Towards Load Balancing and Parallelizing of RDF Query Processing in P2P Based Distributed RDF Data Stores

Liaquat Ali, Thomas Janson, and Christian Schindelhauer
University of Freiburg
Email: {ali, janson, schindel}@informatik.uni-freiburg.de

***Abstract*—For evaluating RDF queries in Peer-to-Peer (P2P) based RDF data stores, the location of a RDF triple in the network must be attainable from a triple pattern in the given query. An existing strategy, used by state-of-the-art distributed RDF data stores, to fulfill this requirement is to store triples at three locations that each triple can be found by the subject, predicate, and object identifier. A major drawback of this strategy is the issue of load-balancing caused by the fact that the frequency of subject, predicate, and object occurrences in triples is not uniformly distributed. While the majority of URIs and literals occur very rarely some occur very frequently (e.g., peer responsible for 'rdf:type' is subjected to a very high storage load). In addition, this skewed RDF triples distribution among network peers also leads to an unfair query processing load distribution and long query processing time. To cope with hotspots caused by unfair data load distribution, we propose an optimized routing index scheme where triples are indexed on the combination of their subject, predicate and object components. This paper will also show how can we exploit this novel index scheme to achieve a better distribution of query processing load and faster query response time by bundling computation resources and bandwidth of peers with parallelism.**

I. Introduction

The increasing amount of RDF [1] data on the Web calls for the development of RDF data stores customized for the efficient management of such data. Thus, several projects have emerged that proposed distributed solutions for the storage and querying of RDF data (RDF triples). State-of-the-art distributed RDF data stores such as RDFPeers [2], Atlas [3], BabelPeers [4], GridVine [5] and 3rdf [6] use P2P overlay networks to store and query RDF data in a distributed manner. To attain an efficient search for RDF triples with the same subject, predicate, or object the triples are stored three times in the network for each of the triple components separately in these distributed RDF data stores. The triples are hence grouped with the same identifier on the same peer, with the advantage that it offers a constraint search for a specific subject, predicate or object in a local database. However, this comes with the drawback that we leverage the load balancing techniques of most overlays, because the triples may not be stored on peers of underlying networks uniformly due to the non-uniform frequency distribution of subject, predicate, and object occurrences in these triples. Some URIs and literals occur very often (e.g., peer responsible for 'rdf:type' is overwhelmed with RDF triples) while others occur only rarely.

In addition, this indexing (storage) technique also leads to an unfair query load distribution and cause substantial cost in local computation and data transmission time. This increase in computation and transmission time is caused due to the fact that a large portion of RDF triples are stored on relatively small portion of the network peers. We can expect that heavily loaded peers will be frequently accessed during query evaluation, and the storage of large number of triples takes these peers a long computation time to compute the results and a substantial time to transmit the resulting triples. The constraint to use only a single constant in triple patterns for routing the query to responsible peer also leads the query evaluation on relatively small number of peers, thus results to an unfair query processing load distribution.

To cope with the aforementioned drawbacks of existing indexing technique, we propose a novel routing index scheme in this paper, where a triple is indexed on the combination of its subject, predicate and object identifiers, i.e., subject+predicate, predicate+object and object+subject. This indexing technique bears the same storage cost as in the state-of-the-art (triples are stored 3 times), but results to a better data load distribution. We will also show in this paper, by using this new indexing technique we can achieve a better distribution of query processing load and faster query response time by bundling computation resources and bandwidth of peers with parallelism.

II. Related Work

Since RDF query languages mainly support constraint search of the triples subject, predicate or object components, existing distributed RDF data stores (RDFPeers, Atlas, BabelPeers, GridVine and 3rdf) index triples 3 times for each of these components. This indexing technique provides the possibility to find triples based on any search criteria as long there exist at least one constant in a triple pattern. However in practice, frequency of subject, predicate and object occurrences in triples is not uniformly distributed, and the peer responsible for the frequent one is heavily loaded.

Cai et al.[2] addressed the issue of load-balancing by limiting the storage of overly popular URIs and literals based on the local capacity of peers. This comes of course at the cost of possibly losing the complete result, when the query is on these popular values. Battre et al.[4] tackled this problem by constructing an overlay tree over DHT position of only overly

popular triple components. This type of load balancing is also very fragile in the case of a node failure in the overlay tree, which results to the loss of whole branch of the tree.

Meitz et al.[7] showed a huge difference among the peer's data load, when triples are indexed 3 times using a fixed hash depth of 1, i.e., hash(subject), hash(predicate), hash(object). To improve the data distribution they proposed the idea to index triples using a random hash depth, for example, for a hash depth of h=4, there would be 4 potential location keys for each triple components. They showed that the higher the value of hash depth is chosen, the better the triples distribution among peers would be. However, this comes at the cost of network communication during query evaluation, since many peers have to be queried for the evaluation of a triple pattern. Finding an optimum hash depth is also missing in their work.

To cope with hotspots of unfair load balancing, we proposed in [8] to index a triple for each possible combination of its 2 components, subject+predicate, subject+object, predicate+subject, predicate+object, object+subject and object+predicate. Extra storage of triples in the network was the price we paid to achieve a fair triple load distribution.

For the improvement in query load distribution, authors in [3] additionally indexed triples by combinations of triple components 'subject+predicate', 'subject+object', 'predicate+object', and 'subject+predicate+object', with 7 replications of each triple in total. They used this extra storage overhead for the distribution of the query processing load among many peers, but have not studied the utilization of this overhead for the improvement in response time of queries.

Harth et al.[9] used the notion of quad (subject, predicate, object, context) to represent the RDF data, and proposed an optimized index structure to support evaluation of RDF queries in centralized RDF data stores. He showed that only 6 indexes are needed to cover all possible (16) access patterns, where as an access pattern is a quad where any combination of subject, predicate, object, context is either specified or a variable. Motivated by their idea of reducing the number of required indexes, we show in Section IV-B that only 3 indexes is needed in our new index scheme to cover all 8 possible triple patterns.

III. SYSTEM MODEL AND DATA MODEL

We simulate a distributed RDF system using the search-tree based overlay 3nuts [10] and compare the performance when using the state-of-the-art indexing for subject, predicate and object, and the new indexing for the triple distribution.

Each peer in the distributed RDF system can publish RDF resources in the network. In RDF [1], resources are expressed as subject-predicate-object expressions, called triples. The subject in a RDF triple denotes the resource, and the predicate expresses a relationship between the subject and the object.

Each peer in the distributed RDF system can also pose SPARQL [11] basic graph pattern queries to retrieve the RDF triples stored in the network. These basic graph pattern queries are composed of triple patterns.

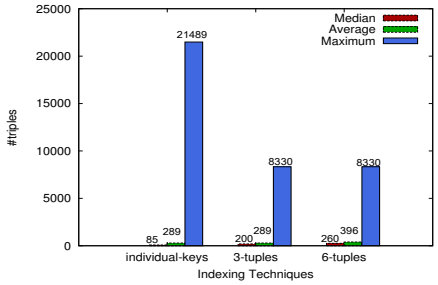


Fig. 1: individual/combined indexes comparison.

IV. DATA DISTRIBUTION

RDF triples in distributed RDF data stores are indexed in such a way that it can answer all possible triple patterns. A triple pattern is a triple where any combination of subject, predicate and object is either specified or a variable.

To find all triples for a specified triple component, the triples are stored in a P2P-network with the triple component as key. Since all three components of a triple can be specified, this makes three storages for keys with subject, predicate, and object identifier. The search structures of P2P-network are originally designed to store single data elements at a unique key and it is not provided to balance several data elements with the same key on several peers for better load distribution. In distributed RDF data stores, a peer stores a set of triples with same triple component as key. Since in RDF triples some triple components are more frequent, for instance 'rdf:type', the peer responsible for such a key store more triples and the built-in load balancing is not able to balance this higher load. State-of-the-art distributed RDF data stores (see Section IV-A) do not tackle this load balancing problem. In Section IV-B we present our own solution, with the basic idea to extend the keys such that the set of triples with same key are smaller and load balancing of the P2P-network performs better.

A. State-of-the-art index scheme

The indexing of triples on their individual components in existing distributed RDF data stores results to a skewed triple distribution because of non-uniform frequency distribution of subject, predicate, and object occurrences in triples.

The effect of this unfair distribution is illustrated in Figure 1. For measurement the data of Lehigh University Benchmark (LUBM [12]), for one university with 100,000 triples, has been stored in the network of 1000 peers. The bars for individual-keys show the statistics of data load (number of triples) per peer using this index scheme. We observe a huge differences among the maximum, average and median load. The number of triples on heavily loaded peer (21489 triples) is about 73 times higher than the peers average load (289 triples).

B. Novel index scheme

When a set of data elements is mapped to the same key in the network, we can only achieve fair load distribution if either the network provides mapping several peers on the same key and distribute the load of the key or we make the keys on

application layer unique such that the load-balancing of P2P-networks only supporting one peer for one key fully applies. In our solution, we decide for the second option and reduce the set of triples with same key by extending the keys which are originally the subject, predicate, and object identifier. First idea coming to mind is adding an arbitrary key extension, for instance a small hash value, and keys would be divided into subsets resulting into better load distribution. However, these subsets would be unstructured and if we evaluate a query containing the original key, we have to process it at all extended keys, for instance subject+hash. So we have decided to extend the keys with another triple component, e.g,. the key consists out of subject+predicate. A possible drawback might be a non-uniform fragmentation of the triples sets but the big advantage is that the triple sets are structured and if subject and predicate are already specified in a triple pattern, it has to be performed only at the location of subject+predicate key.

We propose an index scheme '3-tuple index' where 3 combined routing indexes are created on triples subject, predicate and object components, i.e., subject+predicate, predicate+object and object+subject. The 3-tuple or 6-tuple indexes are based on the notion of tuple index.

Definition 1 (Tuple index): A tuple index concatenates the identifiers of two components as key for the storage of a triple.

To tackle the problem of load-balancing, we proposed a similar storage solution '6-tuple index' in [8], where triple are indexed six instead of three times for each possible combination of its 2 out of 3 components.

We avoid the storage of extra triples in our new index scheme by not considering the usage of redundant routing indexes, which we had in our 6-tuple indexing. For example, the index on subject+predicate suffices for routing a query to the responsible peer when subject and predicate are given in a triple pattern, and we do not need to store triples on predicate+subject. This index also covers the evaluation of triple patterns where only subject is provided (through prefix search). For the search where only predicate is given we can use the index on predicate+object.

The creation of routing indexes on combinations of subject, predicate and object components divides the data load of a heavily load peer to many peers. For example, Figure 2 shows that in our experiment, with indexing triples on its predicate, for instance, there was only one peer responsible for the storage of 18128 triples with predicate 'type'. However, indexing triples on predicate+object subdivided this storage load to many peers storing triples of the classes (UndergradStudent 5916 triples, GraduateStudent 1874 triples, Publication 5999 triples and so on) respectively.

RDF queries where two triple components are unknown get more challenging in this new indexing technique, if not all data for a specific triple component are on the same peer and have to be collected from many peers. To evaluate such queries, we leverage the fact that the search tree based overlays (e.g. 3nuts) provide support for range or prefix queries. For example, a lookup for triple pattern (? : predicate : ?) resolves to a prefix query for a specific predicate on the predicate+object index,

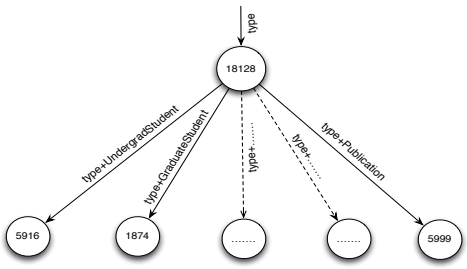


Fig. 2: peers load distribution.

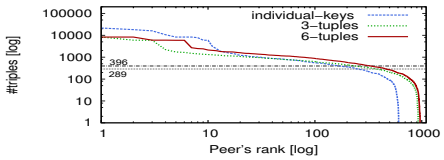


Fig. 3: #triples/peer for individual/combined indexes.

where we go to an arbitrary peer in the predicate's path in the search-tree and scan the subtree for all predicate-object combinations only using direct routing links. In contrast, range queries are not practical in DHT-based overlays. Therefore, we see a trade-off in DHT-based systems with two options, either a more balanced data distribution with combined indexes but limited functionality (evaluation of triple patterns with only 1 constant is not possible) or all functions but more unbalanced data load with state-of-the-art index scheme.

The effect of using new index scheme is reflected in Figure 1, where the bars of 3-tuples index show a significant reduction in the differences among the maximum, average and median load. The number of triples stored on the heavily loaded peer reduces to 8330 triples. Though 6-tuple indexing, 6-tuples bars, stores more triples than 3-tuple indexing, but this extra storage has no impact on further reducing the hotspots.

Figure 3 shows the triples managed by the peers in decreasing order to the number of triples per peer, When we use combined keys for indexing and compare the top ranked peers, we can in fact prevent hotspots where peers are overloaded by data. Certainly, in an optimal case, all peers would manage the same amount of triples indicated by the constant function of the average value. Both in individual key indexing and 3-tuple indexing the average number of triples managed by a peer is 289, but in 6-tuple indexing the average number of triple is 396. The median peer in 6-tuple index scheme has 260 triples, in 3-tuple indexing 200 and in individual key indexing only 85, indicating a better load distribution for our novel 3-tuples index scheme, without bearing an extra triple storage cost.

V. Query Processing

In addition to having an unbalanced data distribution by storing triples on its individual subject, predicate and object components, this triple distribution technique also leads to

a very unfair query processing load distribution and cause a substantial cost in terms of local computation and data transmission time. This increase in computation and data transmission time is caused due to the fact that in this triple distribution method a large portion of the RDF data reside on relatively small portion of the network peers. We can expect that heavily loaded peers will be frequently accessed during query evaluation, and the storage of large number of relevant triples takes these peers a long local computation time to compute the result and subsequently a substantial cost in transfer time to transmit the resulting triples. The use of only 1 constant in triple patterns for routing also directs the evaluation of query on relatively small number of network peers and thus results to a an unfair query processing load distribution.

For example, The evaluation of given query in Listing 1 is carried out by sending the query to the peer responsible for the predicate 'rdf:type'. The presence of large number of triples with predicate 'rdf:type' on the corresponding peer cause a long time to respond these triples.

We can exploit our 3-tuple index scheme to improve the response time and processing load of queries. Triples indexed on combination of subjects, predicates and objects are distributed relatively on a larger portion of network peers, consequently evaluation of RDF queries are carried out on large part of the network peers (fair query load distribution). Peers responsible for the combination of subjects, predicates and objects are also supposed to contain relatively small number of triples. The corresponding peers thus have to spent less time for the computation and transmission of these triples.

For example, with this index scheme, the evaluation of given query in Listing 1 is carried out on the peer responsible for the key 'rdf:type+ub:GraduateStudent'. The corresponding peer definitely contain less triples than the one responsible for the key 'rdf:type', and thus results to a shorter query time.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/zhp2/univ-bench.owl#>
SELECT ?X
WHERE { ?X rdf:type ub:GraduateStudent }
```

Listing 1: SPARQL query returning under graduate and graduate students.

The existing distributed RDF data stores, despite their differences on query processing, evaluate RDF queries sequentially. Majority of these data stores (RDFPeers, Atlas, GridVine, 3rdf) use Query Chain (QC) query processing algorithm [3], which moves query processing in sequence from one peer to another and intersect the candidate sets in this way.

We exploit our novel indexing technique to parallelize the processing of RDF queries. Parallelism could be a real boost for computation time and data transfer time, since we can bundle the computation resources and bandwidth of several peers in parallel. For the parallel processing of RDF queries we adopt the query processing algorithm, Spread By Value (SBV) [3], it extends the ideas of QC by exploiting the values of matching triples found during processing triple patterns

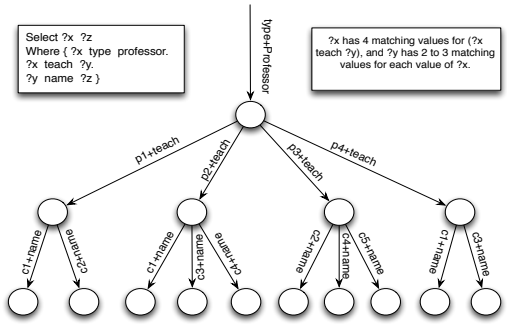


Fig. 4: query chain in parallel query processing.

incrementally, it rewrites the next triple pattern and distributes the responsibility of evaluating it to more peers than QC.

Figure 4 shows Parallel evaluation of an example query. As in QC the first triple pattern is evaluated by the peer responsible for the key 'type+Professor'. From this point on, the query plan produced by SBV is created dynamically by using values of matching triples that peers find at each step. The values of variable *?x* (p1,p2,p3,p4) are used in the second triple pattern to produce a new set of queries that will jointly find answers to this triple pattern. In the next step newly found values of variable *?y* (c1,c2,c3,c4,c5) are used in the third triple pattern and send the resulting query set to responsible peers. Multiple chains of peers will be involved for query evaluation in this way, and the peers at the leaf of these chains will deliver partial results back to the originating node.

VI. Performance Analysis

We present experimental performance evaluation done with a simulator for our RDF system using either state-of-the-art indexing or novel index scheme, and 3nuts [10] as overlay network. For the testing we use the LUBM [12] data-set of one university. There were 102707 triples in total, and the network contained up to 1000 peers. We have generated 5 query sets (based on LUBM queries), the query in Listing 2 represents one type of such queries.

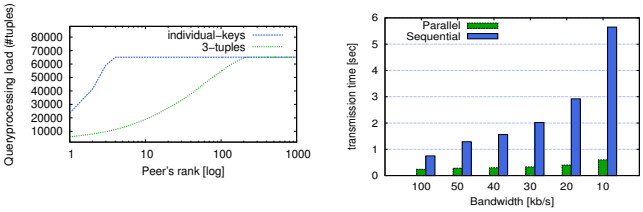
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/zhp2/univ-bench.owl#>
SELECT ?std_name ?course_name
WHERE {
?X rdf:type ub:UndergraduateStudent.
?X ub:name ?std_name.
?X ub:takesCourse ?Y.
?Y rdf:type ub:Course.
?Y ub:name ?course_name }
```

Listing 2: SPARQL query returning names of the students and the courses they have taken.

A. Analysis of Query Load distribution

We analyze the effect of using 3-tuple index scheme on the query load distribution. The query processing load for a peer p is defined as the number of tuples (intermediate results) that arrive at p, for the evaluation of a triple pattern, and are compared against its locally stored triples.

Figure 5a shows the query load distribution, for the query in Listing 2, for both existing and new index schemes. On the *x*-axis peers are ranked starting from the most highly loaded peer. Where as *y*-axis represents the cumulative load. Each point (*x,y*) in the graph represents the sum of load *y* for the *x* most loaded peers. Though for both index schemes the same total query processing load is created in the network, with 3-tuples indexing the query load is distributed to significantly higher number of peers than with the individual-keys indexing. With individual-keys indexing only 5 peers (out of 1000) are involved, where as with 3-tuple indexing 230 peers participated in the query processing. We achieve this nice query load distribution with 3-tuples indexing because it allows to forward the rewritten queries to peers in parallel using the combination of constants in triple patterns as location keys.



(a) Cumulative query processing load (b) Comparison of Sequential and Parallel query processing

Fig. 5: Measurement results for the performance boost with parallel query processing.

B. Analysis of Query Response Time

In our analysis we use connections between peers with homogeneous delay and bandwidth. Therefore we will present in this section how to derive the query response time from the given experimental size of the transmitted data and local computation time with a given physical network model.

Let δ denotes the delay and b the bandwidth in the physical network. In parallel execution the query is split into several sub-queries where each sub-query q_j defines a path in a tree (compare Figure 4). The query response time t for one path q_j is then for m triples patterns p_i

$$t\,(\text{path } q_j) \quad = \quad \sum_{i=1}^{m} \left(c_i + \text{hops}_{i,j} \cdot \delta\right) + \underbrace{\sum_{i=1}^{m} \left(\frac{\text{data}_{i,j}}{b} + \delta\right)}_{\text{transmission time}}$$

where c_i is the computation time for evaluating pattern p_i, hops_i is the number of hops to lookup the peer providing the triples for pattern p_i, and data_i is the data size for triples to transfer to the next peer for the evaluation of of pattern p_i. Since the individual sub-queries are independent, e.g. all lookups are performed in parallel, the total query time of the tree of sub queries is

$$t\,(\text{tree}) \quad = \quad \max_{\text{path } q_j} t\,(\text{path } q_i) \ .$$

Comparing sequential and parallel processing, the data size of the i-th step in sequential execution data_i is equal to the size of all parallel partial results $\text{data}_{i,j}$

$$\text{data}_i \quad = \quad \sum_{j}^{n} \text{data}_{i,j} \tag{1}$$

for n parallel sub-queries. In parallel execution we fragment the partial results of each execution step according to Equation 1. With n parallel sub-queries we have n times the bandwidth and the transmission time is up to factor n faster (equality for equal data sizes of sub queries). Figure 5b shows the query transmission time with varying bandwidth of the query in Listing 2. The transmission time of parallel execution outperforms the time of sequential execution by a factor of at least 3 for delay 0.2 s and bandwidth smaller than 30 kB/s.

VII. Conclusions

In this paper, we discussed our novel indexing scheme which store the same amount of triples in the network as state-of-the-art index scheme but improved the load balancing and could eliminate hotspots in the network where peers had to manage far more triples than others. We showed by using our new index scheme we can achieve a fair query load distribution and faster query response time by bundling the computation resources and bandwidth of peers with parallelism. We performed query optimization with the goal of improving query response time, and in future work will study the network traffic generated in the parallel processing of RDF queries.

References

[1] G. Klyne and J. J. Carroll, "Resource Description Framework (RDF): Concepts and Abstract Syntax," Tech. Rep., 2004. [Online]. Available: http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[2] M. Cai and M. Frank, "RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 650–657.

[3] E. Liarou, S. Idreos, and M. Koubarakis, "Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks," in *International Semantic Web Conference*, 2006, pp. 399–413.

[4] D. Battré, F. Heine, A. Höing, and O. Kao, "On Triple Dissemination, Forward-Chaining, and Load Balancing in DHT based RDF Stores," in *Proceedings of the 2005/2006 international conference on Databases, information systems, and peer-to-peer computing*, 2007, pp. 343–354.

[5] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. V. Pelt, "GridVine: Building Internet-Scale Semantic Overlay Networks," in *The Semantic Web – ISWC 2004*, vol. 3298. Springer-Verlag, 2004, pp. 107–121.

[6] L. Ali, T. Janson, and G. Lausen, "3rdf: Storing and Querying RDF Data on Top of the 3nuts Overlay Network," in *10th International Workshop on Web Semantics*, Toulouse, France, August 2011, pp. 257–261.

[7] R. Mietz, S. Groppe, O. Kleine, D. Bimschas, S. Fischer, K. Römer, and D. Pfisterer, "A p2p semantic query framework for the internet of things," *Praxis der Informationsverarbeitung und Kommunikation*, vol. 36, no. 2, pp. 73–79, 2013.

[8] L. Ali, T. Janson, G. Lausen, and C. Schindelhauer, "Effects of Network Structure Improvement on Distributed RDF Querying," in *6th International Conference on Data Management in Cloud, Grid and P2P Systems (Globe 2013)*, Prague, Czech Republic, September 2013.

[9] A. Harth and S. Decker, "Optimized index structures for querying rdf from the web," *Web Congress, Latin American*, vol. 0, pp. 71–80, 2005.

[10] T. Janson, P. Mahlmann, and C. Schindelhauer, "A Self-Stabilizing Locality-Aware Peer-to-Peer Network Combining Random Networks, Search Trees, and DHTs," in *ICPADS*, 2010, pp. 123–130.

[11] "SPARQL Query Language for RDF." [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[12] Y. Guo, Z. Pan, and J. Heflin, "LUBM: A Benchmark for OWL Knowledge Base Systems," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 3, no. 2-3, 2005.