xStore: Federated temporal query processing for large scale RDF triples on a cloud environment

Jinho Ahn, Jae-Hong Eom, Sejin Nam, Nansu Zong, Dong-Hyuk Im, Hong-Gee Kim

A Biomedical Knowledge Engineering Lab. & Dental Research Institute, School of Dentistry, Seoul National University, Republic of Korea
B National Center of Excellence in Software, Chugnam National University, Republic of Korea
C Department of biomedical informatics, University of California, San Diego
D Department of Computer and Information Engineering, Hoseo University, Republic of Korea

Abstract

Temporal information retrieval tasks have a long history in information retrieval field and also have attracted neuroscientists working on memory system. It becomes more important in Semantic Web where structured data in RDF triples, often with temporal information, are rapidly accumulated over time. Existing triple stores already support loading RDF triples and answering a given SPARQL query with time interval constraints. However, few triple stores have been optimized for processing time interval queries which are important for temporal information retrieval tasks. In this paper, we propose xStore, a federated SPARQL engine running on a cloud environment, which supports a fast processing of temporal queries. xStore is built on top of heterogeneous storages such as key-value stores and conventional triple stores. Experiments over real-world temporal datasets showed that our approach is faster than a conventional SPARQL engine for processing temporal queries.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

Temporal information retrieval tasks have a long history in information retrieval field [1]. Neuroscientists and cognitive scientists also pointed out that events (e.g. temporal information) are one of important part of declarative memory [2,3]. Temporal information retrieval here can be defined as finding some information associated with events that occur at given time constraints. It becomes more important as a vast amount of structured data is made available on the Web, which is often called Semantic Web [4]. RDF (Resource Description Framework) is one of standards for Semantic Web to represent information in triples (subject, predicate and object). With the popularity of RDF, Linked Data, a community effort to maintain RDF triples, begins to gain interest from general public.

Extensive research has been conducted on temporal data processing, representation and management in Linked Data [5–7]. However, little attention has been paid to temporal query processing for large scale RDF data in a cloud computing environment. Cloud computing in the context of this paper refers to distributed computing where heterogeneous systems are operating in a collaborative manner to do specific tasks [8,9]. It has successfully been applied to diverse fields such as collaborative working environment [10,11], personal tutoring [12], distributed image processing [13], big data processing [14], Semantic Web [15] and so on. In this paper, we address temporal query processing by utilizing a cloud computing environment.

Temporal information can be represented in RDF triple like (Turing, birthYear, 1912) that describes the fact that “Turing was born in 1912”. The predicate birthYear can be regarded as a temporal predicate since its range belongs to time. SPARQL, a query language for RDF triple, can be used to express a temporal information retrieval task over Linked Data. For example, the user mights want to find out a place where a person was born who is active between 2000-01-01 and 2013-12-31. That query is written in SPARQL query as follows:
SELECT * WHERE {
    FILTER (?s = "2000-01-01"^^xsd:date &&
          ?s <= "2013-12-31"^^xsd:date &&
          ?e >= "2000-01-01"^^xsd:date &&
          ?e <= "2013-12-31"^^xsd:date) 
}

Temporal constraints are expressed in FILTER constructs to restrict birthDate and deathDate predicates by a time interval (i.e. 2000-01-01 – 2013-12-31). Existing triple stores can be used to store RDF triples and search for desired temporal information using this SPARQL query. However, these are not optimized to process FILTER constructs intended for full-text search and numerical comparison [16,17]. Furthermore, no user-customizable functionality is provided by these triple stores that allow users to choose predicates (e.g. temporal predicates) in order to guarantee a good performance for a particular query-workload (e.g. temporal queries are dominant). The importance of this kind of query workload awareness approach has been mentioned in [18] showing that no triple store can do the best for every query type. As temporal information retrieval task is important in open domain, we focus on temporal query processing.

In this paper, a federated SPARQL engine called xStore is proposed, which is designed for temporal query processing based on a cloud environment consisting of heterogeneous storages. xStore is designed to run on top of key-value stores and triple stores to exploit each one’s strengths. Key-value stores have an advantage over triple stores in processing simple queries faster. Triple stores are better in dealing with complex queries and maintenance issues. In addition, the nature of key-value stores make them suited to operate on a cloud environment that can easily be scaled to handle large volume of data by distributing data across multiple machines.

The rest of this paper is organized as follows. The related work is discussed in Section 2. We explain the system architecture and workflow of xStore in Section 3. Experimental evaluation is discussed in Section 4. We conclude this paper in Section 5.

2. Related works

Triple store on a single machine is categorized into non-native and native [19]. The non-native ones store triples in a relational data model while the native ones have their own data structure. A straightforward way of using relational data model is to introduce a table with three columns (subject, predicate, object). Each triple occupies one row. One can easily observe that the same subject redundantly appears across multiple rows. In addition, to evaluate SPARQL queries, it involves costly self-joins. Property Table approach was proposed to store subject one-time. Unlike the straightforward approach, the table schema varies, i.e. one column for subject and the other columns for object binding to its associated predicate. In other words, the value of the predicate column is filled by its object. The drawback is that it is not easy to optimize query processing because of its various table schema. Abadi et al. [20] proposed Vertical Partitioning approach where tables are created for each predicate. In this case, all the tables has the same schema with two columns (subject, object), which helps process subject–subject join. However, to insert new triple, it first needs to search for tables corresponding to its predicate.

A representative native approach is RDF-3X [21], which compresses triples by only storing deltas between ordered triples and keeps in B+Trees. Exhaustive indexes of SP0 permutation of B+Trees are built to provide a fast query processing. gStore [22] encodes triples in bitstrings which then put into a tree structure, enabling SPARQL query processing by bit-wise operations. However, these works are not designed for a distributed environment. MapReduce-based approaches have been proposed [23,24] to utilize a distributed environment.

A federation over existing storages on a cloud has been adopted for triple stores [25]. Sesame Lucene Sail [16] uses a Lucene engine to store text in object and Sesame is used for the other triples. The rationale is that full text search is more efficient in Lucene than Sesame. Filt [17] follows the same line that uses Jena TDB instead of Sesame. Filt showed that a numerical search is also better in the hybrid approach than using conventional triple store alone. Making use of key-value stores is another option [26], which is however for relational database. There has been works on key-value storages for triple store [27–30]. We combine these approaches to realize an optimized temporal information retrieval against a large scale RDF triples.

3. System architecture

xStore is a federated SPARQL engine based on heterogeneous backend storages in a distributed environment. It is designed to deal with triples differently according to the predefined types of triple. Backend storages are chosen and customized for each triple type. If we know in advance that the input data have two triple types, we would configure xStore to have two different kinds of backend storages. The configuration is determined in the installation and deployment time. In this paper, we only consider the case where two triple types are available such as temporal and general (i.e. the other triples except temporal ones). A key-value store is used for temporal triples as it provides a better performance for simple queries (i.e. without join) than relational database management systems and conventional triple stores. General triples are maintained in conventional triple stores that support a standard SPARQL query processing.

Fig. 1 depicts a data-flow of xStore. The input and output are similar with conventional triple stores. A RDF file in N-Triple format is taken as input which represents one triple per one line. Triples read from the file are stored into the distributed backend storages. A SPARQL query given by user is federated over the backend storages to search for matched triples. The results from each backend storage are merged in the federation engine and then given to user. The detailed steps are illustrated in the following sections.

3.1. Storing

xStore stores the input triples into backend storages in a distributed fashion. Fig. 2 depicts how to store triples. For each triple in N-Triple file, Triple Type Filter determines its triple type. The process is done by comparing the predicate with a list of pre-selected predicates. Several predicates (e.g. birthYear, deathYear) would belong to a list of pre-selected temporal predicates. The data type (e.g. xsd:dateTime) of the value in object position in a triple might be a clue to automatically determine the type of encountered triple. In real-world dataset, however, it is not always guaranteed that every object of each triple be tagged with its data type. Triples that have been determined as temporal type go to the key-value store, which stores a triple in key-value pairs. For example, the following two triples

(Turing,birthYear,1912)
(Turing,deathYear,1954)

are stored in key-value pairs as follows:

{(subject,Turing),(birthYear,1912),(deathYear,1954)}
The other triples go to the triple store. In real-world cases, temporal triples are smaller than the other triples (e.g. 3% in DBpedia 3.9 Mapping-based Properties). It allows us to distribute only general triples to several backend storages. Triple Distributor depicted in Fig. 1 and also in the bottom part of Fig. 2 applies a hash function to subject URI to determine a target backend storage that stores the triples. In this manner, general triples are stored across several storages disjointly.

As each key-value store and triple store runs on different machines, this way of storing triple allows triples to be stored in each machine in a parallel fashion.

3.2. Querying

A federated query processing method is employed in xStore to process a SPARQL query over multiple backend storages. Since a key-value store is used as a backend storage that does not support join evaluation, we implemented a join method. There exist well-known join methods in database community such as hash join, merge join and nested loop join. The hash join is only for equi-join which is not designed for inequality. The merge join is required to sort the joining data before it joins by merging. The method is not appropriate for Linked Data which is big and changes frequently. We decided to adopt the nested loop join method, which expands intermediate results obtained by scanning each table step by step like graph exploration.

Fig. 3 depicts a data-flow of query processing in a nested loop join fashion. A SPARQL query given by user is decomposed into temporal-related ones and the other ones. The example query in Fig. 3 is of a time interval query intended to retrieve a list of birth place of person who lives between 1900 and 2000. That can be represented in SPARQL with FILTER constructs: both ?birthDate and

---

**Fig. 1.** Data flows of xStore implemented on top of backend storages. xStore reads N-Triple files to load triples and accepts a SPARQL query to perform a search.

**Fig. 2.** An example data-flow of storing triples into xStore. Triples are stored into different storages according to the triple types determined by its predicate.
\texttt{?deathDate} should be greater than and equal to 1900 and less than and equal to 2000 as shown in Table 2. During a query processing, the temporal constraint is converted into a key-value store query with inequality. The key-value store then returns a list of subject URIs whose value of \texttt{birthDate} and \texttt{?deathDate} predicate is matched to the given time interval. xStore then constructs a normal SPARQL query consisting only of BGPs (Basic Graph Patterns) made from the subject URIs returned from key-value store. By applying a hash function, BGPs are put into different SPARQL queries sent to distinct triple stores. Results returned from the backend triple storages are merged in Triple Aggregator to give the final answers to the user.

4. Evaluations

Experiments were conducted to evaluate our approach compared to existing systems over real RDF and syntactic datasets. The choice of vendor of backend storages is not specified in the architecture of xStore. Any key-value store and triple store can be used as backend storages for xStore. In this experiment, MongoDB 2.6.5\textsuperscript{1} and Virtuoso 7.0.0\textsuperscript{2} were used as key-value store and triple store, respectively. MongoDB and Virtuoso are well known open-source systems that are easy to be utilized. xStore runs on two machines, with a MongoDB and a Virtuoso instance. A baseline system called VirtTwo indicates a cloud based Virtuoso running on two machines, each of which has a single Virtuoso instance. As a cloud based Virtuoso is only commercially available, we implemented an endpoint that stores triples into each single Virtuoso instance disjointly and answers a given SPARQL query by interacting with the two Virtuoso instances. The systems are installed in two physical machines operating Ubuntu 14.04.1 64bit, each of which has two physical 2.30GHz Xeon CPUs and 94GB RAM.

4.1. Datasets

The datasets used in this experiment were obtained from DBpedia 3.9 Mapping-based Properties. The other ones (e.g. D2, D4 and D8) are syntactic datasets generated from D1. The right part shows the required loading time in minutes and space requirements in gigabytes for each method.

<table>
<thead>
<tr>
<th>Datasets</th>
<th>Total</th>
<th>Temporal</th>
<th>Method</th>
<th>Loading(min)</th>
<th>Space(GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1 (DBpedia 3.9)</td>
<td>26M</td>
<td>0.4M</td>
<td>xStore</td>
<td>9.8</td>
<td>2.7 (0.5)</td>
</tr>
<tr>
<td>D2</td>
<td>77M</td>
<td>1.4M</td>
<td>VirtTwo</td>
<td>6.5</td>
<td>2.1</td>
</tr>
<tr>
<td>D4</td>
<td>129M</td>
<td>2.4M</td>
<td>VirtTwo</td>
<td>19.8</td>
<td>64.1 (1.2)</td>
</tr>
<tr>
<td>D8</td>
<td>233M</td>
<td>4.2M</td>
<td>xStore</td>
<td>38.5</td>
<td>72 (1.3)</td>
</tr>
</tbody>
</table>

\begin{itemize}
\item D1 refers to the real-world original data.
\item D2 refers to the real-world original data.
\item D3 refers to the real-world original data.
\item D4 refers to the real-world original data.
\item D5 refers to the real-world original data.
\item D6 refers to the real-world original data.
\item D7 refers to the real-world original data.
\item D8 refers to the real-world original data.
\end{itemize}

\textsuperscript{1} http://www.mongodb.org/
\textsuperscript{2} https://github.com/openlink/virtuoso-opensource
replicated ones. For example, if we have a triple (Turing, birthYear, 1912) in D1, D2 is obtained by including every triple in D1 and adding new triples such as (Turing, 1, birthYear, 1112). D4 is created by including every triple in D2 and adding new triples such as (Turing, 3, birthYear, 1854) and so on. Note that in the replicated datasets, the object value of temporal predicate is randomly filled. The other triples such as (Turing, birthPlace, London) is replicated without changing its object value. For example, in D2 we have an additional triple such as (Turing, 1, birthPlace, London).

4.2. Test queries

Table 2 lists temporal queries used in this experiment, which are borrowed from time interval queries in [31]. These query are intended to find the birth place of person who lives in a given time interval. The date value is assigned randomly during the experiment for each run. The internal sequence of scanning triples in MongoDB and Virtuoso is different. It could make us obtain different number of triples as results from two systems if the query has LIMIT constraints. For a fair comparison, we explicitly imposed a LIMIT constraint on the number of retrieved person. A example SPARQL query is as follows in the case of BEFORE query.

SELECT * WHERE {
}

SELECT * WHERE {
?s <http://dbpedia.org/ontology/deathDate> ?e .
FILTER (?e <= "2000-01-01"^^xsd:date)
} LIMIT 50

It should be noted that LIMIT is imposed only on the nested-query. The number of retrieved person can be restricted to 50 by the LIMIT constraint. We will call the results from the nested-query as the intermediate results, which will be discussed later in details.

4.3. Loading time

The loading time and the space requirements are shown in the right part of Table 1. In order to store triples into two machines disjointly, the input N-Triple file is firstly split into two files. The way of splitting is different in xStore and VirtTwo. In the case of xStore, one file would contain only temporal triples while another file would contain general triples. In the case of VirtTwo, the file is split by applying a hash function to subject URIs of each triple. The loading time is measured by summing up the time for splitting a N-Triple file and the time for loading the split files into the systems. Since the two files contain triples disjointly, the loading can be processed in parallel. It allows to load the split files concurrently into two machine. For the smaller datasets (e.g. D1 and D2), it took more time for xStore to load data than VirtTwo did. However, for the larger datasets (e.g. D4 and D8), xStore showed better performance than VirtTwo in terms of loading time. It means that xStore is scalable for larger datasets.

4.4. Space requirements

Regarding the space requirements, xStore occupies a little bit more space than the other approaches for every dataset. The size of space that is only occupied by MongoDB is shown within parenthesis in Table 1. The space requirement for VirtTwo is the sum over two Virtuoso instances where approximately the same amount of data is stored disjointly. Note that each ratio of space when storing D8 over D1 are 4.0 times (10.8 GB / 2.7GB) in xStore and 4.3 times (9.2 GB / 2.1 GB) in VirtTwo. Even if xStore requires the larger space, it does not increase dramatically as the data size increases. This makes xStore to be feasible for bigger size of datasets.

4.5. Query response time

To evaluate the query response time, 10 queries were randomly generated for 7 query types (e.g. BEFORE, DURING, EQUAL, FINISHES, MEETS, OVERLAPS, STARTS). These queries are sent to the systems sequentially. The query response time was measured from the time it sends a query to the time it receives the answers to the query. The experimental result is obtained by performing the same experiment for 5 times. The averaged query response time of 10 queries over 5 runs is listed in Fig. 4. Overall, xStore showed a better performance. For selective query types (e.g. often yield small number of results) such as EQUAL, FINISHES, MEETS and STARTS, once it finds out that there is no person who satisfies the given time constraints, it would immediately stop processing. In this case, the only storage that needs to be accessed is key-value store. This makes the selective queries more efficient in xStore. In addition, in xStore, a query is split and distributed to key-value stores and triple stores. This enables xStore to show a shorter query response time. On the other hand, in the case of VirtTwo, the same query is sent to each machine, which means that it does not take the advantages of distributed environment.

In the case of BEFORE and OVERLAPS, xStore showed not as better performance as the other types of queries. The reason is related to the intermediate results. See Fig. 5, there were more intermediate results for BEFORE, DURING, OVERLAPS query types than the other query types. Note that the maximum of intermediate results is 50 due to the LIMIT constraint specified in test queries. The nested loop join employed in xStore is not efficient when the nested query yields large size of intermediate results. Refer to the query processing flow in Fig. 3, the nested query has FILTER constraints which are converted into a key-value store query. The more subject URIs are retrieved from the key-value store, the more BGP are put into the generated SPARQL query that would then be sent to triple stores. A large number of BGP in a SPARQL query could normally make a triple store slow, which eventually makes the overall performance of xStore slow.

The total number of retrieved triples as results is depicted in Fig. 6. As can be seen, the total number of answers to the same

<table>
<thead>
<tr>
<th>Type</th>
<th>FILTER constraints in SPARQL</th>
</tr>
</thead>
</table>
Fig. 4. The query response time in milliseconds averaged over 10 runs for each method against each dataset.

Fig. 5. The number of intermediate results for xStore averaged over 5 runs against each dataset. The number of intermediate results triples is different for each run because the date value was randomly assigned.

Fig. 6. The total number of retrieved triples as results for xStore and VirtTwo averaged over 5 runs against each dataset. The total number of retrieved triples is different for each run because the date value was randomly assigned.
query is different in the compared systems, which is not big difference though. This is due to the fact that even if both systems returns top-50 persons from the nested query, persons could be different because of ranking strategies employed in MongoDB and Virtuoso. For example, if someone’s birth place is missing, the total number of retrieved triples would be reduced. As it is not easy to fully control the ranking strategies, we regard the answers as the same ones if there is no big difference in terms of the total number of retrieved triples. Let’s now discuss the total number of retrieved triples together with the query response time.

In the case of BEFORE, see again Fig. 4, xStore showed better query response time for D4 than D2 even if the size of D4 is larger than D2. This is understood by the fact that the total number of retrieved triples is smaller for D4 than D2. In the case of DURING and OVERLAPS, Virtuoso returns more triples than xStore. It leads DURING queries to be faster in xStore than Virtuoso. Note that the case of OVERLAPS is opposite. This can be explained by the number of constraints in queries. The number of inequalities is 4 for DURING query while 2 for OVERLAPS query. More inequalities does not mean a slower query response time for key-value stores compared to conventional triple stores. In other words, xStore is more applicable to complex time interval queries.

5. Conclusion

We proposed xStore, a federated SPARQL engine on a cloud environment. It is built on top of key-value stores and triple stores to realize optimized temporal query processing. Temporal triples are stored in key-value stores while the rest are maintained by triple stores. For a given SPARQL query, key-value stores are responsible for dealing with temporal information retrieval constraints. The other constraints are processed by triple stores. Experiments demonstrated that our system outperformed conventional triple store for temporal query workloads as temporal information retrieval is also important in Linked Data. We also discussed the limitation of our work related to the size of intermediate results. To further reduce the space requirements, we plan to design a compressed index schema for temporal data to save the space without losing the query processing performance. In addition, we also apply the proposed approach to spatial queries as spatial and temporal retrieval often comes together in a real-world query workload.

Acknowledgment

This work was supported by Institute for Information & Communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0101-15-0054, WiseKB: Big data based self-evolving knowledge base and reasoning platform) and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning(NRF-2014R1A1A002326). This research was also supported by Basic Science Research Program (BSRP) through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2065656).

References


Jinhyun Ahn received his BS and MS degree in computer science education from Korea University, Korea, in 2005 and 2007, respectively. He is currently a PhD student in the biomedical knowledge engineering laboratory at Seoul National University, Korea. His research interests include graph processing, bioinformatics, big data processing and semantic web technology.
**Jae-Hong Eom** is currently a principal researcher at the SK Telecom (SKT) R&D Center, Seoul, Korea. He is also co-affiliated as a research scientist at the Institute of Computer Technology, College of Engineering, Seoul National University, Seoul, Korea. He received his Ph.D. degree in computer science from the Seoul National University, Seoul, Korea, in 2009. Prior to joining SKT, he was a Research Scientist at the Max Planck Institute for Informatics (MPI-INF) in Germany from 2009 to 2012, and was a lead of Semantic Knowledge Research Group in Seoul National University as a Research Fellow from 2013 to 2015. His current research interests include text mining, machine learning, and knowledge engineering in the context of artificial intelligence and machine intelligence.

**Sejin Nam** received his Ph.D. in school of dentistry from Seoul National University, Korea. He is currently working for LIST incorporation as a research director. His research interests include semantic web, text mining, ontology based decision support system, especially in bio-medical domain.

**Nansu Zong** received his Ph.D. in computer science and engineering from Seoul National University. He is currently working as a postdoctoral fellow in Department of biomedical informatics, UC, San Diego. His main research areas are Big Data and knowledge management.

**Dong-Hyuk Im** received his BS degree from Korea University, in 2003 and his MS and Ph.D. degree in school of computer science and engineering from Seoul National University, in 2005 and 2011, respectively. He was with the Biomedical Knowledge Engineering Lab. (BIKE) from 2011 to 2012 as a research fellow. He is currently an assistant professor of the department of computer and information engineering at Hoseo University. His research interests include database system, semantic web technology and big data processing.

**Hong-gee Kim** is the director of Biomedical Knowledge Engineering Laboratory (BIKE), and a professor and dentistry library dean at School of Dentistry, Seoul National University (SNU). He is adjunct professors in Computer Science, Collaborative Medical Informatics Program, Collaborative Cognitive Science Program, Collaborative Archiving Studies Program, and Graduate School of Convergence Science and Technology at Seoul National University. He earned degrees in Philosophy, Psychology, and Computer Science. His current research interests include Semantic Web in bio-medical informatics, Ontology Engineering, and Semantic Knowledge Space.