Hash Tree Indexing for Fast SPARQL Query in Large Scale RDF Data Management Systems

Wenwen Li[1,3], Bingyi Zhang[1], Guozheng Rao[1,3,*], Renhai Chen[1,3], and Zhiyong Feng[2,3]

[1] School of Computer Science and Technology,Tianjin University, Tianjin 300350, P. R. China,
[2] School of Computer Software,Tianjin University, Tianjin 300350, P. R. China
[3] Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin 300350, P.R. China
[*] Corresponding author.
{lww1204 ,byzhang, rgz ,renhai.chen, zyfeng}@tju.edu.cn

Abstract. In the past decade, the volume of RDF (Resource Description Framework, which is a standard model for data interchange on the Web) data has grown enormously, and many RDF datasets (e.g., Wikipedia) have reached up to billions of triples. As a result, efficient management of this huge RDF data has become a tremedous challenge. In this paper, we present HTStore, a hash tree based system for fast storing and accessing large scale RDF data. The design of HTStore has three salient features. First, the compact design can effectively reduce the size of the indexes. Second, HTStore utilizes the hash function to significantly reduce the query time. Third, the proposed hash tree structure can easily adapt to the changes in data volume (e.g., data expansion). The experimental results demonstrate that the proposed system can improve the query efficiency up to 21.3% compared with the representative RDF data management systems.

1 Introduction

The RDF (Resource Description Framework) data model and its query language SPARQL have been widely used for managing schema-free structured information. Large amounts of semantic data are available in RDF format in many fields, such as Yago[1], DBLP[2], and DBpedia[3]. The statistics from Yago show that more than 100 billion triples were published by September 2015.

Several systems, such as Gstore[4] and RDF-3x[5], have been proposed to support RDF store and SPARQL query. According to the data management method, these systems are generally classified into three categories: relational database based RDF management, RDF triple management, and graph-based RDF data management. Systems based on relational database leverage the relational database to manage RDF data. In such systems, RDF data are stored in the database tables and performed data query using the traditional SQL language. Leveraging mature data management technique of relational database, RDF storing and querying are easy to implement. However, systems based on relational database will destroy the original structure of RDF, and thus introduce a large number of time consuming join operations and waste a lot of storage space. Systems based on triple or RDF graph, such as RDF-3x and RDF Cube, optimize the RDF data management by using B+ tree index or hash index to improve query performance. Although this approach has shown to accelerate joins by orders of magnitude, the lengthy comparison operations and high collision rate with data explosion have become the Achilles' heel of an RDF data management system.

In this paper, we propose HTStore to fast store and access large scale RDF data. In HTStore, we organize RDF data in the form of an RDF graph and establish indexes according to the RDF graph. The index structure includes two layers: the hash layer containing a hash table and the tree layer containing hash trees. More specifically, we construct a hash table for fast lookup. When a hash collision happens in a hash table, new hash trees will be established in the second layer. With such a structure, only limited hash operations are required to perform a data query. As a result, the query time is significantly reduced. We conduct experiments over LUBM datasets to confirm the effectiveness and efficiency of our proposed approach. The experimental results prove the proposed system can improve the query efficiency by 21.3% compared with the representative RDF data management systems.

2 Design of HTStore

In HTStore, we organize RDF data in the form of an RDF graph and establish index according to the vertexes in the graph. Fig. 1 shows an overview of the proposed structure used to manage RDF graph. The left part of the figure is an RDF graph, and the right part is the index built according to the vertexes in the graph. Before building an index in a RDF dataset, each vertex in the RDF graph is assigned a unique identifier. The index structure includes two parts: the hash layer containing a hash table and the tree layer containing hash trees. In the first layer, a m-length hash table is constructed. In the second layer, we build hash trees dynamically based on a prime sequence $P = \{p_1, p_2, p_3 \cdots\}$ during inserting RDF graph vertexes.

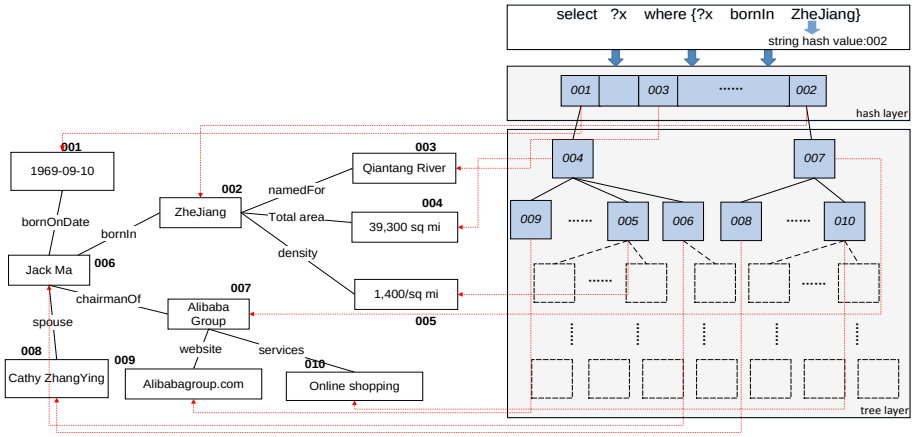
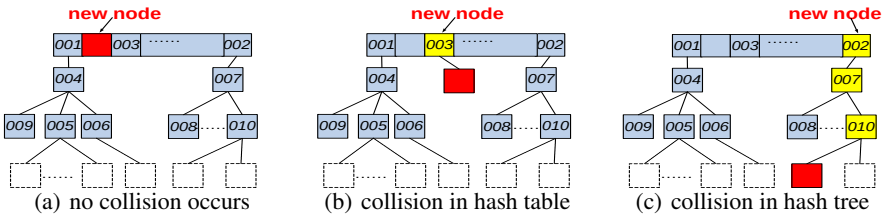


Fig. 1: System view of HTStore

Suppose that we intend to insert a new vertex into an RDF datasets, Fig. 2 shows different solutions for different insertion situations. The simplest case is when no collision happens in hash table. The vertex will be added to the blank bucket directly(as shown in Fig. 2(a)). In Fig. 2(b), collision happens in the hash table and there exists no hash tree of the collided vertex. Therefore, a new hash tree should be established in the second layer. If hash tree of the collided vertex has been constructed(as shown in Fig. 2(c)) and collision still occurs between the new node and the root node of hash tree, according to hash tree's construction regulation, we will leverage first prime number p_1 to obtain a

hash value t of the new node. It means that we will consider the t-th child node of root node – $cnode$. If collision still occurs in the $cnode$, the second prime number p_2 will be used to determine which child node of $cnode$ will be considered. Similar operations by different prime number will be conducted until there is no collision occurs.



(a) no collision occurs (b) collision in hash table (c) collision in hash tree

Fig. 2: Different solutions for different insertion situations

The query processing is similar to the insertion processing in that we determine the location of vertex by several hash operations. Fig. 3 shows the processing of searching the keyword *Online shopping*. Before executing lookups in index, the keyword will be transformed into a integer by string hash function. If we locate a blank vertex, we will terminate the query processing and return null value. As shown in Fig. 4, the deletion is quiet simple. If some vertexes need to be deleted, we will search these vertexes and then just set the corresponding locations to empty without adjustment of index structure.

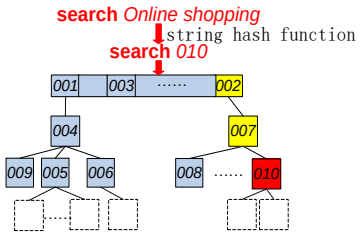


Fig. 3: Querying processing

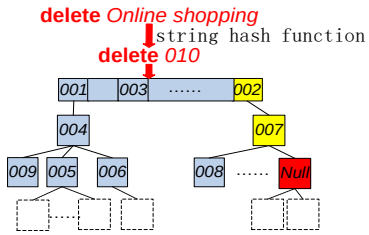


Fig. 4: Deleting processing

According to the intrinsic feature of hash tree, we can create an index for billion data by just a few layers of a hash tree. Owing to the low depth of hash tree, the query processing only need several hash operations. The time complexity of querying is $O(1)$. Moreover, our index is built dynamically. There is no need for a long initialization processing. The simple structure also allow data to be updated without adjustment of the index structure. Therefore, such index is suitable for RDF data management systems.

3 Experiment

All the experiments were conducted on a Dell optiptex 7040 PC with a 3.20 GHz CPU, 16 GBytes of RAM. The operating system is a 64-bit Linux with 4.8.0 kernel. We use LUBM as our datasets. LUBM (Lehigh University Benchmark) is developed to facilitate the evaluation of Semantic Web repositories in a standard and systematic way. It consists of a university domain ontology, customizable and repeatable synthetic data, and several performance metrics. Different LUBM datasets have different sizes and different triple numbers. We also compare our experiments with RDF-3x and Gstore.

Table 1 lists the SPARQL query used in our experiment. We execute these six queries over different LUBM datasets. Table 2 compares the query performance of our method, Gstore and RDF-3x. In our query samples, Gstore always perform better than

Table 1: SPARQL Query Samples

Q1	SELECT distinct ?y WHERE{ ?x uni:worksFor <http://www.Department20.University400.edu >. ?x uni:teacherOf ?z. ?y uni:takesCourse ?z. }
Q2	SELECT ?x ?y ?z WHERE{ ?y uni:teacherOf ?z. ?y rdf:type uni:FullProfessor. ?z rdf:type uni:Course. ?x uni:advisor ?y. ?x rdf:type uni:UndergraduateStudent. ?x uni:takesCourse ?z. }
Q3	SELECT ?x ?y ?z WHERE{ ?z uni:subOrganizationOf ?y. ?y rdf:type uni:University. ?z rdf:type uni:Department. ?x uni:memberOf ?z. ?x rdf:type uni:GraduateStudent. ?x uni:undergraduateDegreeFrom ?y. }
Q4	SELECT ?x WHERE{ ?x rdf:type uni:Course. ?x uni:name ?y. }

RDF-3x. While the quantity of RDF data is small, the query performance of our method is not obvious, almost as fast as Gstore. When the amount of data increases, the query efficiency is improved significantly. HTStore can improve the query efficiency up to 21.3% compared with Gstore.

Table 2: Query Performance on LUBM

Query	Query Response Time (msec)								
	LUBM100			LUBM500			LUBM800		
	HTStore	Gstore	RDF-3x	HTStore	Gstore	RDF-3x	HTStore	Gstore	RDF-3x
Q1	53	49	55	158	170	193	261	285	294
Q2	287	392	410	1265	1779	2031	2019	2749	3397
Q3	568	834	8471	9471	15927	38680	19895	30091	58716
Q4	523	701	1692	1876	2096	3415	2864	3413	5562

4 Conclusion

In this paper, we propose HTStore to manage large scale RDF data. HTStore utilizes the hash tree structure to significantly reduce the query time. In addition, the proposed management scheme can also easily adapt to the changes in data volume. Experimental results demonstrate that HTStore can effectively improve performance of SPARQL query.

Acknowledgement

This work is supported by the programs of the National Natural Science Foundation of China (61373165 and 61702357).

References

1. Farzaneh Mahdisoltani, Joanna Biega, and Fabian Suchanek. Yago3: A knowledge base from multilingual wikipedias. In *7th Biennial Conference on Innovative Data Systems Research*. CIDR Conference, 2014.
2. Michael Ley. Dblp: some lessons learned. *Proceedings of the VLDB Endowment*, 2(2):1493–1500, 2009.
3. Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Chris Bizer. DBpedia - A Large-scale, Multilingual knowledge Base Extracted from Wikipedia. *Semantic Web Journal*, 2014.
4. Lei Zou, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao. gStore: answering SPARQL queries via subgraph matching. *Proceedings of the VLDB Endowment*, 4(8):482–493, 2011.
5. Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.