after the insertion. Since the insertion of s makes the real positions of all landmarks of the subsequence blocks moved afterward for one space.

Fig.3 shows the procedure of $Insert(i_2, R_1)$ when inserting a single triple (i_2, R_1, i_7) into the index. Since the subject i_2 does not exist in the posting list of R_1, i_2 is then inserted into $Block_1$ of R_1's posting list. The local position of all subsequence individuals in this block (i_3 and i_5) should be updated by increasing the offset value by one. The old local positions of i_3 and i_5 should be updated, which are stored in the position list of their corresponding objects. By reading the position list of i_3 and i_5 in R_1's posting list, we can easily get these corresponding objects (i_4 and i_6) and skip to their positions in the posting list of R_1^-. Then we can find in i_4 and i_6' position lists and update i_3 and i_5's old local position by increasing the offset value by one. In the similar way, the object i_7 is inserted into $Block_2$ of R_1^-'s posting list and all the local positions of subsequence individuals in this block have to be updated. As a result, their old local positions which are stored in the position list of the corresponding subjects in R_1 should be changed.

The delete operation is much easier, for we only need to delete the local positions of the subject and the object in each other's position lists. Considering the expenses of inserting an individual in the posting list, we do not delete an individual with empty position list for further insertion.

4.3 Batch Update Operation

In this section we present a batch update operation, which can reduce the number of update operations in position lists for multi-triple's insertion. In the batch update operation, every time we insert all individuals which belong to the same block into the posting list. It will avoid the redundant update in position lists. The Algorithm 3 shows how it works.

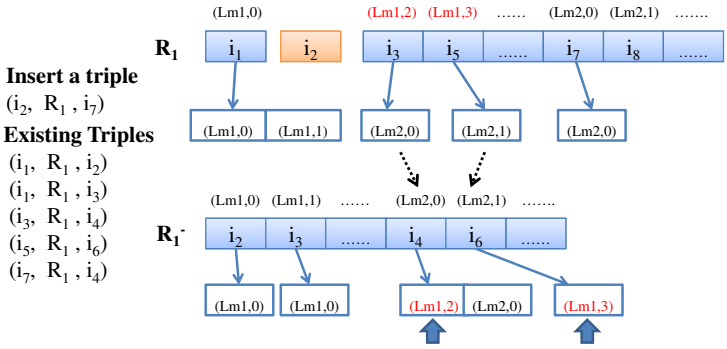

Fig. 3. Procedure of inserting a single individual into posting list

When more triples of a relation is inserted into the index, each time we select all the individuals which belong to the same block and insert them into the posting list at one time. The operation of $BatchInsert(S, B, R)$ is an expansion of $Insert(s, R)$ presented in Section 4.2. Once inserting these individuals in the posting list, the local position of every original individual in the block which behind the minimum inserted individual should be moved afterward. These local positions are stored in the position lists of the corresponding objects in the inverse relation's posting list. The offset value are updated due to the number of individuals which will be inserted in front of it. The batch insert operation is more efficient since it reduce the number of position update when inserting individuals belong to the same block.

Algorithm 3: Batch Update Algorithm

Input: Inserted triples (s_1, R, o_1),(s_2, R, o_2) ... (s_n, R, o_n)

foreach *block* $B_i \sqsubseteq Posting(R)$ **do**
  $S_{sub} = \{s_t \mid s_t \notin Posting(R) \wedge s_t \geq Landmark(B_i) \wedge s_t < Landmark(B_{i+1})\}$;
  $BatchInsert(S_{sub}, B_i, R)$;
foreach *block* $B_i \in Posting(R^-)$ **do**
  $S_{obj} = \{s_t \mid s_t \notin Posting(R^-) \wedge s_t \geq Landmark(B_i) \wedge s_t < Landmark(B_{i+1})\}$;
  $BatchInsert(S_{obj}, B_i, R^-)$;
foreach (s_i, R, o_i) **do**
  Add $LocalPosition(R, s_i)$ to $PositionList(o_i, R^-)$;
  Add $LocalPosition(R^-, o_i)$ to $PositionList(s_i, R)$;

In order to be efficient for the update operation, the block size must be chosen in a certain range, which will be discussed in Section 5.3. After the batch update, if a block size exceeds the threshold, it will be splited into two smaller blocks.

Procedure `BatchInsert(S,B,R)`

1 **foreach** *instance* $i \in B \wedge i > min\{S\}$ **do**
2 **foreach** $< p_l, p_o > \in PositionList(i, R)$ **do**
3 $o = Skip\_To(< p_l, p_o >, Posting(R^-))$;
4 Find $< n_l, n_o >$ in $PositionList(o, R^-)$ that $Skip\_To(< n_l, n_o >, Posting(R)) == i$;
5 $n_o = n_o + |\{s \mid s \in S \wedge s < i\}|$;
6 Add each $s \in S$ to $Posting(R)$;
7 Update the Landmark Table;

5 Performance Analysis

5.1 Space Requirement

According to [7], the landmark-offset encoding for local position does not increase the space requirement of the index. Suppose k bits are allocated for a location position in the posting list, then the same k bits can be used to encode a $< landmark\_ID, offset >$ pair with $b < k$ bits for the landmark ID and the rest $k - b$ bits for the offset. However, an extra landmark table will be stored on disk and loaded in memory during index update and query processing. The size of landmark table is usually small. For a index with average block size B, the total number of landmarks is $\Sigma(\lceil 2L_R/B \rceil)$,where L_R is the number of triples of relation R.

5.2 Query Performance

In essence, query evaluation time with block index is not significantly affected comparing to the index structure mentioned in 3.2. For concept individuals, which are stored in traditional inverted index without blocks, the IR engine provides fast processing time. For relation triples, the main difference is that the individual's real positions in the posting lists needs to be computed by seeking the real position of the landmark in the landmark table and adding the individual's offset. In the landmark table, all landmarks in a certain relation's posting list is sorted by their real positions. Using the binary search, the seek operation takes $O(log(L/B))$ time, where L is the length of posting list and B is the average block size.

5.3 Index Update Time

In the block based index, contents in the same block are stored continuously. Based on the optimization of traditional index maintenance([3, 6]), inserting individuals into a block is not time consuming. Moreover, with the help of landmark table, seeking in the posting lists can be finished efficiently. During the index update, the main cost is to look up and update the offset values in the position lists. Since contents of a position list are physically stored continuously

Table 2. Triples of real world dataset

Dataset	Version 2.0	Version 3.0	Percentage
No. of triples	557,126	569,051	-
Inserted triples	-	157,127	28.2%
Deleted triples	-	145,280	26.1%

in modern IR engines, the look up operations in the position lists enjoys the benefit of spatial locality for fast access.

Using the batch update introduced in Section 4.3, the total index update time for all newly arrived triples depends on the block size and the number of blocks which will be inserted with new individuals. So our update mechanism is especially efficient when the index size is large and the number of update triples is small. Blocks with larger size lead to more individuals whose local positions are affected during the insertion. Thus it increases the number of update operations in the position list. While block with too small size will decrease the efficiency of both index updating and query processing. That's because the individuals' real positions are computed by looking up the landmark table. Small blocks will increase the size of landmark table and thus slow down the look up operation. Small blocks also produce many fragments on disk which will affect the disk access time. In our experiment in Section 6.2, we will demonstrate and further discuss the impact of block size to the index update.

For concept names, relation names or concept individuals, which only stored in posting lists, the index structure is the same as it defines in Section 3.2 and we update the index by using the existing optimizations for traditional IR index maintenance([3, 6]). Since the update in these posting lists are infrequent and less time costing comparing to the update of triples.

6 Evaluation

6.1 Experiment Setup

We use both the real world data and the artificial semantic data in our experiment. In order to simulate the data change on the Semantic Web, a representative NTriple file (persons.nt) from DBpedia is used as the real world data. Table 2 shows its content update during a time interval of five months [3].

In order to test the efficiency of our index update mechanism and its impact on query answering, we also use the LUBM [18] benchmark data. In the LUBM dataset, data is randomly generated and can be scaled to an arbitrary size. For each dataset from LUBM(1,0) to LUBM(20,0), we treat its content as triples to be inserted into an existing block index which is build for the LUBM(50,0) dataset. Table 3 shows the number of triples from LUBM(1,0) to LUBM(50,0).

[3] The DBPedia 2.0 version was launched in 09/2007 while the 3.0 version was launched in 02/2008

Table 3. Triples of artificial datasets

Dataset	LUBM(1,0)	LUBM(5,0)	LUBM(10,0)	LUBM(15,0)	LUBM(20,0)	LUBM(50,0)
No. of triples	102,737	643,435	1,311,787	2,014,462	2,772,017	6,865,225

Table 4. Index construction for persons.nt v2.0

Person.nt v2.0 from DBpedia	Semplore	Block Index
Index Construction Time (s)	218	231
Index Space (MB)	37	38

In Section 6.3 we also evaluate the query processing time under LUBM(20,0) and LUBM(50,0).

The proposed experiments are carried out on a desktop PC with Pentium 4 CPU of 3.2 GHz and 2Gb memory, running Microsoft Windows Server 2003 with Sun Java JRE 1.5.0. Note that single indexing thread is used in our experiment to obtain a raw indexing speed for ease of comparison.

6.2 Index Update Performance

In this section, we evaluate the efficiency and scalability of our index update mechanism. Table 4 shows both the index construction time and index space size for persons.nt v2.0 using the two different index structures. Semplore [9] is based on the index structure already introduced in Section 3.2. Note that our proposed block index only slightly increases the index construction time. Moreover, the same conclusion can be drawn on the size of index space, which indicates that it would not lead to the space overhead.

When updating the index to persons.nt v3.0, we choose different block size to test the performance. As shown in Fig. 4, we can see that when the block size is increasing, the update time increases. Blocks with larger size lead to more individuals whose local positions are affected during the insertion and thus increase the number of update operations. However, when the block size is chosen as 50, it took more time to update the index than with block size equals to 100. The main reason is that getting individuals' real position needs to look up the landmark table. When the block size is small, the size of landmark table is increased and thus the index update time is slow down. Since Semplore does not provide the index update mechanism, it's index can only be updated by complete reconstruction, which takes 225 seconds.

For the LUBM dataset, we first build the index based on the Block Index structure under the dataset of LUBM(50,0). Here we chose the block size equals to 1000. For every dataset from LUBM(1,0) to LUBM(20,0), we take it as the set of triples be to inserted into the existing Block Index. Then we insert each of them into the original index of LUBM(50,0) to evaluate the index update time. For Semplore, we rebuild the index for the original dataset LUBM(50,0) together with the new inserted dataset. The results are shown in Fig.5. We can find that

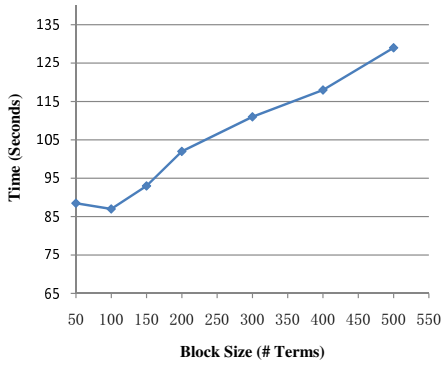

Fig. 4. Index update time with different block sizes

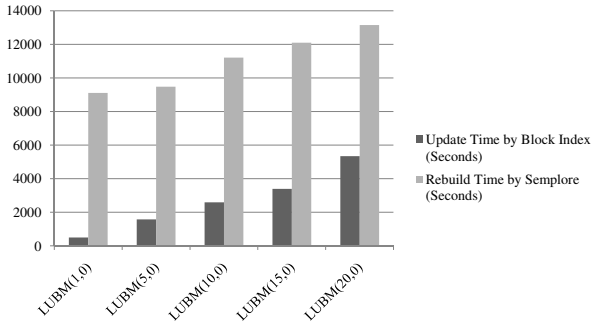

Fig. 5. Index update time vs. index rebuild time

the index update time is a fraction of complete rebuild due to the portion of the inserted triples.

6.3 Query Response Time

In additional to the efficiency of index update performance, it is also important to test whether the block index structure would largely influence the time of query answering. We choose 8 of 14 LUBM benchmark queries mentioned in [18] for the evaluation, which is shown in Table 5. The excluded queries are either cyclic or with multiple variables which are out of the query capability of Semplore (i.e. unary tree-shaped conjunctive query). Here we only focus on testing the efficiency on retrieval but not the reasoning capability.

Table 6 shows the query response time under LUBM(20,0) and LUBM(50,0) by the two different indices. The block index is built by setting the block size as 1000. The response time of the block index is slightly slower than that of Semplore as it needs to lookup the landmark table stored in the main memory when returning the individuals local position. Moreover, the retrieval time is

Table 5. LUBM benchmark queries

Q1	(type GraduateStudent ?X) (?X takesCourse Department0.University0.GraduateCourse0)
Q3	(type Publication ?X) (?X publicationAuthor Department0.University0.AssistantProfessor0)
Q5	(type Person ?X) (?X memberOf Department0.University0)
Q6	(type Student ?X)
Q10	(type Student ?X) (?X takesCourse Department0.University0.GraduateCourse0)
Q11	(type ResearchGroup ?X) (?X subOrganizationOf University0)
Q13	(type Person ?X) (University0 hasAlumnus ?X)
Q14	(type UndergraduateStudent ?X)

Table 6. Query Response Time for LUBM Datasets (ms)

Query	LUBM(20,0)		LUBM(50,0)	
	Semplore	Block Index	Semplore	Block Index
Q1	14	63	14	84
Q3	0	43	0	51
Q5	0	32	0	43
Q6	16	19	31	34
Q10	0	43	0	47
Q11	0	0	0	0
Q13	0	39	13	56
Q14	0	0	32	36

almost the same as Semplore when queries are tend to find individuals of concepts or keywords. This is due to the fact that the corresponding posting lists do not use the position lists thus are not stored in the block index. This way, the block index is proved to provide much more flexibility for index update mechanisms while preserving the efficiency of query answering.

7 Conclusion

In this paper, we present a well-designed update mechanism on the state of the art IR index (Semplore) for triples. Benefited from the basic idea of dividing the index into blocks, it reduces the number of update operations during inserting triples, which results in several orders of magnitude decrease on the index update time compared to that by complete reconstruction. Moreover, both the size of index space and query response time are not notably effected. Thus, our proposed mechanism makes it possible for Semplore to handle frequent semantic data update while preserving efficient hybrid query answering. One future work we are considering is to offer more suitable block sizes and update strategies to meet the requirements of different situations in order for self-tuning.

References

1. Ding, L., Finin, T., Joshi, A., Pan, R., Cost, S.R., Peng, Y., Reddivari, P., Doshi, V., Sachs, J.: Swoogle: a search and metadata engine for the semantic web. In: Proceedings of the 13th ACM CIKM, New York, NY, USA, ACM Press (2004) 652–659
2. d'Aquin, M., Baldassarre, C., Gridinoc, L., Sabou, M., Angeletou, S.: Watson: Supporting next generation semantic web applications. In: WWW/Internet conference. (2007)
3. Tomasic, A., García-Molina, H., Shoens, K.: Incremental updates of inverted lists for text document retrieval. (1994) 289–300
4. Brown, E., Callan, J., Croft, W.: Fast incremental indexing for full-text information retrieval. In: Proceedings of the 20th International Conference on Very Large Databases (VLDB), Santiago, Chille (1994) 192 – 202
5. Büttcher, S., Clarke, C.L.A., Lushman, B.: Hybrid index maintenance for growing text collections. In: SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, New York, NY, USA, ACM (2006) 356–363
6. Lester, N., Zobel, J., Williams, H.: Efficient online index maintenance for contiguous inverted lists. Inf. Process. Manage. **42**(4) (2006) 916–933
7. Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Agarwal, R.C.: Efficient update of indexes for dynamically changing web documents. In: World Wide Web. (2007) 37–69
8. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems **30**(1–7) (1998) 107–117
9. Zhang, L., Liu, Q., Zhang, J., Wang, H., Pan, Y., Yu, Y.: Semplore: An ir approach to scalable hybrid query of semantic web data. In: ISWC/ASWC '07. (2007) 652–665
10. Lempel, R., Mass, Y., Ofek-Koifman, S., Sheinwald, D., Petruschka, Y., Sivan, R.: Just in time indexing for up to the second search. In: CIKM. (2007) 97–106
11. Jang, H., Kim, Y., Shin, D.: An effective mechanism for index update in structured documents. In: CIKM. (1999) 383–390
12. Rocha, C., Schwabe, D., Aragao, M.P.: A hybrid approach for searching in the semantic web. In: Proceedings of the 13th international conference on World Wide Web, ACM Press (2004) 374–383
13. Nejdl, W., Siberski, W., Thaden, U., Balke, W.T.: Top-k query evaluation for schema-based peer-to-peer networks. In: International Semantic Web Conference. (2004) 137–151
14. Chu-Carroll, J., Prager, J.M., Czuba, K., Ferrucci, D.A., Duboué, P.A.: Semantic search via xml fragments: a high-precision approach to ir. In: SIGIR. (2006) 445–452
15. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: The VLDB Journal. (2001) 361–370
16. Tatarinov, I., Viglas, S., Beyer, K.S., Shanmugasundaram, J., Shekita, E.J., Zhang, C.: Storing and querying ordered XML using a relational database system. In: SIGMOD Conference. (2002)
17. Horrocks, I., Tessaris, S.: Querying the semantic web: A formal approach. In: International Semantic Web Conference. (2002) 177–191
18. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. J. Web Sem. **3**(2-3) (2005) 158–182