Distributed Storage-Adaptable RDF Graph Store Over Cloud Infrastructure

Ahmed I.A. Al-Ghezi
Institute of Computer Science
University of Göttingen
ahmed-imad-aziz.al-ghezi@tu-ilmenau.de

Lena Wiese
Institute of Computer Science
University of Göttingen
wiese@cs.uni-goettingen.de

ABSTRACT

The Resource Description Framework (RDF) pioneered by the W3C is being widely used to model data of the Web from many sources. RDF with the concepts like Linked Data is creating the semantic Web. The result is linked huge data that needs to be efficiently queried, analyzed and integrated with other diverse data sources and entities in real life. Central systems may poorly perform when used to manage such web-scale data; instead, a parallelized system or distributed nodes of cloud infrastructure could speed up the performance. However, this involves partitioning of RDF graph to many machines, and efficiently index, replicate, gather statistics and materialize pre-computed results. Those different aspects would always require more storage space. The storage space is an important resource that needs to be wisely optimized over the mentioned aspects to serve the query execution performance. Moreover, the current application requirements are rapidly increasing in diverse directions which put more challenges on the distributed environment. In this paper we survey the optimization needs in distributed triple store, and present the novel design of our triple store which is distributed and adaptable to current storage space on different levels. The system would dynamically employ the available storage to the best query execution performance.

Keywords

RDF triple store, distributed RDF Processing, cloud storage optimization.

1. INTRODUCTION

The available data in the virtual digital world become enormous, and the growth in the data and the complexity of its processing has shown exploding increase over the last years. The Resource Description Framework (RDF) pioneered by the W3C is increasingly being adopted to model data from different sources, in particular data to be published or exchanged on the Web, this is mainly due to the structure of RDF. An RDF dataset consists of what is called triples. Each triple is describing a relationship between two resources; the first is called Subject (S), while the second is called Object (O). The relationship is also labeled with a value called Predicate (P). Thus, a triple has the format: Subject, Predicate, and Object. What moves web triple data into another dimension is the concept of the Linked Data [24]. Different data sources could be linked together using unique URI and this enables the ambitious idea of querying and talking to the web. However, the heterogeneity of the data makes writing a query in the RDF formal languages like SPARQL a time consuming task in its own, as the user might often need to look into the raw data while writing his query.

The support for systems that enable the user to talk to the web with natural language queries would highly reduce the time used to build the query and would increase the number of users who are able to write these queries and would as a result increase the number of queries the triple store can receive [25].

Managing RDF data on a Web scale in a central system may not perform well; instead recent research works moved to make use of the parallelization provided by cloud based systems. However, storing RDF-Graph on a distributed system store where a query processing engine can efficiently query data is a challenging task due to the heterogeneity, big size, and complex relationships found in the data, beside the growing application needs for complex analytical queries and to provide the answers on time. The current researches on parallel RDF data management involve enormous open problems [4]. In such a distributed system the RDF graph needs to be partitioned to the working nodes. Moreover, in order to serve queries more efficiently, some parts of the RDF graph need to be replicated to more than one node. Deciding how the data is partitioned and replicated would have its direct influence on the performance of the query processing. To illustrate this, consider for example a query that is to be processed by the distributed triple store, if each working node receives the query and runs it against its local part of data; a poor performance would be when only one node has the availability of required data locally, and has to do the whole work load alone. A better performance happens when the partitioning and replication are well optimized to enable a parallel processing of the data on an increased number of working nodes.

However, the replication requires more storage space on each machine and the space itself might be considered scarce resource with respect to each node, because the triple store needs to build indexes for better performance of query processing, which

29th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 30.05.2017 - 02.06.2017, Blankenburg/Harz, Germany.
Copyright is held by the author/owner(s).

requires more storage space. Moreover, the space is also needed to materialized-queries results and statistics. The complexity of taking a decision about data partitioning can be overcome by performing the optimization decision based on some application trends. As we mentioned earlier the support of natural query processing would provide rich and strong application trends. In this context we present our triple store and its proposed structure. The storage layer is designed to contain an optimizer that would instruct the working nodes on the optimized way to employ any available storage space for better performance.

The rest of this paper is structured as follows: In section 2 we present briefly the work related to distributed RDF triple stores. In section 3 we present our system architecture and show our optimization directions on three aspects; the RDF graph partitioning, the RDF indexes, and the space optimizer.

2. RELATED WORK

The distributed storage and processing of RDF data has taken a lot of attentions in the latest research. Some direction of work maintained the RDF graph in a distributed file system like HDFS [19], those systems are designed for reliable and scalable storage, but the drawback is that DFS does not natively provide fine grained access to the data in the stored files. Some works performed indexing mechanism to enhance the access over HDFS like [20, 7]. Another direction is based on key-value stores. Such systems depend on indexing the triples by keys. The key can be the Subject S, Object O or Predicate P; or any combination of them. The most basic indexes are SPO, POS, and OSP which are used in systems like Rya [14] and AMADA [21]. RDF-3X [8] is a central RDF store; it focuses on providing high performance by relying on building many indexes. It pre build all the six possible indexes of S, P and O of the triple data. Other types of systems using key value stores include H2RDF [9] and MAPSIN [22] built on top of HBase [19]. Other systems like Trinity.RDF [3] achieved high performance by processing the query as graph exploration algorithm. On the other hand, some systems maintain cluster of centralized triple-store nodes, coordinated by single master node. The Master node partitions the graph using partitioning algorithms that aim to minimize the network communications during query execution while increasing the parallelization among slave nodes. The work of [10] follows this approach and uses METIS [11] which partitions the graph trying to achieve min-cut. Each triple is assigned to the machine where its subject belongs (1-hop guarantee) or where its subject and object belong (2-hop guarantee). More guarantee can be achieved by replication of those triples that are located on the partitions boundary (n-hop guarantee). However, such approaches do not scale well when the query execution targets certain regional part of the graph; this happens when there are non-variable inputs in the query. Such situation affects the parallelization of the query. WARP [15] extends the approach of [10] to consider query work load as a factor to determine the triples that need to be replicated. Partout [12] also focuses on query load to achieve balanced partitioning by counting the number of queries that reference each fragment and the triples that match the fragment. The works [15, 12, 10] made use of the high performance provided by the indexes strategy of RDF-3X [8] which is used as the underlying central store on each node. However, they didn't provide a mechanism or a strategy to work on when there is not enough storage available for such exhaustive indexes. TriAD [26] partitions the graph also based on METIS and presents an interesting approach to provide RDF graph summary; the author's results of this summary showed enhancements in the query execution time. However, this graph summary requires more storage space and this could add another storage optimization variable to our storage optimizer shown latter in section 3.3.

3. SYSTEM ARCHITECTCTURE

Our distributed triple store is composed of a cluster of working nodes, where each node is built on top of RDF-3X [8]. One of the nodes is set to be the master node. Its role is to perform the initial RDF data partitioning to the working nodes. Each node has its own local indexes and query processing engine. The storage system has initially all the 6 possible permutation of indexes for S, P and O. The Master node contains a dictionary which compresses the textual data representing URIs into integer codes which consume much less storage space. The partitioning, replication and shipment of data between nodes is done in the integer format. The master has also a global storage optimizer, and each node also has its own local storage optimizer.

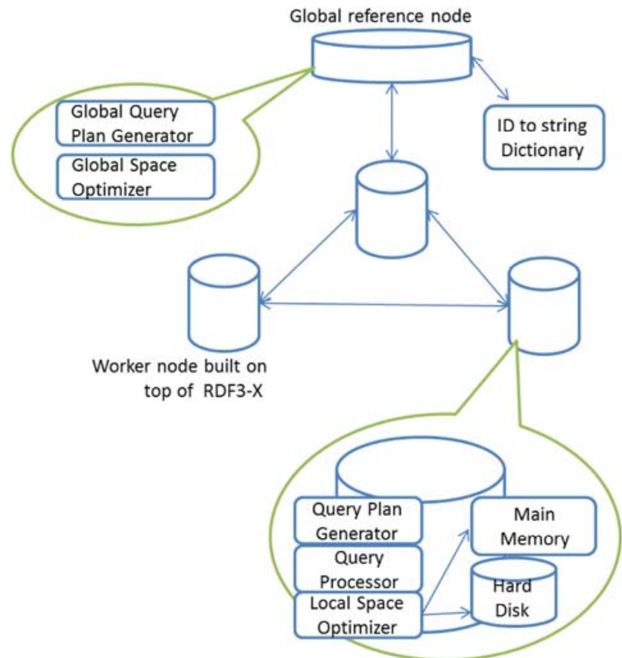

Figure 1. System Architecture.

3.1 RDF Graph Partitioning and NL Queries

The RDF data is to be partitioned at the Master node. The partitioning objective is to have maximum parallelization later when executing a query in the distributed store. Due to the nature of the linked RDF data, it is usually very complex to have a partitioning approach that directly achieves absolute maximum parallel query performance. So far, we have adopted random partitioning using hashing similar to the key value stores mentioned earlier in related work. Some of the optimization over the random partitioning is to use the min-cut mentioned earlier in [10], or to adjust the partitioning results depending on the query

workload [15]. To overcome the optimization process we propose to partition the RDF graph according to a given direction of application trends. The pre optimization in the partitioning algorithm serving these query trends would greatly affect the performance in the parallel sense. One of the very important application trends is the request of Natural Language (NL) Queries interface that we would deeply investigate to make the partitioning and replication optimized towards better performance of such queries. What makes the optimization on these NL queries more beneficial is that they have more common trends than the normal SPARQL queries. When the natural query translation receives an NL query, it would usually try to understand the question queried by the user so that it can generate the suitable SPARQL query and return the result back to the user. The translation process itself requires generating several SPARQL sub-queries that would run against the triple store. On the contrast with the final SPARQL queries that is required by the user, the sub-queries generated by the NL translator have similar trends and this would obviously enable the pre-optimization process to serve the translating queries much better than supporting the optimization on the final SPARQL query. Consider for example the NL query "*What is the country that contains the highest mountain in the world?*" It could be generally transferred to "*What is the (x-->instance of: country) that contains the highest (y-->instance of: mountain) in the (z--> instance of: world)*". In this type of translation, the detected variables in the NL query can be always associated with the predicate *"instance of*". Thus we would have a lot of queries generated on the same trend overcoming the problem of heterogeneity and diversity which are found in the normal queries, and this allows performing pre-optimization when partitioning and replicating the data.

3.2 Indexes Optimization

We use in each node a central RDF store which is built upon RDF-3X [8]. This underlying store uses all the six possible permutations of the S, O and P for faster data access; besides using aggregate indexes which returns the counts of existence rather than the values of S, P, and O. This excessive indexing creation has shown very good performance, but it requires more storage space. On the other hand, graph-based triple stores like Trinity.RDF [3] have also shown notable good performance due to its indexing and query processing methods which are graph-based approaches. In our system, we would have the ability to adaptively create more indexes when there is more assigned storage space from the space optimizer. The extra indexing would look to the RDF data conceptually as a graph, and build extra indexes in important and hot regions of the graph per working node; the important regions of the graph are the parts that are highly queried with respect to other parts. In an RDF graph: S and O are modeled as vertices, and P is modeled as edges. Then for some important vertices v (S or O), we would have extra SP(iO) or OP(iS) indexes, where *i* is some *classified-entity ID*, that is a common entity-id between the neighbors of v. The usual index SPO would return all of the O which match for given S and P; each of the returned O might be investigated again in the index, and this could be a huge list if O is a big vertex in term of its connected edges. The index SP(iO) would accept further parameter *i* about the required O, and would efficiently returns only the triples that have same S,P and i in one step.

We would explain the concept with an example query shown in Figure 2. "*Find all the museums located in France and exhibit paintings of an Italian artist* ". The SPARQL representation of the query is shown in Fig2.B, while the corresponding query graph is shown in Fig2.A. Any sub graph from the distributed RDF graph that matches the given query graph in Fig1.A should be part of the answer. The query has five triple patterns, and the order of their execution obviously affects the query execution time. If for example, the query pattern (call it p_1) is *?z :located_in :france*, was the first to execute, then its result set (call it R) is all the entities that located in France which is typically a big set. The second triple pattern to be executed say p_2: *?z exhibits ?y* would be run against R instead of the whole RDF graph. The partial result is so far all the entities which are located in France and have the predicate "exhibits".

The entities which are connected to ":france" by the predicate ":located ", can be classified and grouped according to a common properties. If the data is considered holistic and on a web-scale , there should be more than 1 k museums in France, and hundreds times this number of entities that are located in France. If we are able to extract all the museums in one step, then we would speed-up the local processing of the triple pattern p_1 hundreds of times. Each of the neighbors of vertex France is marked with its most frequent predicate which is in this example used as the *classified-entity ID* of vertex "France". Now, when we process p1, we would use the index OP(iS) and set O to ":france", P to located and i to exhibits then retrieve all the S that match. In this example, we used the most frequent predicate that lies one step ahead form the currently executing triple pattern. However, a more complicated combination of nearby edges and vertices could be used.

To some extent, these extra indexes may be considered some type of materializing join results, and it requires more space that could be assigned by the storage space optimizer to increase the performance.

Adding more indexes within important nodes is away for increasing conceptual local knowledge within the RDF graph; which would consume more space and require more processing but it would pay off with a better execution time. Moreover, the analyzing process and indexes building are very scalable and suitable to be done in parallel on clustered machines, as each machine can work on its local part of the graph.

The selection of eligible vertices to build the local indexes is done by an iterative algorithm. At working node *i*, after the execution of any query-pattern on single or set of vertices, the index optimizer records a stat about each queried vertex "v" and its neighbors, then calculate or adjust the value of the performance gain g(v), which is a referential metric of the expected performance increase gained of building the local index in this vertex. When the gain of a vertex g(v) reaches a certain threshold, the vertex v is pushed into eligible-vertices-queue which is basically a priority queue on the values of g(v). Assume that the local space optimizer in working node *i* is assigned *k* space to build extra local indexes; the index optimizer will pop a vertex from the priority queue and start building the indexes for the popped vertex. Then it recursively pops another vertex until the queue is empty or there is no more storage space left for further employment

Any vertex which has been assigned an index is pushed into another priority queue called indexed vertices queue, which

organizes its vertices priority on the reverse value of g(v). The continues execution of queries triple pattern on a vertex v would adjust its v(g) value. In the case of no more storage space available, and when the value of $g(v_e)$, (where v_e is the vertex on the top of the eligible-vertices-queue) becomes larger than the value of $g(v_i)+t$, (where v_i is the vertex on the top of the indexed vertices queue, and t is stability margin), the local index in v_i is eligible for deletion and the space gained is employed for the benefit of v_e.

The value of t would ensure that the deletion only happens when there is obvious performance gain from building the new index in the other selected vertex.

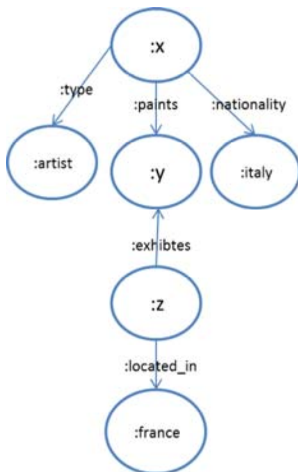

Fig.2A

```
select ?y ?z
where{
    ?x :type :artist.
    ?x  :paints  ?y.
    ?x  :naltionality :italy.
    ?z :exhibits  ?y.
    ?z :located_in :france.
    }
```

Fig.2B

Figure 2. Query Example.

3.3 Space Optimizer

The storage space is a very precious resource in a triple store especially in a distributed environment. The space is required to build indexes and replication. And if there is enough space it is always a good idea to employ it for performance. For example, replicating all of the data so that each node has all the data available locally. However, the available storage space is a dynamically changing variable, since within some point in time, more data could be added to the store, more storage equipped in the system, or when needs more space to build indexes, the decision about what to do with the available space needs to be taken wisely.

In our triple store, we would have local space optimizer in each working node as well as a global optimizer in the Master node. The role of the optimizer is to provide decision about the assignment of space to each task that requires space. Those are in our case indexes, materialized results, statistics and replications. This decision would be taken dynamically and adaptively. That means that if the storage availability variables change, the system would adapt itself to the new situation. This would usually happen when new data is added to the store, or when new storage space is assigned; we would then have that at any moment in time, all the available storage space is employed and used for better performance.

4. ACKNOWLEDGMENTS

The author would like to thank the German Academic Exchange Service (DAAD) for providing funds for research on this project.

5. REFERENCES

[1] Gallego, M. A., Fernández, J. D., Martínez-Prieto, M. A., and de la Fuente, P. 2011. *An empirical study of real-world SPARQL queries*, 1st International Workshop on Usage Analysis and the Web of Data (USEWOD2011) at the 20th International World Wide Web Conference (WWW 2011), Hydebarabad, India.

[2] Kaoudi, Z and Manolescu, I. 2013. *Triples in the clouds*, ICDE - 29th International Conference on Data Engineering, pages 1258–1261.

[3] Shao, B., Wang, H., and Li, Y. 2012. *The Trinity graph engine*. Technical Report 161291, Microsoft Research, Tavel, P. 2007.

[4] Kaoudi, Zoi, and Manolescu, I. 2015. *RDF in the Clouds: A Survey*. The VLDB Journal—The International Journal on Very Large Data Bases 24.1, 67-91.

[5] Zeng, K., Yang, J., Wang, H., Shao, B. and Wang, Z. 2013. *A Distributed Graph Engine for Web Scale RDF Data*. In PVLDB.

[6] Yang, S., Yan, X., Zong, B., and A. Khan. 2012. *Towards effective partition management for large graphs*. In SIGMOD Conference, pages 517-528.

[7] Dittrich, J., Quiane-Ruiz, J.A., Richter, S., Schuh, S., Jindal, A., and Schad, J. 2012. *Only aggressive elephants are fast elephants*. In PVLDB, pages 1591–1602.

[8] Neumann, T., and Weikum, G. 2010. *The RDF-3X Engine for Scalable Management of RDF Data*. LDBJ, 19(1).

[9] Papailiou, N., Konstantinou, I., Tsoumakos, D., and Koziris, N. 2012. *H2RDF: adaptive query processing on RDF data in the cloud (demo)*. In WWW.

[10] Huang, J., Abadi, D. J., and Ren, K. *Scalable SPARQL*. 2011 *Querying of Large RDF Graphs*. PVLDB, 4(11):1123–1134.

[11] METIS. http://glaros.dtc.umn.edu/gkhome/views/metis.

[12] Galarraga, L., Hose, K., and Schenkel, R. 2012. *Partout: A Distributed Engine for Efficient RDF Processing*. Technical Report: CoRR abs/1212.5636.

[13] Witte, D.D., De Vocht, L., Verborgh, R., Knecht, K., Pattyn, F., Constandt, H., Mannens, E., and Van de Walle, R. 2016. *Big linked data ETL benchmark on cloud commodity hardware*. In Proceedings of the International Workshop on Semantic Big Data (SBD '16). ACM, New York, NY, USA, Article 12 , 6 pages. DOI: http://dx.doi.org/10.1145/2928294.2928304.

[14] Punnoose, R., Crainiceanu, A. and Rapp, D. *Rya: A Scalable RDF Triple Store for the Clouds*. 2012. In Workshop on Cloud Intelligence (in conjunction with VLDB).

[15] Hose, K. and Schenkel, R. 2013. *WARP: Workload-Aware Replication and Partitioning for RDF*. In DESWEB Workshop (in conjunction with ICDE).

[16] Harris, S. and Seaborne, A. 2013. *SPARQL 1.1 Query Language. W3C Recommendation*, http://www.w3.org/TR/sparql11-overview/.

[17] Colazzo, D., Goasdou´e, F., Manolescu, I., and A. Roatis. 2014. *RDF Analytics: Lenses over Semantic Graphs*. In WWW.

[18] Damljanovic, D., Agatonovic, M., Cunningham, H. 2011. FREyA*: an interactive way of querying linked Data using natural language*. In: García-Castro, R., Fensel, D., Antoniou, G. (eds.) ESWC. LNCS, vol. 7117, pp. 125–138. Springer, Heidelberg (2012). doi:10.1007/978-3-642-25953-1_11.

[19] Apache Hadoop. 2012. http://hadoop.apache.org/.

[20] Dittrich, J., Quiane-Ruiz, J.-A., Jindal, A., Kargin, Y., Setty, V., and Schad, J. 2010 .*Hadoop++: making a yellow elephant run like a cheetah (without it even noticing)*. In PVLDB, pages 518–529.

[21] Aranda-Andujar, A., Bugiotti, F., Camacho-Rodriguez, J., Colazzo, D., Goasdou´e, F., Kaoudi, Z., and Manolescu, I. 2012. Amada: *Web Data Repositories in the Amazon Cloud (demo)*. In CIKM.

[22] Schatzle, A., Przyjaciel-Zablocki, M.n., Dorner, C., Hornung, T., and Lausen, G. 2012 .*Cascading Map-Side Joins over HBase for Scalable Join Processing*. In SSWS+HPCSW.

[23] Damljanovic D., Agatonovic M., Cunningham H. 2012. FREyA: *An Interactive Way of Querying Linked Data Using Natural Language*. In: García-Castro R., Fensel D., Antoniou G. (eds) The Semantic Web: ESWC 2011 Workshops. ESWC 2011. Lecture Notes in Computer Science, vol 7117. Springer, Berlin, Heidelberg .

[24] Bizer, C., Heath, T.; Berners-Lee, T. 2009. *Linked Data The Story So Far*. International Journal on Semantic Web and Information Systems. 5 (3): 1–22. doi:10.4018/jswis.2009081901. ISSN 1552-6283.

[25] Kaufmann, E., Bernstein, A. 2007. *How useful are natural language interfaces to the semantic web for casual end-users?* In: Franconi, E., Kifer, M., May, W. (eds.) ESWC. LNCS, vol. 4519. Springer, Heidelberg (2007).

[26] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, Martin Theobald. 2014. *TriAD: a distributed shared-nothing RDF engine based on asynchronous message passing*. SIGMOD Conference: 289-300.