Load-balancing in P2P based RDF stores

Dominic Battré and André Höing
and Felix Heine and Odej Kao

2006
Load-balancing in P2P based RDF stores

Dominic Battré, Felix Heine, André Höing, and Odej Kao

University of Paderborn
Paderborn Center for Parallel Computing
Fürstenallee 11, 33102 Paderborn, Germany
{battre,fh,andrehoe,okao}@uni-paderborn.de

Abstract. RDF stores that use structured P2P networks are often based on the idea to store three copies of each triple in a distributed hash table using subject, predicate, and object as keys of the hash function. A common scalability issue of this strategy is the load-imbalance of nodes due to vast differences in the number of occurrences and requests of keys. This paper presents and compares load-balancing strategies for RDF stores in structured P2P networks leading to better performance in networks of many nodes. These strategies are based on the ideas of replicating and relocating data on an overlay tree.

1 Introduction

RDF [1] and RDF Schema [2] have received a lot of attention in recent years for storing ontologies and semantically rich data. Major movements like the Semantic Web [3] and the Semantic Grid [4] demonstrate the potential of this technology and indicate the expected growth of data. We believe that the amount of semantic data available in the future will exceed the capabilities of current RDF data stores and reasoning engines such as Sesame [5] or Jena [6] as we have explained in [7, 8]. Because of scalability issues we consider P2P based approaches superior to centralized RDF databases. Examples for such P2P based approaches include our BabelPeers [7, 8] implementation, Atlas [9], and RDFPeers [10].

All these distributed RDF stores follow a common strategy where RDF triples, consisting of the three components subject, predicate, and object, are stored in a distributed hash table three times, once hashed by each component. This approach has several advantages over centralized repositories. Firstly, data is distributed among many nodes that can process queries concurrently. This increases not only the available storage capacity for triples but also the available CPU power for query processing. Secondly, a global repository eases the integration of data compared to many smaller centralized repositories as suggested by [11].

As RDF data and queries can be represented by labeled directed graphs [1], the core of query processing is the problem of sub-graph isomorphism. Most implementations use backtracking approaches at various levels of sophistication to find matches of a query pattern to the RDF database. BabelPeers for example uses strategies to reduce the search space. A client which tries to process a query
(e.g. a SPARQL query) collects all triples relevant for the query (or a superset thereof) by issuing requests to peers to send all triples matching a certain pattern. A pattern like \( \{ \{ A, B \}, D, ?v1 \} \) can be read as “give me all triples with either A or B in the subject, D in the predicate, and anything in the object.” It can be answered by the peer which is responsible for hash(D) or together by the peers responsible for hash(A) and hash(B). As we use a distributed hash table (DHT), such requests are sent in \( O(\log N) \) routing steps to the responsible peer, where \( N \) is the number of peers in the network. BabelPeers uses these patterns as building blocks to resolve more complex graph queries. Details can be found in [7].

As the RDF data necessary for the query processing is distributed in the P2P network, peers need to gather this information from the network and conversely are exposed to a permanent influx of data requests. Therefore, the performance of the P2P network depends not only on sophisticated query processing algorithms but also on load-balancing that mitigates the problem of bottlenecks. Because queries are often a conjunction of many parts that all need to be processed, a bad performance of a single peer can be detrimental for the overall query processing speed.

The crucial issue of DHT based RDF stores is the violation of a central assumption of generic DHT approaches—the assumption that the hash function distributes the objects uniformly over the value range. Owing to the very nature of RDF data and the hashing scheme, a vast number of collisions are unavoidable. For example, each individual has an rdf:type property which is encoded as a triple with predicate rdf:type. The hash function maps all these triples, one for each individual, to the same ID and therefore to the same peer in the network. This creates a tremendous load for this peer.

But not only predicates suffer from the problem of collisions. All transitive properties, such as rdfs:subClassOf and rdfs:subPropertyOf, foster collisions for the objects of triples if the P2P network uses forward chaining for reasoning. The idea of forward chaining is to generate all knowledge derivable from the original data such that queries can be processed rather easily on the union of generated and original data. If \( B \) is sub-class of \( A \), an individual of class \( B \) is automatically an individual of class \( A \) as well. In a deep hierarchy of classes or properties, the generated data will have many triples with common objects (in this case there will be many triples of type “\(<?v1> \text{ rdf:type } A\)” that collide at hash(rdf:type) and at hash(A)). Even suggestions not to hash triples by predicate rdf:type mitigate but do not solve this problem.

A second issue creating overloaded regions (called hot-spots) is the unequal popularity of data. Most query processors try to order triples in the recursive algorithm such that candidates for triples whose lookup returns the smallest triple set are fetched first. As each peer follows the same strategy, this can create a prevalence of data requests issued to a particular peer which happens to store a small set of triples. The number of queries submitted to this peer can therefore establish another hot-spot.
Finally, the heterogeneous hardware in a P2P network, ranging from CPU capacity over storage capacity to network connectivity, is another serious issue for overloaded peers.

Benchmarks show that load-imbalance creates a serious scalability problem. We have created DHT rings with 2, 4, 8, 16, 32, and 64 peers. Each peer served as an RDF store and continuously processed a set of 100 queries, which of course required the consultation of other peers. The processing of a new query was started instantly after the previous query was completed.

![Figure 1: Query throughput in DHTs of different sizes](image)

Figure 1 shows the performance decline if the P2P network does not employ any load-balancing. Conversely, it demonstrates the positive impact of two simple load-balancing strategies we show in this paper.

In section 2 we present an overview of recent load balancing strategies for DHTs. Section 3 explains load-balancing strategies that focus on rebalancing only those peers that appear overloaded. Section 4 gives an algorithm to detect these overloaded peers. Section 5 contains the evaluation of benchmarks and section 6 concludes the paper.

## 2 Related Work

Zhu and Hu [12] give a very good overview about recent research on load-balancing in DHTs based on a paper by Rao et al. [13]. It shows three principal approaches being investigated in the past few years.
The authors of Chord [14] propose to start up to $\log N$ “virtual nodes” on each actual node, where $N$ is the number of actual nodes in the network. Then nodes have independent hash identifiers to achieve an average bound on the load-imbalance. Rao et al. [13] build on this idea and allow virtual nodes with high utilization to migrate to less utilized peers in order to achieve better load balance. Zhu and Hu incorporate proximity information in this approach.

Karger and Ruhl [15] migrate under-utilized nodes to ranges of the DHT that are highly populated. This helps if order preserving hash functions map many objects onto a small range of the DHT.

Neither of these approaches is capable of handling the discrete hot-spots encountered in DHT based RDF stores as illustrated in figure 2. Owing to the scarcity of different URIs in a schema (represented by URIs $U_1$ to $U_5$) one observes discrete peaks of many collisions in the DHT. Approaches as described above are incapable of splitting this load to different nodes.

Byers et al. [16] propose the “power of two” principle in which each element can be inserted into a DHT according to two different hash functions. Each insert operation is preceded by asking the peers responsible for the respective ranges of the DHT for their load. An object is then inserted at a location covered by the less utilized peer while the location of the higher utilized peer references to the other one. This approach is very close to what we will describe as static replication below. We will argue below why this approach fails to do proper load-balancing in the case of DHT based RDF stores.

3 Strategies

There are two major categories of load-balancing strategies that we consider in this paper, namely the replication of data and the relocation of data. The following sections discuss four strategies that fall into these categories.


3.1 Replication

**Static replication** The simplest approach of load-balancing by replication defines *a priori* a static number $R$ of replica for each triple. Replication can be achieved for example by using not only one hash function for inserting triples but a whole family of $R$ hash functions. Each insert operation is repeated once with each hash function; a lookup uses a randomly chosen hash function.

This strategy can be implemented easily and is often offered by P2P middlewares natively (e.g. by FreePastry). The disadvantage of this approach is the bad scalability. The load of the most occupied nodes can be reduced only by a factor of $\frac{1}{R}$, and if only $p\%$ of the peers of a network are responsible for any RDF triples at all, this fraction of peers can be augmented to at most $R \cdot p\%$. This can be still few peers compared to the total network size. Further, this kind of replication balances the load in terms of query pattern processing, but peers responsible for frequent URIs still have to store a large number of triples.

It is difficult to find an appropriate value for $R$ in advance and as $R$ is global knowledge it is difficult to change this value at runtime. Even though bigger values for $R$ improve the load-balancing, it cannot be chosen arbitrarily high. All replicas need storage capacity and create network load because of update messages due to soft state implementations. The idea of soft state is that triples have a life time and expire unless they are refreshed by their originators. This creates a permanent flow of update messages proportional to the number of triples stored in the DHT.

As most DHT based P2P networks require replication anyway, it makes sense to exploit this replication with a small constant $R$ for load-balancing. From the argument above, however, we conclude that static replication with a constant factor $R$ is insufficient because it does not scale up to arbitrary large networks.

**Dynamic replication** As the load of peers differs greatly, it is desirable to replicate only triples of those nodes that suffer from very high loads. This provides load-balancing while keeping storage requirements and the overhead of update messages within limits.

In order to initiate the replication of data due to load, we need to define a reasonable load measure that can be determined at run-time. Examples for such load measures are the number of messages received by a peer within a certain time, the total number of bytes sent in result sets by a peer within a certain time, or the average wait time of queries in a queue. Each of these measures is easy to determine locally for a single node but difficult to determine for the global network. The global values (or approximations) are necessary, however, to evaluate the relative performance of a peer. The next section discusses sampling techniques to detect whether a node suffers from high load.

Dynamic replication follows a simple pattern once a load-imbalance was detected. From the sampling technique to be discussed, a node knows the load of several peers. This allows to copy its content to the two least occupied nodes known and to forward subsequent queries to either of these nodes picked at random each time. Recursively applying this procedure creates an overlay-tree for load-balancing in which the leaf nodes do the actual query processing.
3.2 Relocation

As replication increases the number of triples in the network that need to be stored and further need to be updated in order to prevent their soft-state expiration, relocating triples becomes an interesting alternative to replication. We propose two simple relocation strategies based on the idea of constructing a binary overlay tree.

**Binary splitting with query broadcasting** It is possible to split the data of an overloaded node randomly into two or more equal sized parts which are disseminated to those peers that appeared to be least busy during the load-detection phase. These nodes become children in the overlay tree. The union of the children’s result sets resembles the result of the parent peer. As a split peer has to broadcast incoming requests to all of its children and because the peer issuing the original request has to collect all sub-result sets, the total number of messages increases in the network. As triple patterns in the queries are much smaller than result sets, we expect the increase in the number of messages to be of little influence.

**Binary splitting with query unicasting** If components of triples (subjects, predicates, objects) have a total order, for example by using lexicographical comparison, it is easy to define a total order on full triples as well. That way, it is possible for a peer in the overlay tree to memorize a split-element, such that all triples less than or equal the split element are located at the left child and all remaining triples are located at the right child. Queries can sometimes be routed to only one of the children instead of being broadcasted to them. This is particularly important for Top \( k \) searches (see [8]) where requests have the form “give me up to \( k \) triples matching to the triple pattern \((A, \{A, B, C\}, ?v1)\) that appear after triple \((A, B, A)\).”

3.3 Summary

The focus of this paper is on replication and relocation based on dynamic criteria. These approaches use binary or \( n \)-ary overlay trees. Routing messages to the destination in an overlay tree requires \( O(\log N + d) \) routing steps, where \( N \) is the number of peers in the DHT and \( d \) is the depth of the tree. The \( O(\log N) \) part originates from the DHT routing of a message to the root of an overlay tree. All messages within the tree can be sent directly from a parent node to its children without DHT message routing. Therefore, we do not expect this to have a serious impact on the performance.

4 Overload Detection

Figure 3 shows a histogram of the distribution of the number of bytes sent by peers to answer query pattern requests in a simulation of 1024 peers with no load-balancing. The histogram shows a distribution very typical of various load-indicators: The vast majority of peers has values close to zero while few peers
have very high values. As the slow nodes of a DHT determine the number of queries that can be processed per time unit, the goal is to shift the load of these slow nodes to others.

Note the rug below the bars of the histogram in figure 3 that indicates encountered loads of peers that did not add up to visible bars. Without defining a crisp border, it appears reasonable to consider peers with loads to the right of the visible bars ($>0.1e+05$) as outliers. The central problem is that a peer needs to detect whether it shall be considered an outlier without knowing the entire distribution.

For that reason we use a sampling technique, where each peer sends its statistics to 25 (this number is called sample size, SS) randomly generated peer-IDs after a certain period of time. At this point it compares its computed load to the 25 most recently received values from other peers. In case the local load ex-
ceeds the maximum of the remote load values by a certain \textit{sample factor} (SF), it initiates a replication or split of the local data.

This approach shows some very nice features that we can observe in the lower part of figure 3. The graph shows the probability (y-axis) that a peer with a certain load (x-axis) collects a sample of remote load statistics that requires the local peer to be split. We see that this probability is zero for the majority of nodes and then increases the larger the load of a local peer is. Therefore, only nodes with very high load are split.

When considering loads $>0.1e+05$ as outliers, the algorithm does not produce any false positives (nodes that are split even though their load is smaller than the bound). This has proven to be important because too many splits create a tremendous network load that throttles the overall performance. On the other hand, the algorithm produces several false negatives. This, however, is no problem because peers with high load not detected to be outliers will be reevaluated in a later iteration. The probability of a high load peer not to be detected $k$ times is $1 - (1 - p)^k$, where $p$ is the probability of a split or replication in one iteration. As this tends to 0 for large $k$, we expect high load peers to be discovered eventually.

Another positive feature of this load-detection strategy is its convergence. The splitting of high load peers has hardly any impact on the lower bound where the splitting probability shown in the lower part of figure 3 grows $> 0 + \epsilon$.

Splitting based on the median of a sample turned out not to work well. Medians of samples were often found to be 0 such that any node initiated a split, even if it stored only two triples. Using the mean and a large factor might work but it is difficult to justify any chosen sample factor.

\section{Benchmark and Analysis}

Owing to the lack of access to resources with many nodes, our analysis of load-balancing strategies falls back to simulations. We will first describe the experimental setup, then argue why we expect the simulation to deliver representative data, and finally analyze the results of the simulation.

\subsection{Experimental Setup}

RDF and RDF(S) data can be characterized by many factors such as the number of classes, properties, and individuals (instances of classes). Also the degree of inheritance, i.e. many individual classes, a very deep hierarchy, classes with few or many ancestors, and so on, has a high impact on performance benchmarks. As it is difficult to find \textit{representative} real world data, we have chosen to generate test data artificially. Our data does not claim to model real data, but shall rather give an example that exhibits the problems of load-imbalance in DHT based RDF stores.

We have generated the test data in multiple steps. The algorithm starts with generating a \textit{class hierarchy} and a \textit{property hierarchy} and then proceeds to generating triples from these hierarchies.
The hierarchy generation is driven by three parameters, viz. the number of classes/properties, the depth, and the maximum number of inheritances per node. Figure 4 gives an example of how 9 different classes are distributed over 3 levels. Each class has up to two classes it is derived from. Once hierarchies for the classes and the properties are generated, we create many instances from the classes. Finally, randomly chosen instances are connected by random properties. These connections produce the triples.

For the following benchmarks we have created an RDF graph of 60,000 triples that describe 10,000 individuals from 200 classes connected by 50,000 triples from 200 different types of properties. The classes and properties hierarchies were generated as described above with 20 and 10 levels respectively in the hierarchy and at most 3 ancestors per node. Reasoning (forward chaining) augmented this RDF graph by approximately 135,000 additional rdfs:type triples and 330,000 additional connections between URIs.

The queries were generated starting with a random subgraph of the RDF graph. In this subgraph, we have then randomly replaced URIs with variables. Thus each query has at least one answer, up to over 1,000,000 answers in some cases.

In order to quantify the load-balance, we measure the average time, a triple pattern remains in the in-queue of a peer before this peer starts processing the query. If these wait-times are large for one node, this peer is overloaded and slows down the query-processing of other peers.

Figure 5 shows Lorenz-curves [17] of the load-imbalance as defined above. For each node we add the queuing times of all triple patterns and sort the nodes in ascending order. The figure then shows for how many percent of the summed queuing times the first $p\%$ of peers account. It shows for example for 8 nodes that the 75% percent of the nodes with the least queuing times amount together for only 25% of the sum of all queuing times in the network. The Lorenz-curves show that the load-imbalance increases with the number of peers in the network. As idle peers do not contribute to the performance of the network, we consider load-imbalance the major factor for a decline in query performance as shown in Figure 1.
5.2 Simulation

The strategies described above leave a wide range concerning the concrete implementation and choice of parameters. First, we have to choose whether to optimize the storage load or the query throughput. This drives the choice of the key indicator used in overload detection and the balancing strategy. In this context, we investigate whether it is possible to balance storage load and query performance simultaneously. After picking a load indicator, we have to choose a sampling strategy and its parameters. We close with an analysis of the overhead produced by our strategies compared to their effect.

When choosing to balance storage load, the number of triples stored at each node is used as key indicator during overload detection. However, if we chose to maximize query throughput, we have multiple indicators at hand. In real-life scenarios, we can use the queue wait times. However, in the simulation, we do not have this indicator. We can either use the number of triple requests or the accumulated number of bytes sent in response to these requests. Therefore, we have analyzed in small-scale (not simulated) benchmarks whether these indicators can be used as indicators for the average queuing time.

Figure 6 shows the correlation between the two factors and the queuing time of triple patterns. Fig. 6c and 6d show the plot after removing the two outliers shown in figures 6a and 6b. We can see a strong correlation between the number of bytes a peer sends and the average waiting time of this peer. On the other hand, we do not find a strong correlation that supports our initial hypothesis that hot-spots are created by a large number of requests even though the results are small. For that reason, we consider approaches successful if they balance the load regarding the number of bytes sent per peer. In heterogeneous environments this assumption does not hold of course.
Fig. 6: Effect of the factors “total bytes sent by peer” and “number of query-patterns processed by peer” on the average waiting time of query-patterns to be processed.

In order to better quantify the load-imbalance we use the Gini index. It expresses the ratio of the area between the 45 degree line and the Lorenz curve and the area under the 45 degree line, which is 0.5. It ranges from 1 to 0; lower values indicate smaller imbalances between the data.

Figure 7 shows the development of various Gini indexes over time. We have executed 30 turns of processing 300 queries on a DHT of 1024 nodes. After each turn, the nodes used the sampling based load-balancing strategy described above. We have used recursive replication and split strategies whose criterion for detecting load-imbalance were the number of bytes sent and the number of triples stored. We did not use the replication strategy to balance the number of
stored triples, because this creates many peers with very high numbers of triples that split recursively in each iteration.

The two diagrams of figure 7 show the effect of the load-imbalances regarding the number of bytes sent and the number of triples stored. The replication by number of bytes sent did not complete all 30 turns because the growth in data exceeded the machine’s 4 GB of main memory.

First we look at the left diagram, where we balanced the number of bytes sent. The strategy “Split by Bytes Sent” is by far the best. In the first iterations the Gini index drops from approx. 0.85 down to 0.6. The replication strategy is not that successful in this case. Balancing the storage load decreases the imbalances regarding number of bytes sent only at a very low rate. The right diagram shows the development of storage imbalances when using the same strategies. Clearly, the “Split by #Triples” strategy wins. However, we see that balancing the number of bytes sent either by splitting or by replication makes the storage imbalance even worse. The splitting strategy, however, is again superior to the replication.

To summarize, we have two main choices. Either we maximize query performance using the splitting strategy, which increases the storage imbalance slightly, or we choose to balance the storage load, which leads to persisting performance problems during query processing.

Figure 8 compares how the parameters of the sample size (SS) and sample factor (SF) influence the development of the Gini index and how much maintenance overhead is needed, expressed in number of splits initiated. We can observe that the effect on the Gini index correlates with the number of splits. SS=25, SF=1.5 has a large impact on the Gini index, while using the highest number
of splits. We can further observe that smaller values for both SS and SF lead to a better load balancing. All versions tend to stabilize in the long run, after an initial number of iterations where numerous splits are performed.

6 Conclusion

We have presented and evaluated load-balancing strategies that distribute the load of individual overloaded peers while not distributing the load of low loaded peers. The load-balancing strategies are based on recursively extending an overlay tree where nodes either split their local triple stores and assign it to their children in the overlay tree or replicate their content. We have further introduced a sampling technique to determine whether a peer suffers from high load and therefore throttles the total query-processing throughput in the network.

These load-balancing strategies have proven to be successful in benchmarks of up to 64 nodes (see figure 1). In simulations we have shown that the strategies are capable of distributing the load of overloaded peers even for larger networks.

We have further observed that balancing the storage load and the query load simultaneously is not possible, at least with the presented strategies. We have further shed some light on the trade-off between maintenance cost and efficiency of a load-balancing strategy. However, we have seen that even those strategies with a large balancing effect come to a state where the maintenance effort is reasonably small.

References