Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks*

Erietta Liarou[1], Stratos Idreos[2], and Manolis Koubarakis[3]

[1] Technical University of Crete, Chania, Greece
[2] CWI, Amsterdam, The Netherlands
[3] National and Kapodistrian University of Athens, Athens, Greece

Abstract. We study the problem of evaluating conjunctive queries composed of triple patterns over RDF data stored in distributed hash tables. Our goal is to develop algorithms that scale to large amounts of RDF data, distribute the query processing load evenly and incur little network traffic. We present and evaluate two novel query processing algorithms with these possibly conflicting goals in mind. We discuss the various tradeoffs that occur in our setting through a detailed experimental evaluation of the proposed algorithms.

1 Introduction

Research at the frontiers of P2P networks and Semantic Web has recently received a lot of interest [23]. One of the most interesting open problems in this area is how to evaluate queries expressed in Semantic Web query languages (e.g., RDQL [22], RQL [17], SPARQL [21] or OWL-QL [10]) on top of P2P networks [9,4,19,20,25,18].

In this paper we study the problem of evaluating *conjunctive queries* composed of *triple patterns* on top of RDF data stored in distributed hash tables. *Distributed hash tables (DHTs)* are an important class of P2P networks that offer distributed hash table functionality, and allow one to develop scalable, robust and fault-tolerant distributed applications [2]. DHTs have recently been used for the distributed storage and retrieval of various kinds of data e.g., relational [12,14], textual [27], RDF [9] etc. Conjunctions of triple patterns are core constructs of some RDF query languages (e.g., RDQL [22] and SPARQL [21]) and used implicitly in all others (e.g., in the generalized path expressions of RQL [17]).

The contributions of this paper are the following. We present two novel algorithms for the evaluation of conjunctive RDF queries composed of triple patterns on top of the distributed hash table Chord [24]. This has been an open problem since the proposal of RDFPeers [9] where only *atomic* triple patterns and conjunctions of triple patterns with the *same* variable or constant subject and possibly different *constant* predicates have been studied. Extending these query

* This work was supported in part by the European Commission project Ontogrid (http://www.ontogrid.net/).

I. Cruz et al. (Eds.): ISWC 2006, LNCS 4273, pp. 399–413, 2006.
© Springer-Verlag Berlin Heidelberg 2006

classes considered by RDFPeers to full conjunctive queries is an important issue if we want to deal effectively with the full functionality of existing RDF query languages [22,17,21]. But notice that the resulting query class is more challenging than the ones considered in RDFPeers. In the terminology of relational databases: we now have to deal with arbitrary *selections, projections* and *joins* on a virtual ternary relation consisting of all triples.

The focus of our work is on the experimental evaluation of the proposed algorithms. We concentrate on three parameters that are critical in a distributed setting: *amount* of data stored in the network, *load distribution* and generated *network traffic*. Our algorithms are designed so that they involve in the query evaluation as many network nodes as possible, store as little date in the network as possible, and minimize the amount of network traffic they create. Trying to achieve all of these goals involves a tradeoff, and we demonstrate how we can sacrifice good load distribution to keep data storage and network traffic low and vice versa.

The rest of the paper is organized as follows. Section 2 presents a synopsis of the underlying assumptions regarding network architecture, data model and query language. Sections 3 and 4 present the alternative data indexing and query processing algorithms. Then, in Section 5, we present an optimization to further reduce the network traffic generated by the algorithms. In Section 6, we show a detailed experimental evaluation and comparison of our algorithms under various parameters that affect performance. Finally, Section 7 discusses related work, and Section 8 presents conclusions and future work directions.

2 System Model and Data Model

System model. We assume an overlay network where all nodes are *equal*, they run the same software and have the same rights and responsibilities. Each node n has a unique key (e.g., its public key), denoted by $key(n)$. Nodes are organized according to the Chord protocol [24] and are assumed to have synchronized clocks. This property is necessary for the time semantics we describe later on in this section. In practice, nodes will run a protocol such as NTP and achieve accuracies within few milliseconds [6]. Each data item i has a unique key, denoted by $key(i)$. Chord uses consistent hashing to map keys to identifiers. Each node and item is assigned an m-bit identifier, that should be large enough to avoid collisions. A cryptographic hash function, such as SHA-1 or MD5 is used: function $Hash(k)$ returns the m-bit identifier of key k. The identifier of a node n is denoted as $id(n)$ and is computed by $id(n) = Hash(key(n))$. Similarly, the identifier of an item i is denoted by $id(i)$ and is computed by $id(i) = Hash(key(i))$. Identifiers are ordered in an *identifier circle (ring)* modulo 2^m, i.e., from 0 to $2^m - 1$. Key k is assigned to the first node which is equal or follows $Hash(k)$ clockwise in the identifier space. This node is called the *successor* node of identifier $Hash(k)$ and is denoted by $Successor(Hash(k))$. We will often say that this node is *responsible* for key k. A query for locating the node responsible for a key k can be done in $O(\log N)$ steps with high probability [24], where N is the number of nodes in the network. Chord is described in more detail in [24].

The algorithms we describe in this paper use the API defined in [26,14,13]. This API provides two functionalities not given by the standard DHT protocols: (i) send a message to multiple nodes (multicast) and (ii) send d messages to d nodes where each node receives exactly one of these messages (this can be thought of as a variation of the multicast operation). Let us now briefly describe this API. Function $send(msg, id)$, where msg is a message and id is an identifier, delivers msg from any node to node $Successor(id)$ in $O(\log N)$ hops. Function $multiSend(msg, I)$, where I is a set of $d > 1$ identifiers $I_1, ..., I_d$, delivers msg to nodes $n_1, n_2, ..., n_d$ such that $n_j = Successor(I_j)$, where $1 < j \leq d$. This happens in $O(d \log N)$ hops. Function *multiSend()* can also be used as $multiSend(M, I)$, where M is a set of d messages and I is a set of d identifiers. In this case, for each I_j, message M_j is delivered to $Successor(I_j)$ in $O(d \log N)$ hops in total. A detailed description and evaluation of alternative ways to implement this API can be found in [13].

Data model. In the application scenarios we target, each network node is able to describe in RDF the resources that it wants to make available to the rest of the network, by creating and inserting metadata in the form of *RDF triples*. In addition, each node can *submit queries* that describe information that this node wants to receive all possible *answers* that are available at this time. We use a very simple concept of schema equivalent to the notion of a namespace. Thus, we do not deal with RDFS and the associated reasoning about classes and instances. Different schemas can co-exist but we do not support schema mappings. Each node uses some of the available schemas for its descriptions and queries.

We will use the standard RDF concept of a triple[1]. Let D be a countably infinite set of URIs and RDF literals. A triple is used to represent a statement about the application domain and is a formula of the form $(subject, predicate, object)$. The *subject* of a triple identifies the resource that the statement is about, the *predicate* identifies a property or a characteristic of the subject, while the *object* identifies the value of the property. The subject and predicate parts of a triple are URIs from D, while the object is a URI or a literal from D. For a triple t, we will use $subj(t)$, $pred(t)$ and $obj(t)$ to denote the string value of the subject, the predicate and the object of t respectively.

As in RDQL [22], a *triple pattern* is an expression of the form (s, p, o) where s and p are URIs or variables, and o is a URI, a literal or a variable. A *conjunctive query* q is a formula

$$?x_1, \ldots, ?x_n : (s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge \cdots \wedge (s_n, p_n, o_n)$$

where $?x_1, \ldots, ?x_n$ are variables, each (s_i, p_i, o_i) is a triple pattern, and each variable $?x_i$ appears in at least one triple pattern (s_i, p_i, o_i). Variables will always start with the '?' character. Variables $?x_1, \ldots, ?x_n$ will be called *answer variables* when we want to distinguish them from other variables of the query. A query will be called *atomic* if it consists of a single conjunct.

[1] `http://www.w3.org/RDF/`

Let us now define the concept of valuation (so we can talk about values that satisfy a query). Let V be a finite set of variables. A *valuation* v over V is a total function v from V to the set D. In the natural way, we extend a valuation v to be identity on D and to map triple patterns (s_i, p_i, o_i) to triples, and conjunctions of triple patterns to conjunctions of triple patterns.

We will find it useful to use various concepts from relational database theory in the presentation of our work. In particular, the operations of the relational algebra utilized in algorithm QC below follow the *unnamed perspective* of the relational model (i.e., tuples are elements of Cartesian products and co-ordinate numbers are used instead of attribute names) [5].

An *RDF database* is a set of triples. Let DB be an RDF database and q a conjunctive query $q_1 \wedge \cdots \wedge q_n$ where each q_i is a triple pattern. The *answer* to q over database DB consists of all *n-tuples* $(v(?x_1), \ldots, v(?x_n))$ where v is a valuation over the set of variables of q and $v(q_i) \in DB$ for each $i = 1, \ldots, n$.

In the algorithms we will describe below, each query q has a unique key, denoted by $key(q)$, that is created by concatenating an increasing number to the key of the node that posed q.

3 The QC Algorithm

Let us now describe our first query processing algorithm, the *query chain* algorithm (QC). The main characteristic of QC is that the query is evaluated by a chain of nodes. Intermediate results flow through the nodes of this chain and finally the last node in the chain delivers the result back to the node that submitted the query. We will first describe how triples are stored in the network and then how an incoming query is evaluated by QC.

Indexing a new triple. Assume a node x that wants to make a resource available to the rest of the network. Node x creates an RDF description d that characterizes this resource and publishes it. Since, we are not interested in a centralized solution, we do not store the whole description d to a single node. Instead, we choose to split d into triples and disperse it in the network, trying to distribute responsibility of storing descriptions and answering future conjunctive queries to several nodes. Each triple is handled separately and is indexed to three nodes. Let us explain the exact details for a triple $t = (s, p, o)$. Node x computes the index identifiers of t as follows: $I_1 = Hash(s)$, $I_2 = Hash(p)$ and $I_3 = Hash(o)$. These identifiers are used to locate the nodes r_1, r_2 and r_3, that will store t. In Chord terminology, these nodes are the successors of the relevant identifiers, e.g., $r_1 = Successor(I_1)$. Then, x uses the $multiSend()$ function to index t to these 3 nodes. Each node that receives a triple t stores it in its local *triple table* TT. In the discussion below, TT will be formally treated as a ternary relation (in the sense of the relational model).

Evaluating a query. Assume a node x that poses a conjunctive query q which consists of triple patterns $q_1, \ldots, q_k$. Each triple pattern of q will be evaluated by a (possibly) different node; these nodes form the *query chain* for q. The order we

use to evaluate the different triple patterns is crucial and we discuss the issues involved later on. Now, for simplicity, we assume that we first evaluate the first triple pattern, then the second and so on.

Query evaluation proceeds as follows. Node x determines the node that will evaluate triple pattern q_1 by using one of the constants in q_1. For example, if $q_1 = (?s_1, p_1, ?o_1)$ then x computes identifier $I_1 = Hash(pred(q_j))$ since the predicate part is the only constant part of q_j. This identifier is used to locate the node r_1 (the successor of I_1) that may have triples that satisfy q_1, since according to the way we index triples, all triples that have $pred(q_j)$ as their predicate will be stored in r_j. Thus, n sends the message $QEval(q, i, R, IP(x))$ to node r_1 where q is the query, i is the index of the triple pattern to be evaluated by node r_1, $IP(x)$ is the IP address of node x that posed the query, and R is the relation that will be used to accumulate triples that are *intermediate results* towards the computation of the answer to q. In this call, R receives its initial value (formally, the trivial relation $\{()\}$ i.e., the relation that consists of an empty tuple over an empty set of attributes).

In case that q_1 has multiple constants, x will heuristically prefer to use first the subject, then the object and finally the predicate to determine the node that will evaluate q_1. Intuitively, there will be more *distinct* subject or object values than *distinct* predicates values in an instance of a given schema. Thus, our decision help us to achieve a better distribution of the query processing load.

Local processing at each chain node. Assume now that a node n receives a message $QEval(q, i, R, IP(x))$. First, n evaluates the i-th triple pattern of q using its local triple table i.e., it computes the relation $L = \pi_X(\sigma_F(TT))$ where F is a selection condition and X is a (possibly empty) list of natural numbers between 1 and 3. F and X are formed in the natural way by taking into account the constants and variables of q_i e.g., if q_i is $(?s_i, p_i, o_i)$ then $L = \pi_1(\sigma_{2=p_i \wedge 3=o_i}(TT))$. Then, n computes a new relation with intermediate results $R' = \pi_Y(R \bowtie L)$ where Y is the (possibly empty) list of positive integers identifying columns of R and L that correspond to answer variables or variables with values that are needed in the rest of the query evaluation (i.e., variables appearing in a triple pattern q_j of q such that $j > i$). Note that the special case of $i = 1$ (when $R' = \pi_Y(L)$) is covered by the above formula for R', given the initial value $\{()\}$ of R. If R' is not the empty relation then n creates a message $QEval(q, i+1, R', IP(x))$ and sends it to the node that will evaluate triple pattern q_{i+1}. If R' is the empty relation then the computation stops and an empty answer is returned to node x.

In the case that $i = k$, the last triple pattern of q is evaluated. Then, n simply returns relation R' back to x using a message $Answer(q, R')$. Now R' is indeed a relation with arity equal to the number of answer variables and contains the answer to query q over the database of triples in the network.

In the current implementation, $R' = \pi_Y(R \bowtie \pi_X(\sigma_F(TT)))$ is computed as follows. For each tuple t of R, we first rewrite q_i by substituting variables of q_i by their corresponding values in R. Then, we use q_i to probe TT for matching triples. For each matching triple, the appropriate tuple of R' is computed on

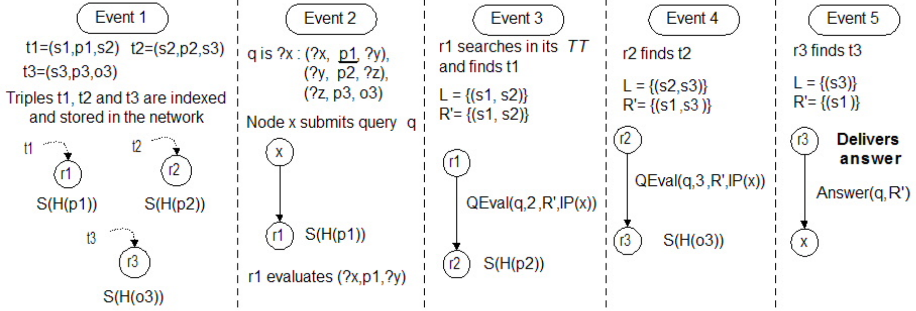


Fig. 1. The algorithm QC in operation

the fly. Access to TT can be made vary fast (essentially constant time) using hashing. In relational terminology, this is a nested loops join using a hash index for the inner relation TT. This is a good implementation strategy given that we expect a good evaluation order for the triple patterns of q to minimize the number of tuples in intermediate relation R (see relevant discussion at the end of this section).

Example. QC is shown in operation in Figure 1. Each *event* in this figure represents an event in the network, i.e., either the arrival of a new triple or the arrival of a new triple pattern. Events are drawn from left to right which represents the chronological order in which these events have happened. In each event, the figure shows the steps of the algorithm that take place due to this event. For readability, in each event we draw only the nodes that do something due to this event, i.e., store or search triples, evaluate a query etc. Finally, note that we use S for the function $Successor()$, H for the function $Hash()$ and we use comma to denote a conjunction between two triple patterns.

In Event 1, node n inserts three triples t_1, t_2 and t_3 in the network. In Event 2, node n submits a conjunctive query q that consists of three triple patterns. The figure shows how the query travels from node n to r_2, then to r_4 and finally to r_7, where the answer is computed and returned to n.

Order of nodes in a query chain. The order in which the different triple patterns of a query are evaluated is crucial, and affects network traffic, query processing load or any other resource that we try to optimize. For example, if we want to minimize message size for QC, we would like to put early in the query chain nodes that are responsible for triple patterns with *low selectivity.* Selectivity information can be made available to each node if statistics regarding the contents of TTs are available. Then, when a node n determines the next triple pattern q_{i+1} to be evaluated, n has enough statistical information to determine a good node to continue the query evaluation. The details of how to make our algorithms *adaptive* in the above sense are the subject of future work.

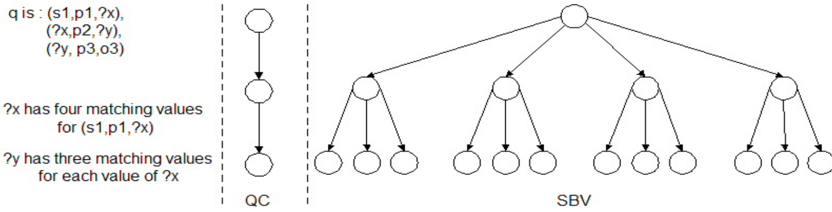


Fig. 2. Comparing the query chains in QC and SBV

4 The SBV Algorithm

Let us now present our second algorithm, the algorithm *spread by value* (SBV). SBV extends the ideas of QC to achieve a better distribution of the query processing load. It does not create a single chain for a query as QC does, but by exploiting the values of matching triples found while processing the query incrementally, it distributes the responsibility of evaluating a query to more nodes than QC. In other words, it is essentially constructing multiple chains for each query. A quick understanding of the difference between QC and SBV can be obtained from Figure 2. There, we draw for each algorithm, all the nodes that participate in query processing for a query q that consists of 3 triple patterns. QC creates a single chain that consists of only 3 nodes and query evaluation is carried out by these nodes only. On the contrary, SBV creates multiple chains which can collectively be seen as a tree. Now the query processing load for q is *spread* among the nodes of this tree. Each path in this tree is determined by the *values* used by triples that match the respective triple patterns at the different nodes (thus the name of the algorithm).

Indexing a new triple. Assume a new triple $t = (s, p, o)$. In SBV t will be stored at the successor nodes of the identifiers $Hash(s)$, $Hash(p)$, $Hash(o)$, $Hash(s + p)$, $Hash(s + o)$, $Hash(p + o)$ and $Hash(s + p + o)$. We will exploit these replicas of triple t to achieve a better query load distribution.

Evaluating a query. As in QC, the node that poses a new query q of the form $q_1 \wedge \cdots \wedge q_k$ sends q to a node r_1 that is able to evaluate the first triple pattern q_1. From this point on, the query plan produced by SBV is created *dynamically* by exploiting the values of the matching triples that nodes find at each step in order to achieve a better distribution of the query processing load. For example, r_1 will use the values for variables of q_1, that it will find in local triples matching q_1, to bind the variables of $q_2 \wedge \cdots \wedge q_k$ that are common with q_1 and produce a new set of queries that will jointly determine the answer to the original query q. Since we expect to have multiple matching values for the variables of q_1, we also expect to have *multiple next nodes* where the new queries will continue their evaluation. Thus, multiple chains of nodes take responsibility for the evaluation of q. The nodes at the leafs of these chains will deliver answers back to the node that submitted q. Our previous discussion on the order of nodes/triple patterns in a query chain is also valid for SBV. For simplicity, in the formal description

of SBV below, we assume again that the evaluation order is determined by the order that the triple patterns appear in the query.

To determine which node will evaluate a triple pattern in SBV, we use the constant parts of the triple pattern as in QC. The difference is that if there are multiple constants in a triple pattern, we use the *combination* of all constant parts. For example, if $q_j = (?s_j, p_j, o_j)$, then $I_j = Hash(pred(q_j) + obj(q_j))$ where the operator + denotes *concatenation* of string values. We use the concatenation of constant parts whenever possible, since the number of possible identifiers that can be created by a combination of constant parts is definitely higher and will allow us to achieve a better distribution of the query processing load.

Assume a node x that wants to submit a query q with set of answer variables V. x creates a message $Eval(q, V, u, IP(x))$, where u is the empty valuation. x computes the identifier of the node that will evaluate the first triple pattern and sends the message to it with the *send*() function in $O(\log N)$ hops.

When a node r receives a message $Eval(q, V, u, IP(x))$ where q is a query $q_1 \wedge \cdots \wedge q_n$ and $n > 1$, r searches its local TT for stored triples that satisfy triple pattern q_1. Assume m matching triples are found. For each satisfying triple t_i, there is a valuation v_i such that $t_i = v_i(q_1)$. For each v_i, r computes a new valuation $v'_i = u \cup v_i$ and a new query $q'_i \equiv v_i(q_2 \wedge \cdots \wedge q_n)$. Then r decides the node that will continue the algorithm with the evaluation of q'_i (as we described in the previous paragraph), and creates a new message $msg_i = Eval(q'_i, V, v'_i, IP(x))$ for that node. As a result, we have a set of at most m messages and r uses the *multiSend*() function to deliver them in $O(m \log N)$ hops. Each node that receives one of these messages reacts as described in this paragraph.

In the case that a node r receives a message $Eval(q, V, u, IP(x))$ where q consists of a single triple pattern q_1 (i.e., r is the last node in this query chain), then the evaluation of q finishes at r. Thus, r simply computes all triples t in TT and valuations v such that $t = v(q_n)$ and sends the set of all such valuations v back to node x that posed the original query in one hop (after projecting them on the answer variables of the initial query). These valuations are part of the answer to the query. This case covers the situation where $n = 1$ as well (i.e., q consists of a single conjunct). Figure 3 shows an example of SBV in operation.

5 Optimizing Network Traffic

In this section we introduce a new routing table, called *IP cache* (IPC) [14] that can be used by our algorithms to significantly reduce network traffic. In both our algorithms, the evaluation of a query goes through a number of nodes. The observation is that similar queries will follow a route with some nodes in common and we can exploit this information to decrease network traffic. Assume a node x_j that participates in the evaluation of a query q and needs to send a message to a "next" node x_{j+1} that costs $O(\log N)$ overlay hops. After the first time that node x_j has sent a message to node x_{j+1}, x_j can keep track of the IP address of x_{j+1} and use it in the future when the same query or a similar one obliges it to communicate with the same node. Then, x_j can send a message to x_{j+1} in

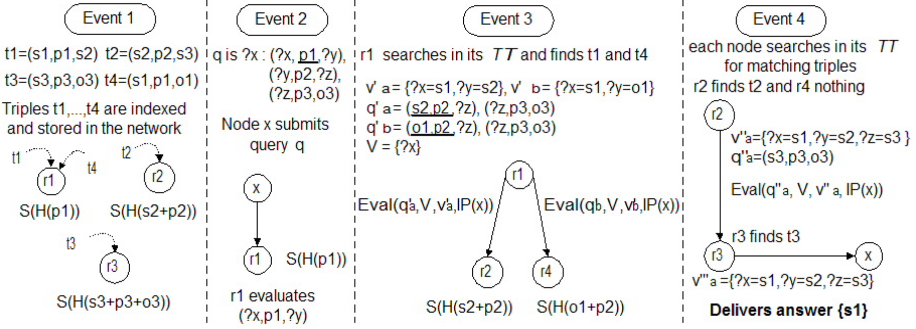


Fig. 3. The algorithm SBV in operation

just 1 hop instead of $O(\log N)$. The cost for the maintenance of the IPC is only local. As we will show in the experiments section, the use of IPCs significantly improves network traffic. Another effect of IPC, is that we reduce the *routing load* incurred by nodes in the network. The routing load of a node n is defined as the number of messages that n receives so as to forward them closer towards their destination, i.e., these are messages not sent to n but through n. Without using the IPC, each message that forwards intermediate results will pass through $O(\log N)$ nodes while with IPCs, it will go directly to the receiver node.

6 Experiments

In this section, we experimentally evaluate the algorithms presented in this paper. We implemented a simulator of Chord in Java on top of which we developed our algorithms. Our metrics are: (a) the amount of network traffic that is created and (b) how well the query processing load and storage load are distributed among the network nodes. Each metric will be carefully described in the relevant experiment. We create a uniform workload of queries and data triples. We synthetically create RDF triples and queries assuming an RDFS schema of the form shown in Figure 4, i.e., a balanced tree with depth d and branching factor k. We assume that each class has a set of k properties. Each property of a class C which is at level $l < d-1$ ranges over another class which belongs to level $l+1$. Each class of level $d-1$ has also k properties which have values that range over XSD datatypes. These data types are located at the last level d.

To create an RDF triple t, we first randomly choose a depth of the tree of our schema. Then, we randomly choose a class C_i among the classes of this depth. After that, we randomly choose an instance of C_i to be $subj(t)$, a property p of C_i to be $pred(t)$ and a value from the range of p to be $obj(t)$. If the range of the selected property p are instances of a class C_j that belongs to the next level, then $obj(t)$ is a resource, otherwise it is a literal.

For our experiments, we use conjunctive *path queries* of the following form:

$$?x : (?x, p_1, ?o_1) \wedge (?o_1, p_2, ?o_2) \wedge \cdots \wedge (?o_{n-1}, p_n, o_n)$$

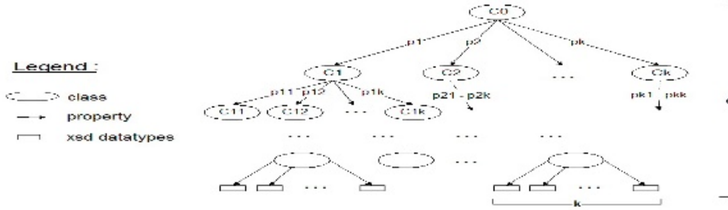


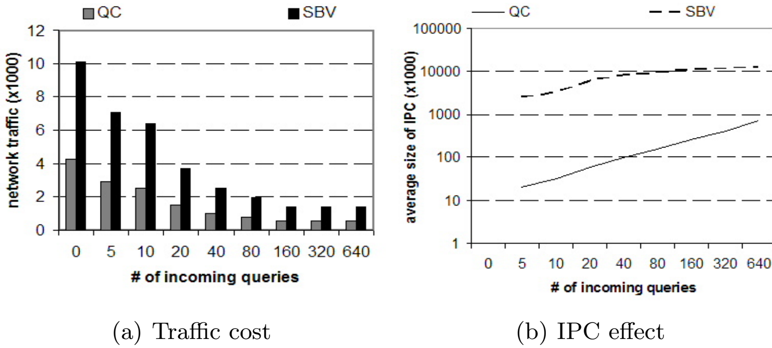
Fig. 4. The schema used in our experiments

In other words, we want to know the nodes in the graph $?x$ for which there is a path of length n to node o_1 labeled by predicates $p_1, \ldots, p_n$. Path queries are an important type of conjunctive queries for which database and query workloads over the schema of Figure 4 can be created easily. To create a query of this type, we randomly choose a property p_1 of class C_0. Property p_1 leads us to a class C_1 from the next level. Then we randomly choose a property p_2 of class C_1. This procedure is repeated until we create n triple patterns. For the last triple pattern, we also randomly choose a value (literal) from the range of p_n to be o_n.

Our experiments use the following parameters. The depth of our schema is $d = 4$. The number of instances of each class is 500, the number of properties that each one has is $k = 3$ while the a literal can take up to 200 different values. Finally, the number of triple patterns in each query we create is 5.

E1: Network traffic and IPC effect. This experiment provides a comparison of our algorithms in terms of the network traffic that they create. To estimate better the network traffic, we use weighted hops, i.e., each hope has as weight the amount of intermediate results that it carries. Furthermore, we investigate the effect of the IPC in each algorithm and the cost of this optimization. We set up this experiment as follows. We create a network of 10^4 nodes and install 10^4 triples. Then, in order to count how expensive it is to insert and evaluate a query, in terms of network traffic, we pose a set Q of 100 queries and calculate the average cost of answering them. In order to understand the effect of IPCs the experiment continues as follows. We train IPCs with a varying number of queries, starting from 5 queries up to 640. After each training phase, we insert the same set of queries Q and count (a) the average amount of network traffic that is created and (b) the average size of IPCs in the network. Each training phase, as we call it, has two effects: query insertions cause the algorithms to work so query chains are created and the rewritten queries are transferred through these chains, but also because of these forwarding actions IPCs are filled with information that can reduce the cost of a subsequent forwarding operation. After each training phase, we measure the cost of inserting a query in the network after all the queries inserted so far, by exploiting the content of IPCs.

In Figure 5(a) we show the network traffic that each algorithm creates. The point 0 on the x-axis has the maximum cost, since it represents the cost to insert the first query in the network. In this case all IPCs are empty and their use has no effect. Thus, this point reflects the cost of the algorithms if we do not use



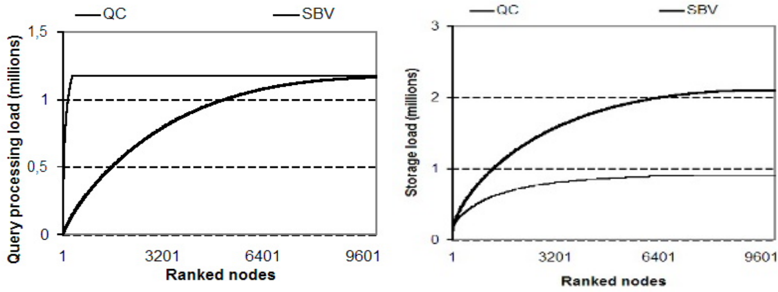
(a) Traffic cost (b) IPC effect

Fig. 5. (E1) Traffic cost and IPC effect as more queries are submitted

IPCs. However, in the next phases where IPCs have information that we can exploit, we see that the network traffic required to answer a query is decreased. For example, observe that after the last phase the cost of QC is 87% lower than it was at point 0. Another important observation is that QC causes less network traffic than SBV. In QC the nodes that participate in query chains are successors of a single value (of the predicate value for the queries we use in these experiments), so it is more possible that a query can use the IPC. SBV always creates more network traffic since the nodes that participate in query chains are successors of the combination of two values (a subject plus a predicate value). Since the combinations of these values are more then just a single one, it is less possible to use the IPC. QC is also cheaper at the point 0 on the x-axis since SBV has to sent the information though multiple chains.

In Figure 5(b) we show the average storage cost of the IPCs. Note, that here for readability we use a logarithmic scale for the y-axis. During the training phases, nodes fill their IPCs so we see that the size of IPC increases, as the number of submitted queries increases. Since even a small IPC size can significantly reduce network traffic, we can allow each node to fill its IPC as long as it can handle its size. The IPC cost in SBV is much more greater than in QC which happens again because SBV creates multiple chains for each query.

E2: Load distribution. In this experiment we compare the algorithms in terms of load distribution. We distinguish between two types of load: query processing load and storage load. The *query processing load* that a node n incurs is defined as the number of triple patterns that arrive to n and are compared against its locally stored triples. Note that for algorithm QC the comparison of a triple pattern with the triples stored in TT happens for each tuple of relation R when R' is computed. Thus, the query processing load of a node n in QC is equal to the number of tuples in R whenever a message $QEval()$ is received. The *storage load* of a node n is defined as the sum of triples that n stores locally. For this experiment, we create a network of 10^4 nodes where we insert $3 * 10^5$ triples. Then we insert 10^3 queries and after that, we count the query processing and the storage load of each node in the network.



(a) Cumulative query processing load (b) Cumulative storage load

Fig. 6. (E2) Query processing and storage load distribution

In Figure 6(a) we show the query processing load for both algorithms. On the x-axis of this graph, nodes are ranked starting from the node with the highest load. The y-axis represents the cumulative load, i.e, each point (a, b) in the graph represents the sum of load b for the a most loaded nodes. First, we observe that both algorithms create the same total query processing load in the network. SBV achieves to distribute the query processing load to a significantly higher portion of network nodes, i.e, in QC there are 306 nodes (out of 10^4) participating in query processing, while in SBV there are 9666 nodes. SBV achieves this nice distribution since it exploits the values used to create rewritten queries by forwarding the produced intermediate results to nodes that are the successors of a combination of two or three constant parts.

Finally, in Figure 6(b) we present the storage load distribution for both algorithms. As before, nodes are ranked starting from the node with the highest load while the y-axis represents the cumulative storage load. We observe that in QC the total storage load is less than in SBV. This happens because in QC we store each triple according to the values of its subject, its predicate and its object, while in SBV we also use the combinations per two and three of these values. Thus, in SBV a triple is indexed/stored four more times than in QC. The highest total storage load in the network is a price we have to pay for the better distribution of the query processing load in SBV.

Notice that our load balancing techniques are at the *application level*. Thus, they can be used together with DHT-level load balancing techniques, e.g., [16].

7 Related Work

The recent book [23] is an up-to-date collection of papers on work at the frontiers of P2P networks and Semantic Web. In the rest of this section, we only survey works that are closely related to our own.

In [9], Min Cai et al. studied the problem of evaluating RDF queries in a scalable distributed RDF repository, named RDFPeers. RDFPeers is implemented on top of MAAN [8], which extends the Chord protocols [24] to efficiently answer multi-attribute and range queries. [9] was the first work to consider RDF

queries on top of a DHT. The authors of [9] propose algorithms for evaluating triple pattern queries, range queries and conjunctive multi-predicate queries for the one-time query processing scenario. Furthermore, a simple replication algorithm is used to improve load distribution. Finally, [9] sketches some ideas regarding publish/subscribe scenarios in RDFPeers. In previous work [18], we have presented algorithms that go beyond the preliminary ideas of [9] regarding publish/subscribe for conjunctive multi-predicate queries.

The ideas in [9] have influenced the design of QC. However, we deal with the full class of conjunctive queries which is an extension of the class of conjunctive multi-predicate queries considered in [9]. In addition, we have presented the more advanced algorithm SBV which achieves efficient load distribution in a novel way.

The other interesting work in the area of RDF query processing on top of DHTs is GridVine [4]. GridVine is built on top of P-Grid [1] and can deal with the same kind of queries as RDFPeers. In addition, it has an original approach to global semantic interoperability by utilizing gossiping techniques [3].

Another distributed RDF repository that provides a general RDF-based metadata infrastructure for P2P applications is the Edutella system [19,20]. Edutella has two differences with our proposal: it is based on super-peers (while our proposal assumes that all nodes are equal) and concentrates on data integration issues (while we do not study this topic). Edutella uses HyperCup in its super-peer layer to achieve efficient routing of messages, but it does not consider issues such as the distribution of triples in the network to achieve scalability, load balancing etc. as in our approach.

The paper [25] is another interesting work on distributed RDF query processing focusing on the optimization of path queries over multiple sources.

Our research is also closely related with work on P2P databases based on the relational model [7,11,12,14]. Currently, one can distinguish two orthogonal research directions in this area: work that emphasizes semantic interoperability of peer databases [7,11] and work that attempts to push the capabilities of current database query processors to new large-scale Internet-wide applications by utilizing DHTs [12]. Our work can be categorized in the latter direction since it studies the processing of a subclass of conjunctive relational queries on top of DHTs. The only existing study of conjunctive relational queries on top of DHTs is [12] where join queries are studied. The ideas in this paper complement the ones in [12] and could also be used profitably in the relational case. This is an avenue that we plan to explore in future work together with extensions of our current results on continuous relational queries [14].

8 Conclusions and Future Work

In this paper we presented two novel algorithms for the distributed evaluation of conjunctive RDF queries composed of triple patterns. The algorithms manage to distribute the query processing load to a large part of the network while trying to minimize network traffic and keep storage cost low. The key idea is to decompose each conjunctive query to the triple patterns that it consists of,

and then handle each triple pattern separately at a different node. The first algorithm establishes a chain of nodes that carry out the query evaluation. The second algorithm dynamically exploits matching triples to determine the next node in the query plan and creates multiple node chains that carry out the query evaluation. As a result, it achieves a better distribution of the query processing load at the expense of extra network traffic and storage load in the network.

Our future work concentrates on extending our algorithms so that they can be adaptive to changes in the environment (e.g., changes in the data distribution), be able to handle skewed workloads efficiently, take into account network proximity etc. We also plan to extend our algorithms to deal with RDFS reasoning. Eventually, we want to support the complete functionality of languages such RDQL [22], RQL [17] and SPARQL [21]. The algorithms will be incorporated in our system Atlas [15] which is developed in the context of the Semantic Grid project OntoGrid[2] (Atlas currently implements QC).

References

[1] K. Aberer. P-Grid: A Self-Organizing Access Structure for P2P Information Systems. In *CoopIS '01*.

[2] K. Aberer, L. O. Alima, A. Ghodsi, S. Girdzijauskas, M. Hauswirth, and S. Haridi. The essence of P2P: A reference architecture for overlay networks. In *IEEE P2P 2005*.

[3] K. Aberer, P. Cudré-Mauroux, and M. Hauswirth. Start making sense: The chatty web approach for global semantic agreements. *Journal of Web Semantics*, 1(1), December 2003.

[4] K. Aberer, P. Cudre-Mauroux, M. Hauswirth, and T. V. Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *WWW '04*.

[5] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

[6] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The Price of Validity in Dynamic Networks. In *SIGMOD '04*.

[7] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision . In *WebDB '02*.

[8] M. Cai, M. Frank, and J. C. P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Grid '03*.

[9] M. Cai, M. R. Frank, B. Yan, and R. M. MacGregor. A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. *Journal of Web Semantics*, 2(2):109–130, December 2004.

[10] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL: A Language for Deductive Query Answering on the Semantic Web. *Journal of Web Semantics*, 2(1):19–29, December 2004.

[11] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What Can Peer-to-Peer Do for Databases, and Vice Versa? In *WebDB '01*.

[12] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB '03*.

[2] http://www.ontogrid.net

[13] S. Idreos. Distributed evaluation of continuous equi-join queries over large structured overlay networks. Master's thesis, 2005.

[14] S. Idreos, C. Tryfonopoulos, and M. Koubarakis. Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. In *ICDE '06*.

[15] Z. Kaoudi, I. Miliaraki, M. Magiridou, A. Papadakis-Pesaresi, and M. Koubarakis. Storing and querying RDF data in Atlas. In *Demo Papers ESWC '06*.

[16] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for Peer to Peer Systems. In *SPAA '04*.

[17] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *WWW '02*.

[18] E. Liarou, S. Idreos, and M. Koubarakis. Publish-Subscribe with RDF Data over Large Structured Overlay Networks. In *DBISP2P '05*.

[19] W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P Networking Infrastructure Based on RDF. In *WWW '02*.

[20] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Loser. Super-Peer-Based Routing and Clustering Strategies for RDF-Based Peer-To-Peer Networks. In *WWW '03*.

[21] E. Prud'hommeaux and A. Seaborn. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/, 2005.

[22] A. Seaborne. Rdql - a query language for RDF. W3C Member Submission, 2004.

[23] S. Staab and H. Stuckenschmidt. *Semantic Web and Peer-to-Peer*. Springer, 2006.

[24] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*.

[25] H. Stuckenschmidt, R. Vdovjak, J. Broekstra, and G.-J. Houben. Towards Distributed Processing of RDF Path Queries. *International Journal of Web Engineering and Technology*, 2(2/3):207–230, 2005.

[26] C. Tryfonopoulos, S. Idreos, and M. Koubarakis. LibraRing: An Architecture for Distributed Digital Libraries Based on DHTs. In *ECDL '05*.

[27] C. Tryfonopoulos, S. Idreos, and M. Koubarakis. Publish/Subscribe Functionality in IR Environments using Structured Overlay Networks. In *SIGIR '05*.