SPARQL Query Optimization on Top of DHTs*

Zoi Kaoudi, Kostis Kyzirakos, and Manolis Koubarakis

Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens, Greece

Abstract. We study the problem of SPARQL query optimization on top of distributed hash tables. Existing works on SPARQL query processing in such environments have never been implemented in a real system, or do not utilize any optimization techniques and thus exhibit poor performance. Our goal in this paper is to propose efficient and scalable algorithms for optimizing SPARQL basic graph pattern queries. We augment a known distributed query processing algorithm with query optimization strategies that improve performance in terms of query response time and bandwidth usage. We implement our techniques in the system Atlas and study their performance experimentally in a local cluster.

1 Introduction

With interest in the Semantic Web rising rapidly, the problem of SPARQL query processing and optimization has received a lot of attention. This paper concentrates on the optimization of SPARQL queries over RDF data stored on top of distributed hash tables (DHTs). The first such implemented P2P system is RDFPeers [1] where only a restricted query class is supported (conjunctive multi-predicate queries). In [11], we have extended the work of RDFPeers and presented two algorithms for the distributed evaluation of conjunctions of triple patterns. The algorithms in [11] have been evaluated only by simulations and no query optimization techniques or an implemented system has been presented. Motivated by [1, 11], our group has been developing Atlas (`http://atlas.di.uoa.gr`), a full-blown open source P2P system for the distributed processing of RDF(S) data stored on top of DHTs. The RDFS reasoning functionality, the architecture and various applications of Atlas are presented in [7–9].

In this paper, we present for the first time the *query optimization techniques* we have developed in Atlas, and evaluate them experimentally. Although query optimization has been extensively studied and is widely used in the database area, SPARQL query optimization has been addressed only recently even in centralized environments [12, 13, 23]. The first works that dealt with distributed query optimization of SPARQL queries are [17, 24]. However, the architecture proposed in these papers is very different from the one offered by a DHT. In [10] a DHT-based system is presented which supports a SPARQL-like query language and utilizes optimization techniques complementary to the ones we propose.

* This work was partially supported by the European project SemsorGrid4Env.

P.F. Patel-Schneider et al. (Eds.): ISWC 2010, Part I, LNCS 6496, pp. 418–435, 2010.
© Springer-Verlag Berlin Heidelberg 2010

In this work, we address SPARQL query optimization over RDF data stored on top of DHTs and target the minimization of the time required to answer a query and the network bandwidth consumed during query evaluation. Our work starts from the QC algorithm of [11] which we enhance with a query graph representation to avoid Cartesian products and with a distributed mapping dictionary (Section 3). Although mapping dictionaries are by now standard in centralized RDF stores [2, 12, 27], our paper is the first that discusses how to implement one in a DHT environment. In addition, we fully implement and evaluate a DHT-based optimizer which is used to find the best ordering of a query's triple patterns. We describe three greedy optimization algorithms for this purpose: two static and one dynamic. These algorithms utilize selectivity estimates to determine the order with which triple patterns should be evaluated in order to improve query response time and network bandwidth consumption (Section 4). We also propose methods for estimating selectivities of SPARQL basic graph pattern queries utilizing techniques from relational databases (Section 5). We discuss which statistics should be kept at each peer and use histograms for estimating data distributions. We demonstrate that it is sufficient for a peer to create and maintain local statistics, i.e., statistics about the data values for which it is responsible. These statistics can be obtained by other peers by sending low cost messages (Section 6). We implement all our techniques in the system Atlas and present an extensive experimental evaluation in a local cluster using the widely used LUBM benchmark [4] (Section 7).

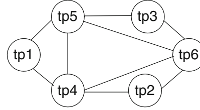
2 System and Data Model

System Model. We assume a structured overlay network where peers are organized according to a DHT protocol. DHTs are structured P2P systems which try to solve the *lookup problem*; given a data item x, find the peer which holds x. Each peer and each data item are assigned a unique m-bit identifier by using a hash function. The identifier of a peer can be computed by hashing its IP address. For data items, we first have to determine a *key* k and then hash this key to obtain an identifier id_k. A Lookup(id_k) operation returns a pointer to the peer responsible for the identifier id_k. Atlas uses the Bamboo DHT [18] but our algorithms can be implemented on top of any DHT network. When a peer receives a Lookup request, it efficiently routes the request to a peer with an identifier that is numerically closest to id_k. This peer is responsible for storing the data item with key k and we will call it *responsible peer for key* k.

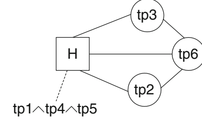
Data Model and Query Language. We assume that the reader is familiar with the notions of RDF *triple* and *triple pattern*. We deal with RDF triples with no blank nodes and SPARQL queries of basic graph patterns (BGP). A SPARQL query with filter expressions involving equality operators can be easily rewritten to a BGP query. In the following, we define an internal representation of a query extending the graph-based approach used in [12, 23].

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub:<http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?x ?y ?z
WHERE {
?x rdf:type ub:Student .  (tp1)
?y rdf:type ub:Faculty .  (tp2)
?z rdf:type ub:Course .   (tp3)
?x ub:advisor ?y.         (tp4)
?x ub:takesCourse ?z .    (tp5)
?y ub:teacherOf ?z . } (tp6)
```

(a) SPARQL query



(b) Initial graph



(c) Intermediate graph

Fig. 1. Query example

Definition 1. *A* query graph *g is a tuple (N, H, E), where N is the set of nodes in g, H is the set of hypernodes in g and E is the set of undirected edges in g. Each node in N denotes a single triple pattern and each node in H denotes a conjunction of triple patterns. Two nodes from $N \cup H$ are connected with an edge in E if and only if the triple pattern or the conjunction of triple patterns represented by these two nodes share at least one variable.*

Initially, the query graph of a query consists only of simple nodes. During query processing, evaluated triple patterns are merged into hypernodes. In the rest of the paper, we focus only on connected graphs. The evaluation of unconnected graphs is straightforward since each connected subgraph can be evaluated independently, and then the union of the results can be created at the peer that posed the query. A *query plan* q_g for graph g is a total order of the nodes in N of g. In Fig. 1(a), we present an example SPARQL query which will be used throughout the paper (LUBM Q9 [4]). The initial query graph is shown in Fig. 1(b). Figure 1(c) shows an intermediate query graph where the hypernode H represents the conjunction of triple patterns $tp1 \wedge tp4 \wedge tp5$ (i.e., triple patterns $tp1$, $tp4$ and $tp5$ have already been evaluated).

3 Query Evaluation

We start by first explaining the triple indexing scheme we have adopted from [1], where each triple is indexed in the DHT three times. The hash values of the subject, predicate and object of each triple are used to compute the identifiers that will indicate the peers responsible for storing the triple. The peer that receives a store request for a set of RDF triples uses a MULTISEND message as in [11] to distribute the triples among the peers. Each peer keeps its triples in a local database consisting of a single relation with four columns (*triple relation*). The first three columns correspond to the three components of the triples stored, while the fourth column indicates which of the three components is the key that led the triple to this peer.

Query processing. The algorithm we use (QC*) is based on algorithm QC of [11]. Unlike [11] that uses lists of triple patterns, we employ the query graph representation, which ensures that no Cartesian product will be computed and transferred through the network.

When a peer receives a query request, it translates it into a query graph g. Based on the query plan generated by the optimizer, the peer also marks the node of the query graph which represents the triple pattern that should be evaluated first and sends a `QEval` message to the peer that will start the query evaluation. Figure 2 shows the pseudocode when such a message arrives at a peer p. Keyword **event** is used for handling messages also indicating the peer where the handler is executed. Keyword **sendto** prefixed by an identifier declares that the message should be sent to the peer which is responsible for this identifier. In this case, a LOOKUP operation is performed first to discover the peer responsible for this identifier and then the message is sent directly to this peer.

```
Algorithm 1: QC*
1 event p.QEval(id, g, interRes, vars, retIP)
2     lR:=MATCH (g.marked_node().triplepattern());
3     if interRes = {} then interRes':=lR;
4     else interRes':=lR join interRes;
5     if interRes' = {} then
6         sendto retIP.queryResp({ });
7         return;
8     end
9     g':=g.MERGE(g.hypernode, g.marked_node);
10    if g'.N={} then   //all triple patterns are evaluated
11        answer := project interRes' on vars;
12        sendto retIP.queryResp(answer);
13        return;
14    end
15    project out unnecessary vars from interRes';
16    g'.MARKNEXTNODE();
17    key':=FINDKEY(g'.marked_node().triplepattern());
18    id':=HASH(key');
19    sendto id'.QEval(id',g',inteRes',vars, retIP);
20 end event
```

Fig. 2. QC* algorithm

First, peer p evaluates the triple pattern which correspond to the marked node of query graph g and forms a temporary relation lR by posing a selection query to its triple relation. If relation $interRes$, which holds the intermediate results so far, is empty, peer p is the first peer of the query evaluation and assigns lR to $interRes'$. Otherwise, p assigns to $interRes'$ the natural join of lR and $interRes$. If the result of the join is an empty relation, p returns an empty set to the peer that posed the query (peer with IP address $retIP$) and query evaluation terminates. Otherwise, query evaluation continues and peer p merges the marked node with the hypernode in g creating a new query graph g'. In case peer p is the first peer participating in the query evaluation, p just transforms the marked node n into a hypernode. If the new graph g' consists only of a hypernode, all triple patterns have been evaluated and p computes the projection of $interRes'$ on the answer variables $vars$ and sends the answer to the peer with IP address $retIP$. Otherwise, query evaluation continues and p projects out from $interRes'$ variables that neither appear in $vars$ nor in the rest of the triple patterns. Then, a new node in g' is marked as the next triple pattern that should be evaluated and p sends a new `QEval` message to the next peer[1]. Local procedure MARKNEXTNODE ensures that the chosen node is connected with the hypernode of g', so that no Cartesian product will be computed.

Mapping dictionary. QC* utilizes a distributed mapping dictionary which replaces long strings (URIs and literals) by unique integer values. Triple storage and query evaluation is, then, performed more efficiently using these integers.

[1] We assume that each triple pattern has at least one bound component. The case where all three components of a triple pattern are variables requires a slightly different implementation which we do not discuss here.

The uniqueness of the integer values used in the mapping dictionary could be ensured in various ways. We propose the following scheme which is fully distributed (thus scalable and fault tolerant) and does not require any kind of coordination between the peers. Each peer keeps a local integer counter consisting of l bits which is initially set to 0. l is incremented by 1 everytime a new integer value needs to be generated. Each peer that joins the network is assigned a unique m-bit identifier by hashing its IP address. We create an n-bit identifier for a triple component by concatenating the m bits of the peer's unique identifier with the l bits of the current local counter. Depending on the network setting and the application requirements, we can determine an appropriate value for l so that each n-bit identifier is of reasonable space.

When a peer receives a store request, it transforms the given triples into new triples containing integers. The peer, then, sends the new set of triples to be stored in the network using MULTISEND. Together with the new triples, it also sends the mapping from strings to integers that created these triples. Note that we use the *string values* of the triple's components as keys to create the identifiers. Each peer that receives a MULTISEND message, stores in its triple relation the triples it is responsible for. Each such peer also maintains a two-column relation which serves as a local dictionary which holds the mappings for *all* the components of its local triples (*dictionary relation*).

During query evaluation, each string appearing in the triple patterns of a query is transformed into the corresponding unique integer. This transformation is performed during the lookup operation as follows. Whenever peer y wants to send a `QEval` message for triple pattern tp, it first sends a LOOKUP request to determine the peer responsible for this triple pattern (peer p). Peer p, which receives the LOOKUP request, retrieves the integers corresponding to the strings of tp from its local dictionary relation and sends them to y together with its IP address. Then, peer y replaces the strings of tp with the integers sent from peer p and continues query processing. In case any of the strings has no assigned integer, the answer to the query is empty and query processing terminates. Finally, the peer which computes the answer to the query is responsible for replacing integers in the triples of the answer set with their string values. To achieve this, it contacts the least possible number of peers that have already participated in the query evaluation and have the appropriate values in their dictionary relation. During query evaluation, the IP address of these peers is appended within the `QEval` message (using an extra parameter).

4 Query Optimization Algorithms

The goal of query optimization is to find a query plan that optimizes the performance of a system with respect to a metric of interest. In our work, we are interested in improving the time required to answer a query (*query response time*) and the *network bandwidth* consumed. The query response time of our algorithm can be improved if the time spent for query evaluation locally by each peer and the time required for network messages to reach relevant peers is improved. One way to accomplish this is by minimizing the size of intermediate

relations produced during query evaluation (*interRes*$'$ in QC*). In this case, we benefit in two ways: first, we achieve lower bandwidth consumption and second, we accomplish the computation of joins with smaller intermediate relations locally at peers. Lessons learned from earlier versions of Atlas persuaded us to concentrate on optimizing these metrics to improve the scalability of our system.

In the following, we present three greedy optimization algorithms which try to minimize the size of intermediate relations produced by the query processing algorithm utilizing selectivity-based heuristics. We describe both static and dynamic optimization algorithms. The two static query optimization algorithms are completely executed by the peer that receives the initial query request and output a fully specified query plan (an ordered list of triple patterns). In the dynamic query optimization algorithm, optimization decisions take place at each step of the query processing algorithm. Using standard terminology from relational systems, the *selectivity of a triple pattern tp*, $sel(tp)$, is the fraction of the total number of triples in the network that match *tp*. Similarly, if H is a conjunction of triple patterns, the *selectivity of the conjunction of triple patterns*, $sel(H)$, is the fraction of total number of triples in the network that match H. We later discuss how we can estimate these selectivities (Section 5).

Naive static algorithm. The naive algorithm (NA) orders triple patterns based on their selectivity (from the most selective to the least selective) and in a fashion where a Cartesian product computation will not be required. The optimization algorithm works as follows. Using the initial query graph representation, each node is assigned with the selectivity of the corresponding triple pattern. The algorithm firstly selects the query graph node n_0 with the minimum selectivity and adds it to the query plan. Then, it marks the nodes that are connected with n_0 and removes n_0 from the graph. The algorithm iteratively chooses the node n_{min} with the minimum selectivity, selecting only from the marked ones, adds it to the query plan, marks the nodes connected with n_{min} and removes n_{min} from the graph. The algorithm terminates when no nodes are left in the graph. NA is based on the assumption that after joining two very selective triple patterns, the joining result will also be selective. Certainly, this assumption is not always true, but the algorithm often performs in a satisfactory way, as we will see in the experimental section.

Semi-naive static algorithm. The semi-naive algorithm (SNA) is a variation of the minimum selectivity algorithm [22] and has also been used in [23]. SNA goes beyond NA by taking into account the selectivity of *pairs* of triple patterns. Besides assigning each node of the graph with the selectivity of its triple pattern, each *edge* of the graph is also assigned with the selectivity of the conjunction of the connected triple patterns. The algorithm begins by selecting the edge with the minimum selectivity, orders its nodes based on their selectivity and adds them to the query plan. Then, SNA iteratively chooses the edge that has the minimum selectivity, but also has one of its nodes in the query plan, and adds the other node to it. SNA terminates when all nodes have been added to the

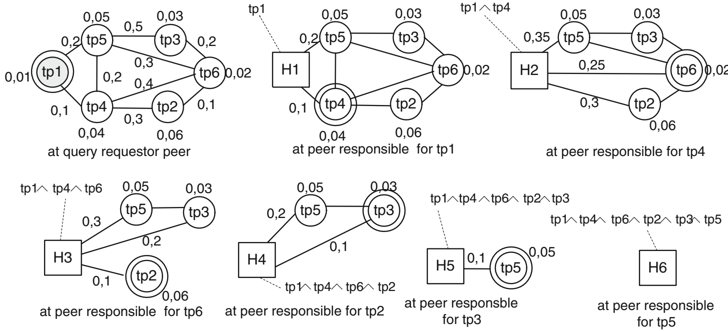


Fig. 3. Dynamic query optimization example

query plan. In case of a tie between the selectivities of two edges, the algorithm chooses the one that has the node with the smaller selectivity.

Dynamic algorithm. Finally, we propose a dynamic optimization algorithm (DA) which seeks to construct query plans that minimize the number of intermediate results during query evaluation. Initially, the peer that received the query request, assigns all edges and nodes of the query graph with the corresponding selectivities and chooses the first triple pattern to be evaluated as in SNA. Then, the optimization step is carried out at each peer p which receives a `QEval` message. After the new query graph g' with the new hypernode H' has been created at peer p, p selects the triple pattern that should be evaluated next. The candidate triple patterns are the triple patterns of the query graph nodes which are directly connected with the hypernode H'. In this way, the computation of a Cartesian product is avoided. Peer p estimates the selectivity of the join between the intermediate results so far (which correspond to H') and each candidate triple pattern and assigns the corresponding edges. Then, peer p selects the node which is connected to H' with the minimum edge selectivity. In case a tie between the selectivities of two edges emerges, p chooses the node with the smaller selectivity. Figure 3 shows an example execution of DA. The query requestor peer assigns the edges and nodes of the query graph and chooses $tp1$ as the first node. At each query processing step, each peer finds the edge with the minimum selectivity from the set of edges connected to the hypernode and marks the corresponding node (shown with a double circle). In the last step, the query graph consists only of the hypernode.

5 Selectivity Estimation

In this section, we propose methods for estimating the selectivity of single triple patterns as well as the selectivity of a conjunction of triple patterns. To achieve this, we need to compute statistics from the data stored in the network. Section 6 describes how these statistics are generated.

5.1 Single Triple Patterns

We present two ways to estimate the selectivity of a single triple pattern; one based on a simple heuristic also presented in [23] and one based on an analytical estimation technique using the attribute value independence assumption [20].

Bound-is-easier heuristic. We consider a simple variation of the standard bound-is-easier heuristic of relational and datalog query processing [25], also used in [23], and assume that the more bound components a triple pattern has, the more selective it is. We further enrich this heuristic by considering the position of the bound components of a triple pattern, if two triple patterns have the same number of bound components. In this case, we assume that subjects are more selective than objects, which in turn are more selective than predicates.

Analytical estimation. Given a triple pattern $tp = (s, p, o)$, where s, p, o are variables or constants, the selectivity of tp using the attribute value independence assumption [20] is computed by the formula $sel(tp) = sel(s) \times sel(p) \times sel(o)$, where $sel(s), sel(p), sel(o)$ are the selectivities of the triple pattern's components. We assume a selectivity of 1.0 for the triple pattern components which are variables as well as for the predicate value `rdf:type`. The selectivity of the other components depends on the frequency with which their value appears in the set of triples stored in the network. We define the frequency of a triple component c with value v (denoted by $freq_c(v)$ where $c \in \{S, P, O\}$) as the total number of occurrences of value v as a triple component c in the set of triples stored in the network. For example, $freq_S($`ub:zoi`$)$ is the number of occurrences of value `ub:zoi` as a subject, while $freq_O($`ub:zoi`$)$ denotes the number of occurrences of value `ub:zoi` as an object in the set of triples stored in the network.

The selectivity of a triple pattern component $c \in \{S, P, O\}$ with value v can now be computed by the formula $sel_c(v) = \frac{freq_c(v)}{T}$, where $freq_c(v)$ is the frequency of value v as a component c and T is the total number of triples. Although in [23], the attribute value independence assumption is also used, their method assumes a uniform distribution for subjects and requires a bound predicate for the objects. In Section 6, we describe how $freq_c(v)$ is computed. For the computation of the total number of triples indexed in the network (T), we use a broadcast protocol. More elegant solutions for distributed counting in P2P are proposed in [14], but adopting such a method is out of the scope of the paper.

5.2 Conjunction of Triple Patterns

The selectivity of the conjunction of *two* triple patterns tp_1 and tp_2 is $\frac{joinCard(tp_1, tp_2)}{T^2}$, where $joinCard$ is the number of tuples (cardinality) of the relation that results from joining tp_1 and tp_2 and T is the number of triples stored in the network.

To compute the expression $joinCard$, we adopt a method proposed for relational systems in [25]. Assume that we have two triple patterns tp_1 and tp_2 and the corresponding relations R_1 and R_2 which contain all the tuples formed with

values existing in the triples stored in the network that satisfy tp_1 and tp_2. Relations R_1 and R_2 have as attributes the variables of triple patterns tp_1 and tp_2, respectively. Since we deal with triple patterns that have at least one constant component, two triple patterns can share at most two variables. The cardinality of joining R_1 with R_2 is computed by the formula:

$$joinCard(R_1, R_2) = \frac{T_{R_1} \times T_{R_2}}{max(I_{R_1(?x_1)}, I_{R_2(?x_1)}) \times max(I_{R_1(?x_2)}, I_{R_2(?x_2)})}$$

where T_{R_1} and T_{R_2} are the number of tuples of R_1 and R_2 respectively, $?x_1$ and $?x_2$ are the variables shared by tp_1 and tp_2, and $I_{R_i(?x_j)}$ is the size of the domain of attribute $?x_j$ of relation R_i. In other words, T_{R_i} is the number of triples that match tp_i, and $I_{R_i(?x_j)}$ is the number of distinct values that variable $?x_j$ has in the bindings of tp_i. This formula can easily be adapted to the case that tp_1 and tp_2 share less than two variables. In [23], the authors propose to precompute the join cardinality by executing the actual SPARQL queries which can become a very expensive operation, especially in a distributed environment.

We can easily determine the number of triples T_R that match a triple pattern tp. If tp has one bound component c with value v, then the number of triples that match tp is equal to the number of occurrences of value v as a component c, i.e., $T_R = freq_c(v)$. If a triple pattern tp has two bound components, then we compute the number of triples that match tp using the selectivity of the triple pattern as explained earlier, i.e., $T_R = sel(tp) \times T$, where T is the total number of triples stored in the network. For the computation of the size of the domain of a variable $?x$ in a triple pattern tp, namely $I_{R(?x)}$, we distinguish two cases. If tp has one variable, then $I_{R(?x)}$ is equal to the number of bindings of variable $?x$. Since no duplicate triples exist in the network and $?x$ is the only variable in the triple pattern, each binding will be unique. In this case, we have $I_{R(?x)} = T_R$. In the case where tp has two variables, the corresponding domain size for the shared variable can be determined using the techniques of Section 6.

We now discuss the use of the above selectivity estimation techniques by the optimization algorithms of Section 4. While NA requires only the selectivity of single triple patterns and thus both the bound-is-easier heuristic and the analytical estimation can be applied, SNA and DA require also the selectivity of conjunctions of triple patterns and hence only the analytical estimations will be used. Especially in DA, the estimation of the selectivity of the join between the intermediate results (R_1) and one triple pattern (R_2) is required. The formulas are the same as described above with the exception that relation R_1 is already formed. Therefore, the number of tuples of R_1 and the domain size of any variable in the attributes of R_1 can be computed on the fly by examining relation R_1.

6 Statistics for RDF

In this section, we present an efficient DHT-based scheme for collecting and using the statistics that enable the estimation of the selectivities described in Section 5. These statistics are the frequency of a triple component and the size of

the domain of a variable in a triple pattern. Peers keep statistics only from their *local data* and specifically for the data values for which they are responsible (i.e., values that are the *keys* that led a triple to a specific peer). These turn out to be *global statistics* required by the optimization algorithms and can be obtained by sending low cost messages. This is a very good property of the indexing scheme and has not been pointed out in the literature before.

subject	predicate	object	object-class
$freq_s$	$freq_p$	$freq_o$	$freq_c$
dp_s	ds_p	ds_o	–
do_s	dp_p	dp_o	–

Fig. 4. Statistics kept at each peer

Creating statistics. Let us first introduce some new notation which is useful for keeping statistics for the sizes of the domain of the variables appearing in a query. We will denote by $ds_c(v)$ ($dp_c(v)$ and $do_c(v)$, respectively) the total number of distinct subject (predicate and object, respectively) values that exist in the triples stored in the network which contain value v as component c. For example, let tp be the triple pattern `(?x,ub:advisor,?y)`. Then, ds_p(`ub:advisor`) denotes the number of the distinct subject values in the triples with predicate `ub:advisor`, i.e., the size of the domain of variable `?x`. Similarly, do_p(`ub:advisor`) denotes the number of the distinct object values in the triples with predicate `ub:advisor`, i.e., the size of the domain of variable `?y`.

The following observation allows us to collect local statistics at each peer of a DHT. *Let v be the value of a bound subject of a triple pattern and p_v the peer responsible for key v. Then, peer p_v is capable of computing the exact frequency of v as a subject, ($freq_s(v)$), the exact number of the distinct predicate values with subject v ($dp_s(v)$), and the exact number of the distinct object values with subject v ($do_s(v)$) in the set of triples stored by looking* only *in its local database.* Given that peer p_v is responsible for key v, our indexing scheme forces all triples that contain v either as a subject, predicate or object to be stored at peer p_v. Therefore, peer p_v can retrieve from its local triple relation all triples that contain v as a subject and hence it can compute the occurrences of v as a subject in the set of all triples stored in the network, i.e., $freq_s(v)$. In addition, peer p_v can compute the number of the distinct predicate values and object values for subject v by projecting the triples that contain v as subject on the predicate and object attribute respectively. The same holds for a triple pattern's bound predicate or object. Following that, it is sufficient for each peer to create and maintain statistics of its *locally* stored RDF data only, and more precisely only of the triple components' values which are the *keys* that led to the peer.

For each triple component, a peer keeps the statistics shown in Fig. 4. A data structure which would keep the exact distribution of each triple component would require excessive memory space for very large amount of data. A commonly used method dating back from relational systems is estimating the frequency distribution of an attribute by creating *histograms* [16]. Given a space budget B for each statistical structure, each peer decides if the exact distribution can be kept in memory or an estimation of the distribution is required by creating a histogram. We use v-optimal-end-biased histograms [16] which we

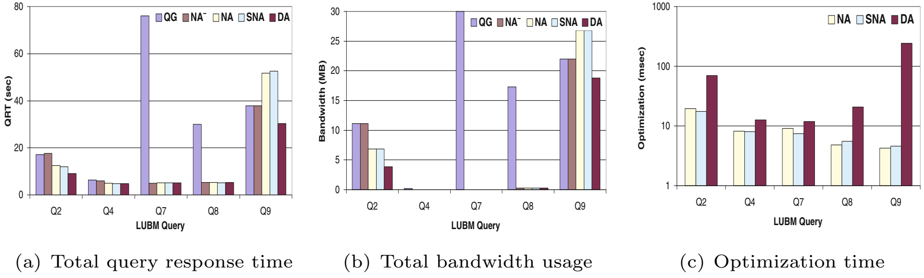
experimentally found to be more suitable. As shown in Fig. 4, we differentiate between objects of triples of the form `(s,rdf:type,o)`, which are classes, and objects of triples `(s,p,o)` with `p` $\neq$ `rdf:type` since we discovered that a more accurate estimation for the objects statistics can be achieved in this way.

Retrieving statistics. Whenever the query optimizer of peer x needs statistics for the selectivity estimation of one triple pattern or a conjunction of triple patterns, it sends a $\textsc{StatsReq}(v_i, c_i)$ message for each bound component value v_i appearing in the triple patterns to the peer responsible for key v_i specifying also the type c_i of the component value (i.e., subject, predicate, object or class). The peer that receives such a message retrieves the required statistics for value v_i from the corresponding statistical structure (depending on the value c_i) and sends them back to peer x. The time required to retrieve the statistics is negligible compared to the time required for evaluating a query, as we will see in the experimental section. The cost of sending these messages is very low since: (a) messages are small in size, (b) messages are sent in parallel, (c) each message requires only $O(logn)$ hops to reach the destination peer, and (d) the statistics at the destination peer are kept in main memory.

7 Experimental Evaluation

In this section, we present an experimental evaluation of our optimization techniques. All algorithms have been implemented as an extension to our prototype system Atlas. In the latest version of Atlas, we have adopted SQLite as the local database of each peer since the Berkeley DB included in the Bamboo implementation was inefficient. For our experiments, we used as a testbed both the PlanetLab network as well as a local shared cluster (`http://www.grid.tuc.gr/`). Although we have extensively tested our techniques on both testbeds, here we present results only from the cluster where we achieve much better performance. The cluster consists of 41 computing nodes, each one being a server blade machine with two processors at 2.6GHz and 4GB memory. We used 30 of these machines where we run up to 4 peers per machine, i.e., 120 peers in total.

For our evaluation, we use the Lehigh University benchmark (LUBM) [4]. In each experiment we first infer all triples and then store them in the network. We present results only for queries with more than 4 triple patterns so that the benefits of the proposed optimizations can be clearly demonstrated. We omit query Q12 since it always produces an empty result set and does not exhibit interesting results. All measurements are averaged over 10 runs using the geometric mean which is more resilient to outliers. In the following, QG denotes that the query graph is used to avoid Cartesian products but no other optimization is utilized. The naive algorithm using the bound-is-easier heuristic is denoted by NA^-, while the naive and semi-naive algorithm using the analytical estimation is denoted by NA and SNA, respectively. Finally, DA denotes the dynamic optimization algorithm.



(a) Total query response time (b) Total bandwidth usage (c) Optimization time

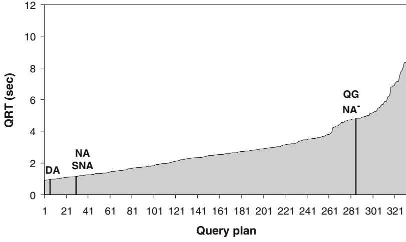
Fig. 5. Query optimization performance for LUBM-50

7.1 Comparing the Optimization Algorithms

In this section, we compare and evaluate the optimization algorithms described in Section 4. For this set of experiments, we store all the inferred triples of the LUBM-50 dataset ($9,437,221$ triples) in a network of 120 peers. Then, using each optimization algorithm, we run the queries. In all graphs of Fig. 5, the x-axis shows the LUBM queries while the y-axis depicts the metric of interest. Figure 5(a) shows the query response time (QRT) for the different LUBM queries. QRT is the total time required to answer a query and it also includes the time required by the query optimizer for determining a query plan (optimization time). Figure 5(b) shows the total bandwidth consumed during query evaluation.

Queries Q2 and Q9 consist of 6 triple patterns having only their predicates bound. In both queries, there exists a join among the last three triple patterns (in the order given by the benchmark) and the combination of all three triple patterns is the one that yields a small result set. DA finds a query plan that combines these three triple patterns earlier than the other algorithms. This results in producing smaller intermediate result sets, as it is also shown by the bandwidth consumption in Fig. 5(b), and thus results in better QRT. Although NA and SNA perform close to DA for query Q2, they fail to choose a good query plan for Q9 affecting both the QRT and the bandwidth consumption. At this point, we should note that QG and NA^- depend on the initial order of a query's triple patterns. For this reason, both algorithms choose a relatively good query plan for query Q9 since the order in which its triple patterns are given by the benchmark is a good one. Q4 is a star-shape query with all its triple patterns sharing the same subject variable, while only the first two triple patterns have two bound components. Therefore, since these two triple patterns are the more selective ones, all optimization algorithms choose the same query plan and perform identically in terms of both QRT and bandwidth. The same holds for query Q7 where QRT is significantly reduced when using either optimization technique compared to QG. Q8 is a query similar to Q7.

In Fig. 5(c), we show the total optimization time in msec on a logarithmic scale. For QG and NA^- the optimization overhead is negligible and is not shown in the graph. The optimization time contains the time for retrieving the required statistics from the network, the time for the selectivity estimation and the time



(a) LUBM Q2 query plan space

LUBM Query	Min QRT	Max QRT	DA QRT
Q2	0.91 s	10.06 s	0.96 s
Q4	2.54 s	17.39 s	2.89 s
Q7	1.53 s	16.62 s	1.55 s
Q8	2.88 s	10.20 s	3.06 s
Q9	3.30 s	64.06 s	4.41 s

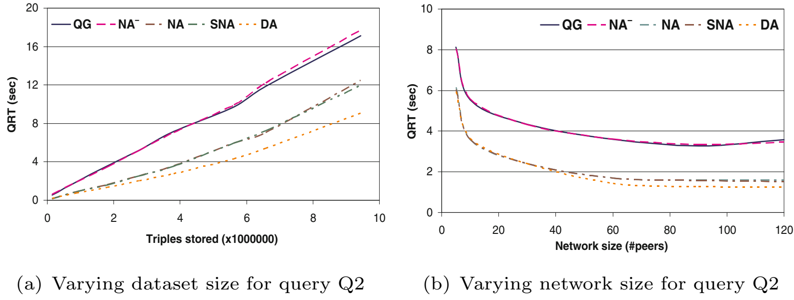
(b) QRT of LUBM query plans

Fig. 6. Exploring the query plan space for LUBM-10

spent by the optimization algorithm. As expected, DA spends more time than the other optimization algorithms since it runs at each query processing step. However, the optimization time is still one order of magnitude smaller than the time required by the query evaluation process. Therefore, although DA requires more time than the other optimization algorithms, the system manages to perform efficiently for all queries when DA is used. We observe similar results for the bandwidth consumed by the query optimizer. NA and SNA consume $\sim 2KB$ while DA consumes $\sim 7KB$, still one order of magnitude less than the bandwidth spent during query evaluation. We omit this graph due to space limitations.

7.2 Effectiveness of Query Optimization

In this section, we explore the query plan space of the LUBM queries to show how effective the optimization algorithms are. The size of the query plan space of a query consisting of N triple patterns is $N!$. Since query plans that involve Cartesian products are very inefficient to evaluate in a distributed environment, we consider only triple pattern permutations which do not produce any Cartesian product. In this experiment, we store the LUBM-10 dataset in a network of 120 peers and run all possible query plans for several LUBM queries. In Fig. 6(a), we depict the QRT of all possible query plans for query Q2 in ascending order. The query plan space of Q2 consists of 335 query plans which do not involve any Cartesian product. In this figure, we highlight the position of the query plans chosen by the different optimization algorithms. We observe that DA chooses one of the best query plans, while NA and SNA perform worse choosing the 27th best query plan. NA^{-} performs poorly choosing one the worst query plans. Similar results are observed for the other queries as well. In Fig. 6(b), we list the QRT for all queries of the best and the worst query plan together with the QRT when using DA. We observe that the QRT when using DA is very close to the QRT of the optimal query plan for all queries. Note that without the query plans that involve Cartesian products, the difference between the min and the max QRT of all queries is not very large.



(a) Varying dataset size for query Q2 (b) Varying network size for query Q2

Fig. 7. Varying dataset and network size

7.3 Varying the Dataset and Network Size

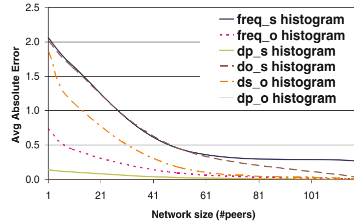
In these sets of experiments, we study the performance of our system when varying the number of triples stored in the network and the number of peers. We show results only for Q2 which involves a join among three triple patterns.

Figure 7(a) shows the behavior of our system using each optimization algorithm as the dataset stored in the network grows. In a network of 120 peers, we stored datasets from LUBM-1 to LUBM-50. Every time we measured the QRT of query Q2 using each optimization algorithm. As expected, QRT increases as the number of triples stored in the network grows. This is caused by two factors. Firstly, the local database of each peer grows and as a result local query processing becomes more time-consuming. Secondly, the result set of query Q2 varies as the dataset changes. For example, for LUBM-1 the result set is empty, while for LUBM-50 the result set contains 130 answers. This results in transferring larger intermediate result sets through the network which also affects the QRT of the query. Besides, this experiment brings forth an interesting conclusion regarding the optimization techniques. While query plans chosen by NA, SNA and DA perform similarly up to approximately $1.8M$ triples stored (i.e., LUBM-10), we observe that for bigger datasets the query plan chosen by DA outperforms the others. This shows that the system becomes more scalable with respect to the number of triples stored in the network when using DA. Similar results are observed for Q9, while for the rest queries all optimization algorithms choose the same query plan independently of the dataset size.

In the next set of experiments, we start networks of 5, 10, 30, 60, 90 and 120 peers and store the LUBM-10 dataset. We then run the queries using all optimization techniques. In Fig. 7(b), we show the QRT for Q2 as the network size increases. We observe that QRT improves significantly as the network size grows up to 60, while it remains almost the same for bigger network sizes. The decrease in the QRT for small networks is caused by the fact that the more peers join the network the less triples are stored in each peer's database and thus local processing load is reduced. The same result was observed in other queries where QRT either improved or remained unaffected as the number of peers increased.

Statistics	Min size	Max size	Avg size
histograms (x6)	580	580	580
predicate	44	448	71.47
object-class	44	288	61.80

(a) Size of statistics per peer (bytes)



(b) Histograms error

Fig. 8. Statistics

7.4 Statistics

We present measurements concerning the size of the statistics kept by each peer. We set a space budget of 500 bytes per statistical structure per peer leading to using histograms of 10 buckets only for values appearing in the subjects and objects of triples. Each statistical structure for the subjects and objects is kept in a separate histogram resulting in a total of six histograms per peer. For the predicates and object classes, we keep the exact distributions of their values. This is typical of a large DHT network where a peer is responsible for very few predicate or class values. Figure 8(a) shows the size of the generated statistics for each peer for the LUBM-50 dataset in a network of 120 peers. Histograms always occupy the same amount of space, while the exact statistics for predicate and object-class vary depending on the amount of values the peer is responsible for. The total statistics kept at each peer result in a total amount of memory of $4K$ in average which is negligible compared to today's powerful machines.

In order to show that it is sufficient to maintain local statistics at each peer and only for the values for which the peer is responsible, we have computed the average absolute error for each histogram for different network sizes ranging from 1 to 120 peers for LUBM-5. A network consisting of a single peer resembles a centralized system where a global histogram is created from all data stored. For every value v_i appearing in the dataset as a component c, we have measured its real frequency $freq_c(v_i)$ and the estimated frequency $\widehat{freq}_c(v_i)$ taken from the corresponding histogram of the peer responsible for value v_i. The absolute error for v_i equals to $e^{abs}(v_i) = |freq_c(v_i) - \widehat{freq}_c(v_i)|$. If N is the total number of distinct values of component c in the dataset, the average absolute error is computed as $\frac{1}{N}\sum_{i=1}^{N} e^{abs}(v_i)$. The same holds for the estimated number of distinct subjects, predicates and objects. Results for each statistical structure that is estimated by a histogram are shown in Fig. 8(b). We observe that as the number of peers increases the error drops significantly. This shows that the values of each triple component are independent and hence the more peers join the network (i.e., more histograms created), the better the estimation becomes.

7.5 Discussion

We have also experimented with different datasets using the SP^2B benchmark [19] as well as a real world dataset of the US Congress vote results presented in [26]. The results were similar to the ones observed using LUBM. For all datasets, DA consistently chooses a query plan close to the optimal regardless of the query type or dataset stored and without posing a significant overhead neither to the total time for answering the query nor to the bandwidth consumed. On the contrary, the static optimization methods are dependent on the type of the query and the dataset, which make them unsuitable in various cases (as shown earlier for query Q9). In addition, we have also tested indexing all possible combinations of the triples' components, as proposed in [11]. In this case, we have used histograms at each peer for combinations of triples' components as well. However, we did not observe any difference in the choice of the query plan and thus, showed results only with the triple indexing algorithm. This results from the nature of the LUBM queries which mostly involve bound predicates and object-classes for which we kept an exact distribution in both cases.

8 Related Work

Earlier works that consider SPARQL query processing on top of DHTs such as [1, 6, 7, 11] lack optimization techniques resulting in handling very small datasets (only thousands of triples). Another DHT-based system is UniStore where a triple-based model and a SPARQL-like query language is supported [10]. In UniStore, a cost-based optimizer is implemented which estimates the cost of physical operators in terms of the number of hops and messages required for each operator. The evaluation presented in [10] is conducted in PlanetLab and hence only small datasets are used. The work of [10] is complementary to ours and the two approaches could actually be combined by an appropriate cost model. Early works that studied query optimization in a distributed environment, although not a DHT, are [17, 24]. In [17], the authors present an engine for federated SPARQL databases and make use of query rewriting and cost-based optimization techniques. For the cost-based optimization, they use iterative dynamic programming but fail to estimate the selectivity of conjunctions of triple patterns and set it to a fixed value instead. Other works in the area of distributed SPARQL query processing are studied in [3, 5, 15]. However, these papers focus on distributed computing platforms based on powerful clusters and do not discuss any optimization techniques.

Finally, a lot of attention has been given to SPARQL query optimization in centralized environments [12, 13, 23]. In [23], the authors present a selectivity-based framework for optimizing SPARQL BGP queries. In RDF-3X [12], the authors propose two kinds of statistics for the selectivity estimation of the joins: specialized histograms which can handle both triple patterns and joins by leveraging the aggregated indexes built, and the computation of frequent join paths in the RDF graphs. In [13], the authors of RDF-3X go one step further to propose more accurate selectivity estimations by precomputing exact join cardinalities

for all possible choices of one or two constants in a triple pattern and materializing the results in additional indexes. This can be a very expensive operation in a distributed setting such as a DHT. Finally, a method for the cardinality estimation of SPARQL queries using a probability distribution is presented in [21].

9 Conclusions and Future Work

We studied the problem of distributed SPARQL query optimization on top of DHTs. We discussed the query optimization techniques we have developed in our system Atlas, and presented an experimental evaluation. Our current research is focused on the implementation and evaluation of algorithm SBV [11], which achieves a better load balancing, in the presence of the query optimization framework that we have developed.

References

1. Cai, M., Frank, M.R., Yan, B., MacGregor, R.M.: A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. Journal of Web Semantics (2004)
2. Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An Efficient SQL-based RDF Querying Scheme. In: VLDB 2005
3. Erling, O., Mikhailov, I.: Towards Web Scale RDF. In: SSWS 2008
4. Guo, Y., Pan, Z., Heflin, J.: LUBM: A Benchmark for OWL Knowledge Base Systems. Journal of Web Semantics (2005)
5. Harth, A., Umbrich, J., Hogan, A., Decker, S.: YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In: Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L.J.B., Golbeck, J., Mika, P., Maynard, D., Mizoguchi, R., Schreiber, G., Cudré-Mauroux, P. (eds.) ASWC 2007 and ISWC 2007. LNCS, vol. 4825, pp. 211–224. Springer, Heidelberg (2007)
6. Heine, F.: Scalable P2P based RDF Querying. In: InfoScale 2006
7. Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Magiridou, M., Miliaraki, I., Papadakis-Pesaresi, A.: Publishing, Discovering and Updating Semantic Grid Resources using DHTs. In: CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture 2006
8. Kaoudi, Z., Koubarakis, M., Kyzirakos, K., Miliaraki, I., Magiridou, M., Papadakis-Pesaresi, A.: Atlas: Storing, Updating and Querying RDF(S) Data on Top of DHTs. Journal of Web Semantics (System paper) (2010)
9. Kaoudi, Z., Miliaraki, I., Koubarakis, M.: RDFS Reasoning and Query Answering on Top of DHTs. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 499–516. Springer, Heidelberg (2008)
10. Karnstedt, M.: Query Processing in a DHT-Based Universal Storage - The World as a Peer-to-Peer Database. PhD thesis (2009)
11. Liarou, E., Idreos, S., Koubarakis, M.: Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In: Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L.M. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 399–413. Springer, Heidelberg (2006)

12. Neumann, T., Weikum, G.: RDF-3X: a RISC-style engine for RDF. In: VLDB 2008
13. Neumann, T., Weikum, G.: Scalable Join Processing on Very Large RDF Graphs. In: SIGMOD 2009
14. Ntarmos, N., Triantafillou, P., Weikum, G.: Distributed Hash Sketches: Scalable, Efficient, and Accurate Cardinality Estimation for Distributed Multisets. ACM TOCS (2009)
15. Owens, A., Seaborne, A., Gibbins, N., schraefel, m.: Clustered TDB: A Clustered Triple Store for Jena. Technical Report (2008) (Unpublished)
16. Poosala, V., Ioannidis, Y., Haas, P., Shekita, E.: Improved Histograms for Selectivity Estimation of Range Predicates. In: ACM SIGMOD 1996
17. Quilitz, B., Leser, U.: Querying Distributed RDF Data Sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 524–538. Springer, Heidelberg (2008)
18. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling Churn in a DHT. In: USENIX Annual Technical Conference 2004
19. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP^2Bench: A SPARQL Performance Benchmark. In: ICDE 2009
20. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access Path Selection in a Relational Database Management System. In: SIGMOD (1979)
21. Shironoshita, E.P., Ryan, M.T., Kabuka, M.R.: Cardinality Estimation for the Optimization of Queries on Ontologies. SIGMOD Record (2007)
22. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and Randomized Optimization for the Join Ordering Problem. VLDB Journal (1997)
23. Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: SPARQL Basic Graph Pattern Optimization using Selectivity Estimation. In: WWW 2008
24. Stuckenschmidt, H., Vdovjak, R., Broekstra, J., jan Houben, G., Eindhoven, T., Amersfoort, A.: Towards Distributed Processing of RDF Path Queries. Int. J. Web Eng. and Tech. (2005)
25. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. I and II. Computer Science Press, Rockville (1988)
26. Vidal, M.-E., Ruckhaus, E., Lampo, T., Martínez, A., Sierra, J., Polleres, A.: Efficiently Joining Group Patterns in SPARQL Queries. In: Aroyo, L., Antoniou, G., Hyvönen, E., ten Teije, A., Stuckenschmidt, H., Cabral, L., Tudorache, T. (eds.) ESWC 2010. LNCS, vol. 6088, pp. 228–242. Springer, Heidelberg (2010)
27. Weiss, C., Karras, P., Bernstein, A.: Hexastore: Sextuple Indexing for Semantic Web Data Management. In: VLDB 2008