Publish/Subscribe with RDF Data over Large Structured Overlay Networks

Chapter · May 2007

DOI: 10.1007/978-3-540-71661-7_12 · Source: DBLP

3 authors, including:

Stratos Idreos
Harvard University
115 PUBLICATIONS **3,683** CITATIONS
SEE PROFILE

Manolis Koubarakis
National and Kapodistrian University of Athens
276 PUBLICATIONS **5,846** CITATIONS
SEE PROFILE

Publish/Subscribe with RDF Data over Large Structured Overlay Networks*

Erietta Liarou, Stratos Idreos, and Manolis Koubarakis

Department of Electronic and Computer Engineering
Technical University of Crete, GR73100 Chania, Greece
{erietta, sidraios, manolis}@intelligence.tuc.gr

Abstract. We study the problem of evaluating RDF queries over structured overlay networks. We consider the publish/subscribe scenario where nodes subscribe with long-standing queries and receive notifications whenever triples matching their queries are inserted in the network. In this paper we focus on conjunctive multi-predicate queries. We demonstrate that these queries are useful in various modern applications e.g., distributed digital libraries or Grid resource discovery. Conjunctive multi-predicate queries are hard to answer since multiple triples are necessary for their evaluation, and these triples will usually be inserted in the network asynchronously. We present and evaluate query processing algorithms that are scalable and distribute the query processing load evenly.

1 Introduction

Evaluating RDF queries in distributed environments is an open research problem. Semantic Web research can gain a lot from recent developments in the area of peer-to-peer (P2P) systems, and especially from results in the area of structured overlay networks. We discuss RDF query processing over a popular kind of such networks, called distributed hash tables (DHTs) [1]. DHT protocols allow nodes holding data items to self-organize and offer data lookup functionality in a provably efficient, scalable, fault-tolerant and adaptive way.

The problem of designing distributed algorithms to evaluate RDF queries over structured overlay networks has been considered by many papers so far e.g., [2–7]. We can distinguish two scenarios for query processing in these papers. In the *one-time query* case [5, 2, 3], a user poses a query like "give me all songs by Leonard Cohen" and the system replies with a set of answers/pointers to nodes that hold related resources. In the *publish/subscribe scenario* [4, 2, 8], a user subscribes with a continuous query like "notify me when a new song of Leonard Cohen becomes available" and receives notifications when matching resources become available.

We consider RDF queries in the style of RDQL [9] using triples as the atomic construct. Our long term research goal is to create a set of algorithms for the publich/subscribe scenario that will support all useful query types in languages such

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

as RDQL and RQL. In this paper we make a first step towards this direction by providing a set of algorithms that support the class of *conjunctive multi-predicate queries* and demonstrate that queries of this class are useful in applications. Conjunctive multi-predicate queries over a distributed DHT environment were first considered in [2] where algorithms for one-time query processing scenarios are proposed. Here we consider the publish/subscribe scenario for such queries over DHTs.

The contributions of this paper are the following. We propose two distributed algorithms for evaluating continuous conjunctive multi-predicate queries on top of DHTs. In our experiments we use Chord [10] as the underlying DHT due to its relative simplicity and widespread popularity. However, the implementation of our ideas which is underway, is *DHT-agnostic:* it will work with any DHT extended with the APIs we define. The case of conjunctive multi-predicate queries is an interesting one since more than one triples may be needed to answer a query. Since typically triples do not arrive in the network at the same time, the network should "remember" the queries that have been partially satisfied and create notifications only when all subqueries of a given query are satisfied. We introduce the notion of *query chains* to handle this problem. We argue that our algorithms are appropriate for *large networks* since their main emphasis is to *distribute the query processing load* to as many nodes as possible, while at the same time keeping the *network cost* in terms of overlay hops low. We experimentally evaluate and compare our algorithms in a simulated environment.

The organization of the paper is as follows. Section 2 describes Chord and our assumptions regarding the system and data model. Sections 3, 4 and 5 describe the two query processing algorithms. In Section 6 we experimentally evaluate the performance of our algorithms. Finally, Section 7 concludes the paper.

2 System model and data model

We assume an overlay network where all nodes are *equal*, as 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 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 [11] and achieve accuracies within few milliseconds. 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 as follows: $id(n) = Hash(key(n))$. Similarly the identifier of an item i is denoted as $id(i)$ and is computed as follows: $id(i) = Hash(key(i))$. 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 [10], where N is the number of nodes in the network. Chord is described in more detail in [10].

We use the API defined in [12, 13] for implementing pub/sub functionality on top of Chord. [12] deals with languages from Information Retrieval while [13] with two-way equi-join queries and there is no overlap of [12, 13] with the algorithms of this paper. Let us now shortly 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 $O(logN)$ hops. Moreover, 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 $d * O(logN)$ 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. For each I_j, message M_j is delivered to $Successor(I_j)$ in $d*O(logN)$ hops. A detailed description and evaluation of this API can be found in [12].

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 *triples*. In addition, each node can *subscribe with a continuous query* that describes information that this node wants to receive *notifications* for. 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 simple 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.

Each triple t has a time parameter called *published time*, denoted by $pubT(t)$, that represents the time that the triple is inserted into the network. Each query q has a unique key, denoted as $key(q)$, that is created by concatenating an increasing number to the key of the node that posed q. Each query q has a time parameter, called *subscription time*, denoted by $subscrT(q)$ that represents its creation time. Each subquery q_i of a query q is also assigned a subscription time $subscrT(q_i) = subscrT(q)$. A triple t can satisfy q_i iff $subscrT(q_i) \leq pubT(t)$, i.e., only triples that are inserted after a continuous query was subscribed can satisfy it. We will not have a complicated formal definition of notification as it might be appropriate for some applications.

We concentrate on the class of conjunctive multi-predicate queries. A conjunctive multi-predicate query q is a formula in the following form:

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

where $?s$ is a variable, $p_1, \ldots, p_n$ are URIs and $o_1, \ldots, o_n$ are variables, URIs or literals. $?x_1, \ldots, ?x_n$ are variables and $\{x_1, \ldots, x_n\} \subseteq \{s, o_1, \ldots, o_m\}$. Variables will always start with the '?' character as in [2]. The formulas $(?s, p_1, o_1)$,...,$(?s, p_n, o_n)$ will be called *subqueries* of q. A query will be called *atomic* if it consists of a single conjunct.

A *substitution* θ is a finite set of the form $\{?v_1/c_1, \ldots, ?v_n/c_n\}$ where each $?v_i$ is a distinct variable and each c_i is a URI or literal. Each constant c_i is

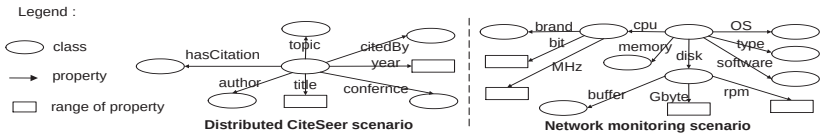


Fig. 1. Possible schemas for example applications

called a *binding* for $?v_i$. Note that we deal only with *ground* substitutions. Let q be a query and θ a variable substitution. Then $q\theta$ will denote the result of substituting each variable of q with its binding in θ.

A set of triples $T = \{t_1, \ldots, t_n\}$ *satisfies* a query $q = q_1 \wedge \ldots \wedge q_k$ with variable substitution θ, if for each $i = 1, \ldots, k$ there exists j, $1 \leq j \leq n$ such that triple t_j satisfies q_i with θ (i.e., $q_i\theta = t_j$) and $subscrT(q_i) \leq pubT(t_j)$. A triple t *satisfies* an atomic query q with variable substitution σ, if $q\sigma = t$ and $subscrT(q) \leq pubT(t)$.

Let T be an RDF database and q be a query in the above form. A substitution θ in variables $?x_1, ..., ?x_n$ is an *answer* to q if T satisfies q. A notification corresponding to a query q of the above form is just a substitution θ which is an answer to q.

Conjunctive multi-predicate queries over a distributed DHT environment were first considered in [2] for one-time query processing scenarios. Note that this class of queries allows join *only* on s (i.e., s is a subject *common to all* triples). Such queries can be used to express many interesting queries for P2P applications using RDF. For example, assume a distributed digital library that provides functionalities like those of CiteSeer. Library nodes could publish descriptions of academic literature in electronic format. The schema of the left graph of Figure 1 can be part of the schema used in such an application. Nodes can also subscribe with queries looking for publications with specific characteristics. A possible query could be: "Notify me when a paper by Smith is published that is related to P2P networks. List all citations in this paper". This is a conjunctive multi-predicate query that can possibly be expressed as follows:

$$?x, ?y : (?x, author, \text{“}Smith\text{”}) \wedge (?x, topic, \text{“}P2P\text{”}) \wedge (?x, citation, ?y)$$

It is well-known from systems such as EDUTELLA [5] that RDF is nicely suited for capturing digital library resource metadata. The fact that resource metadata may enter the network asychronously makes continuous query evaluation an incremental long-running activity (see Sections 3, 4 and 5). In reality, there will be applications where the metadata *about a specific resource* are all inserted in the network at *the same time* and applications where metadata are inserted in *steps*. For example, a digital library such as the ACM Digital Library might be expected to publish all metadata of a specific document (e.g., author, title, etc.) simultaneously. On the contrary, in the CiteSeer scenario, the system continuously crawls the web and collects information on Computer Science publications. In this case, as more details about a specific publication are created, previous CiteSeer entries will be updated.

Another example application where the class of queries studied is useful is Grid resource monitoring. In this application where computational resources (e.g., mainframes, personal computers, mobile devices, etc.) are connected in an overlay network. Users of this network would like to use cpu, memory, disk and other resources available in the overlay to carry out various computation- and data-intensive tasks. Part of the schema used in such a scenario could be the right graph of Figure 1. A continuous conjunctive multi-predicate query according to this schema might be "Notify me whenever a PC running Linux with the BLAST bioinformatics package installed, becomes available". This query can be expressed as follows:

$$?x : (?x, type, PC) \wedge (?x, OS, Linux) \wedge (?x, software, BLAST)$$

Similarly with CiteSeer, evaluating continuous queries for resource discovery in Grid environments needs data that might not be inserted in the system at the same time. Thus, algorithms have to "remember" previously inserted triples that partially satisfy a query. As new triples arrive, this memorized information is used to determine what queries have been fully satisfied. In general, applications where metadata is incrementally refined and updated seem to be prevalent in the Semantic Web and the Semantic Grid and can be nicely served by semi-structured data models like RDF and dynamic P2P networks.

3 A high-level view of our algorithms

In our algorithms, when a continuous query is submitted, it is *indexed* somewhere in the network and waits for triples to satisfy it. Each time a new triple is inserted, the network nodes cooperate to determine what queries are satisfied and create notifications. The case of conjunctive multi-predicate queries is an interesting one, since a single triple may *satisfy* a query q only *partially* by satisfying a subquery of q. In other words, more than one triples may be needed to answer a query. Moreover, since the appropriate triples do not necessarily arrive in the network at the same time, the network should "remember" the queries that have been partially satisfied in the past (e.g., by keeping intermediate results) and create notifications only when all subqueries of a given query are satisfied.

We could index queries to a globally known node or set of nodes, but this would eventually overload these nodes. In a P2P environment we want as many nodes as possible to contribute some of their resources (storage, cpu, bandwidth, etc.) for achieving the overall network functionality. The resource contribution of each node will obviously depend on its capabilities, its gains from participating in the network, etc. In this paper we make the simplifying assumption that all nodes are altruistic, with equivalent capabilities, and, thus, can contribute to query evaluation in identical ways.

Let us first consider an atomic query $q = (?s_1, p_1, ?o_1)$. We can simply assign q to the successor node x of $Hash(p_1)$ by using the constant part p_1 of the query. Triples that have predicate value equal to p_1 will be indexed to x too, where they

will meet q. Assume now the atomic query $q' = (?s_2, p_2, o_2)$. We can index q' either to node $x_1 = Successor(Hash(p_2))$ or to node $x_2 = Successor(Hash(o_2))$. We prefer the second option since intuitively there will be more object values than predicate values in an instance of a given schema, which will allow us to distribute queries to a greater number of nodes. Another solution is to index q' to the node $x_3 = Successor(Hash(p_2 + o_2))$. We use the operator + to denote the *concatenation* of string values. This is the best option because the possible combinations of predicate and object values will be greater than the number of object values alone, so this will lead to an even better distribution of queries.

The difficulty with arbitrary conjunctive multi-predicate queries is that they demand more than one conditions to be satisfied before the whole query can be satisfied. As an example, consider the query $q = q_1 \wedge q_2 \wedge q_3$. Our approach is to *split* the query to the subqueries that it consists of, and to index each subquery *separately*. Then, three usually different nodes will be responsible for query processing regarding q. Each one will be responsible for a single subquery of q, e.g., nodes r_1, r_2 and r_3 will be responsible for q_1, q_2 and q_3 respectively. These nodes will form the *query chain* of q, denoted by $chain(q)$. Each one of these nodes will monitor the satisfaction of only the subquery that it is responsible for. To determine the satisfaction of q, we have to allow some kind of communication between these three nodes. In this way, as triples arrive and satisfy a subquery e.g., in node r_1, r_1 will forward *partial results* of q to r_2. Node r_2 will forward partial results that also satisfy the second subquery to r_3 and r_3 will realize that the whole query is satisfied and create a notification.

The first algorithm that we present creates a single query chain for each conjunctive multi-predicate query while the second one creates multiple query chains for a single query to achieve a better query processing load distribution. The presented algorithms are useful for the evaluation of conjunctive multi-predicate queries. However, the general idea of these algorithms is the base of our research towards the creation of distributed algorithms that will also support more general query types, e.g., arbitrary queries. In the following sections we describe our algorithms in detail.

4 The single query chain algorithm

In this section we introduce the *single query chain* algorithm (SQC). The main characteristic of this algorithm is that for *each* query, it creates a *single* query chain. Let us assume a node n that wants to subscribe with the query $q = q_1 \wedge q_2 \wedge ... \wedge q_k$ where each subquery q_j is of the form $(?x, p_j, ?o_j)$ or $(?x, p_j, o_j)$. We will use functions $subj(q_j)$, $pred(q_j)$ and $obj(q_j)$ to denote the string value of the subject, the predicate and the object of subquery q_j respectively.

Indexing a query. Node n will index q by creating a query chain and assigning responsibility for q to the nodes in the chain as follows. For each subquery q_j, n creates a message INDEX-QUERY$(q_j, key(q), key(n))$ and computes an identifier I_j using the elements of q_j that are constant. If q_j is of the form $(?x, p_j, ?o_j)$, then $I_j = Hash(pred(q_j))$, while if it is of the form $(?x, p_j, o_j)$,

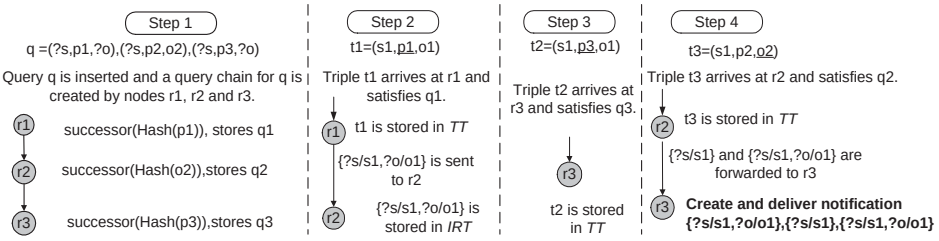


Fig. 2. The algorithm SQC in operation

then $I_j = Hash(obj(q_j))$. The identifier I_j will lead to the node that will be responsible for subquery q_j. In this way, a set M of k messages is created and a set I of k identifiers. The successors of those identifiers are called the *responsible* nodes for each subquery of q and form the query chain of q. Node n calls the function *multiSend*(M, I) to index the query with complexity $k{*}O(logN)$ overlay hops. The *multiSend()* function sorts I in the clockwise direction starting from $id(n)$ so the query chain that will be created will require the minimum overlay hops when forwarding intermediate results [12].

Each node r that receives an INDEX-QUERY$(q_j, key(q), key(n))$ message stores q_j in its local *query table* (QT) along with $key(q)$, $key(n)$, and two other parameters: $next(q_j)$ and $position(q)$. Parameter $next(q_j)$ will be used by r to reach the next node in the chain when needed, i.e., $next(q_j)$ is the identifier of the next node. Parameter $position(q)$ is used to show the position of r in $chain(q)$. *position* takes the value $first$ or $last$ if r is first or last in $chain(q)$ respectively. Otherwise, *position* takes the value *middle*. The construction of query chains in SQC is shown graphically in Figure 2 through an example.

Indexing a new triple. A new triple has to meet all relevant queries. Since subqueries are indexed either according to their predicate or their object value, a new triple $t = (s, p, o)$ has to reach both $Successor(Hash(p))$ and $Successor(Hash(o))$ for SQC to be complete. Thus, a node that inserts a new triple t will use function *multiSend*(msg, F), with msg =INDEX-TRIPLE$(t, key(n))$ and $F = \{Hash(p), Hash(o)\}$, to index t in $2 * O(logN)$ hops.

Forwarding intermediate results when new triples arrive. Let us now discuss how a node reacts upon receiving a new triple t. The node stores t in its local *triple table* (TT) and searches its QT for matching subqueries. We will first discuss what happens if this node is *first* in the query chain of a query q. For simplicity we assume that the nodes of the query chain are ordered as $r_1, \ldots, r_k$ and are responsible for subqueries $q_1, \ldots, q_k$ respectively. If t satisfies q_1 with substitution θ then a message msg=EXTEND-MATCHING $(\{q_1\}, \theta, key(q))$ is created and forwarded to the next node in $chain(q)$ with function $send(msg, next(q_j))$. Otherwise, triple t is ignored and there is nothing to be done.

If a message EXTEND-MATCHING $(\{q_1, \ldots, q_{j-1}\}, \theta, key(q))$ arrives at a node r_j in the middle of a query chain for some query q, then r_j tries to find out if the message can be forwarded further in the query chain. This can happen only if r_j

is storing triples that satisfy the subquery q_j of q that r_j is responsible for. Thus, r_j searches its local TT for such triples. If there is a triple t' and variable substitution σ such that $q_j\theta\sigma = t'$, then the list of satisfied subqueries $\{q_1, \ldots, q_{j-1}\}$ can be extended with q_j and the next node in the chain should be notified with a message EXTEND-MATCHING $(\{q_1, \ldots, q_j\}, \theta\sigma, key(q))$. Furthermore, r_j stores $\{q_j\theta\sigma\}$ locally in its *intermediate results table* (IRT) which is necessary when triples arrive directly to r_j (see below).

Let us now discuss what happens when a node r_{j+1} in the middle of the query chain of a query q, receives a new triple t''. t'' will be stored in the local TT and r_{j+1} will search locally for satisfied subqueries. Assume that t'' satisfies a subquery q_{j+1} of query q with variable substitution λ, so that $q_{j+1}\lambda = t''$. The difference with the case of being first in $chain(q)$ is that r_{j+1} will not forward t'' to the next node unless the previous node in $chain(q)$ has already sent appropriate intermediate results. Thus, r_{j+1} will search its IRT for partially satisfied subqueries of q. If $\{q_1, \ldots, q_j\}$ such subqueries exist, a message EXTEND-MATCHING $(\{q_1, \ldots, q_j, q_{j+1}\}, \theta\sigma\lambda, key(q))$ will be created and forwarded to the next node in $chain(q)$ as in the previous paragraph.

When a node that is at the end of a query chain receives a message EXTEND-MATCHING $(\{q_1, \ldots, q_j, q_{j+1}\}, \theta\sigma\lambda, key(q))$, it will search its IRT for partially satisfied subqueries as in the previous paragraph, but then instead of forwarding intermediate results (there are no more nodes in the chain), it will use the key of the node that posed the query to deliver any notifications.

An example with SQC in operation is shown in Figure 2. Events take place from left to right, i.e., initially query q is indexed and then triples arrive. For readability reasons, only the steps that affect query q are shown.

Grouping queries. Since a large number of subqueries are expected to be similar, i.e., some of their components are identical, they are *grouped* together at each node. For example, all subqueries that have been indexed to a node r using predicate p will be satisfied when a triple with predicate p arrives (since the subject and object are variables), so r can locally store these subqueries as a group, and check their satisfaction in one step when such a triple arrives. In addition, when nodes send messages EXTEND-MATCHING (), subqueries with the same parameter *next* are grouped so that these results are delivered with a single message to reduce network traffic.

Links. Each node in a chain will contact more than once its next node, so nodes can maintain *pointers* (the IP addresses) to their next nodes for efficiency. This happens with a hash table based local data structure, called *query chain routing table* (QCRT). Thus, intermediate results are forwarded in a single hop.

5 The multiple query chains algorithm

In this section we present the *multiple query chains algorithm* (MQC). With this algorithm we extend the ideas of SQC to achieve a better distribution of the query processing load. MQC exploits the values of incoming triples to distribute the responsibility of evaluating a query to more nodes than SQC. More precisely,

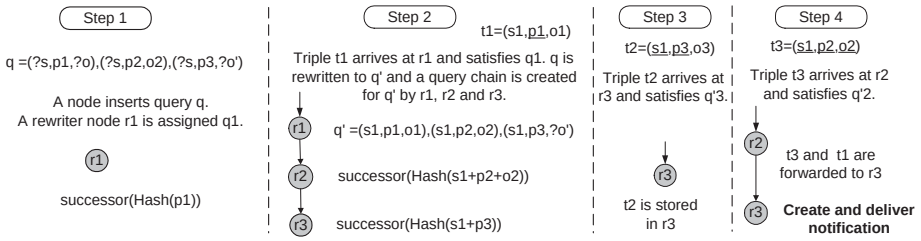


Fig. 3. The algorithm MQC in operation

instead of creating a single query chain for a query q at the time that q is inserted, MQC indexes q to a single node r according to one of q's subqueries. Then, when a triple satisfying this subquery arrives at r, the value of its *subject* is used to *rewrite* q. For each *different* rewritten query derived from q, a *different* query chain is created. Thus, in SQC each subquery of a query q is assigned to a single node, while in MQC rewritten instances of each subquery are assigned to multiple nodes, namely to as many nodes as the distinct subject values in the arriving triples. In addition, MQC combines the known parts of a subquery to index it in order to achieve better distribution as discussed in Section 3.

Indexing a query. Assume a node n that wants to subscribe with the query $q = q_1 \wedge q_2 \wedge ... \wedge q_k$ that consists of k subqueries of the form $(?x, p_j, ?o_j)$ or $(?x, p_j, o_j)$. First, a subquery q_j of q is selected and q is indexed to a node corresponding to q_j. This node is $r = Successor(Hash(pred(q_j) + obj(q_j)))$ if both $pred(q_j)$ and $obj(q_j)$ are constant, or $r = Successor(Hash(pred(q_j)))$ if only $pred(q_j)$ is constant. We call node r the *rewriter* of q. This terminology comes from [13]. Later on, we will discuss good ways to choose a rewriter but at the moment we can assume that this is a random choice. The rewriter stores q in its local QT and waits for triples that satisfy q_j. Since the query is indexed to a single node, the cost is $O(logN)$ overlay hops. Each query has one rewriter, while all queries with the same indexed part have the same rewriter.

Indexing a triple. Since a query might be indexed using a combination of the constant parts of a subquery, we need a new tuple $t = (s, p, o)$ to reach the successor nodes of identifiers $I_1 = Hash(p)$ and $I_2 = Hash(p + o)$. In addition, since queries are rewritten according to their subject values (as we will see below) we also need new triples to reach the successor nodes of the identifiers $I_3 = Hash(s + p)$ and $I_4 = Hash(s + p + o)$. Thus, a node n_1 that inserts a new triple t will use function $multiSend(msg, I)$ to index t to these 4 nodes in $4{*}O(logN)$ hops, where $msg =$INDEX-TRIPLE$(t, key(n_1))$ and $I = \{I_1, I_2, I_3, I_4\}$.

Receiving a new triple. Let us now discuss what happens when a new triple t arrives at a rewriter node r. r checks if its QT contains any query q with a subquery q_j satisfied by t. For each such query q and subquery q_j, r does the following. It *rewrites* the formula $q_1 \wedge \ldots \wedge q_{j-1} \wedge q_{j+1} \wedge \ldots \wedge q_k$ by replacing the subject variable in each subquery with $subj(t)$ to arrive at a new query q'. Then, r uses $subj(t)$ to determine the nodes that participate in the query

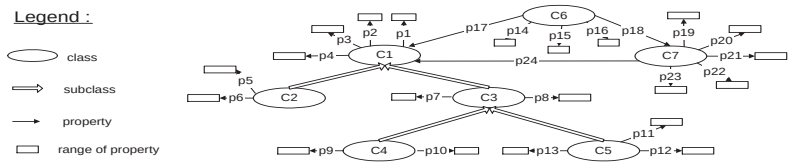


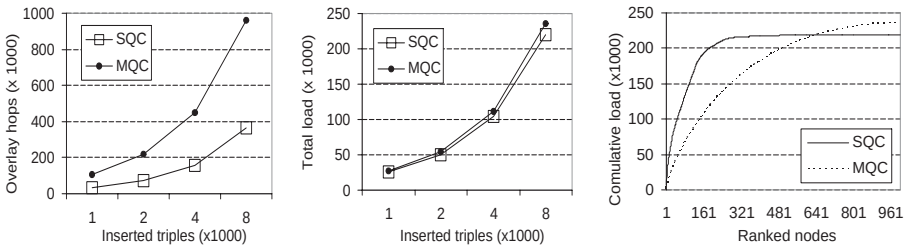
Fig. 4. The schema used in our experiments

chain for q'. If this is the first time that q has been satisfied in r by a triple with $subj(t)$, then there is no chain yet for q and this subject value. *Multiple* chains are created for q and different subject values, as triples arrive. In order to create a query chain for a rewritten query, a rewriter node r performs a similar procedure with the one that a query node performs upon indexing a query in SQC only that this time the first node of the chain is already known, namely it is node r. In addition, instead of calculating the index identifier of each subquery according to predicate or object, the index identifier is calculated according to the concatenated string of $subj(t)$ with the predicate or the predicate/object combination of each subquery. As in SQC, the object option is preferred if the object is a constant. Also, index identifiers are sorted according to their distance from the identifier of r to minimize network traffic. From there on, query chains work exactly as in SQC. Each node is responsible for one of the subqueries in the rewritten query. Intermediate results flow in the chain as in SQC while notifications are created by the last node in the chain.

An example with MQC in operation is shown in Figure 3. Notice that a query chain is created in step 2 after the query is rewritten due to a new triple.

6 Experiments

In this section we experimentally evaluate our algorithms. We implemented a simulator of Chord in Java, on top of which we developed our algorithms. We synthetically create RDF triples and queries assuming the RDFS schema of Figure 4. Since our algorithms do not do RDFS reasoning, subclass links in Figure 4 are used to propagate instantiation links and make all class information explicit. We assume a set of 1000 subject values and randomly assign each subject to a class. We also assume a set of 1000 object values and randomly assign each one to the range of a property. To create an RDF triple t, we first randomly choose a class C. Then we randomly choose an instance of C to be $subj(t)$, a property p of C or of the superclasses of C to be $pred(t)$ and a value from the range of p to be $obj(t)$. We use conjunctive multi-predicate queries with three subqueries. To create a query of this type, we first randomly choose a class that the query will refer to. Then, we randomly choose three distinct properties of this class to be the predicates of the three subqueries. All subqueries have the same subject variable while the object parameter of each subquery can be a constant value or a variable. When an object is constant, we randomly choose a value that belongs to the specific property chosen for this subquery.



(a) Overlay hops (b) Total load (c) Cumulative load (T=8K)

Fig. 5. Evaluating continuous conjunctive multi-predicate queries

We present what happens while increasing the total number of triples in the network. Our metrics are (a) the number of overlay hops needed to insert a number of triples, create and deliver notifications for all matching queries, (b) the total query processing load generated in the network and (c) the distribution of this load. The *query processing load* that a node incurs is defined as the sum of the number of triples that this node receives so as to check if locally stored queries are satisfied plus the number of subqueries that have to be compared against the triples locally stored in this node.

We design our experiment as follows. We create a network of 10^3 nodes and install 10^4 queries. Then, we insert $T = 1K$ triples and we evaluate the metrics described above. The last step is repeated three more times; each time we double T to reach 8K triples.

In Figure 5(a) we show the number of hops needed to insert a number of triples and evaluate all indexed queries. SQC outperforms MQC approximately by a factor of three. This is due to the fact that MQC creates more than one query chains for each query, which means that when nodes in SQC can use QCRTs, nodes in MQC have to create new chains and forward partial results using the Chord infrastructure or in other words in SQC nodes can train their QCRTs more quickly (there are less possible values). For both algorithms network traffic is linearly increased with the number of incoming triples. In addition, experiments where QCRTs are not used showed that MQC has similar performance with SQC (higher by a factor of 4 of the SQC performance with QCRT).

In Figure 5(b) we show the total load created by a number of incoming triples. We observe that the load increases linearly with the number of incoming triples. MQC creates a slightly higher load because more nodes have to be contacted and process messages. In Figure 5(c) we present the cumulative query processing load after 8K triples have been inserted. On the x-axis, nodes are ranked starting from the node with the highest query processing 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. We observe that although MQC reaches a slightly higher total load, it achieves to distribute this load to a significantly

higher portion of network nodes, i.e., in MQC there are 850 nodes (out of 1000) participating in query processing, while in SQC there are only 250 nodes.

MQC manages to fulfill our goals for a better load distribution which comes with a higher cost in total network traffic, as it is shown in Figure 5(a). However, this extra network traffic is suffered by *more nodes* (that have to create and forward the extra messages) in MQC. In a longer version of this paper, various experiments are underway that explore how other parameters (i.e., larger network sizes, increasing numbers of indexed queries, skewed distributions etc.) affect the performance of the algorithms.

7 Conclusions

We deal with the problem of evaluating RDF queries over DHTs. We proposed novel algorithms for resolving continuous conjunctive multi-predicate queries with emphasis on distributing load and keeping network traffic low.

References

[1] Balakrishnan, H., Kaashoek, M.F., Karger, D.R., Morris, R., Stoica, I.: Looking up data in P2P systems. CACM **46** (2003) 43–48

[2] Cai, M., Frank, M., Pan, B., MacGregor, R.: A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. J. Web Sem. **2** (2004)

[3] Aberer, K., Cudré-Mauroux, P., Hauswirth, M., Pelt, T.V.: GridVine: Building Internet-Scale Semantic Overlay Networks. (In: ISWC '04)

[4] Chirita, P.A., Idreos, S., Koubarakis, M., Nejdl, W.: Publish/Subscribe for RDF-based P2P Networks. (In: ESWC '04)

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

[6] Nejdl, W., Wolpers, M., Siberski, W., Schmitz, C., Schlosser, M.T., Brunkhorst, I., Löser, A.: Super-peer-based routing strategies for RDF-based peer-to-peer networks. J. Web Sem. **1** (2004) 177–186

[7] Kokkinidis, G., Christophides, V.: Semantic query routing and processing in P2P database systems: The ICS-FORTH SQPeer middleware. (In: P2P&DB '04)

[8] Chirita, P.A., Idreos, S., Koubarakis, M., Nejdl, W.: Designing Semantic Publish/Subscribe Networks using Super-Peers. In: Semantic Web and Peer-to-Peer. Springer Verlag (Forthcoming)

[9] Miller, L., Seaborne, A., Reggiori, A.: Three implementations of SquishQL, a simple RDF query language. (In: ISWC '02)

[10] Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable P2P Lookup Service for Internet Applications. (In: SIGCOMM '01)

[11] Bawa, M., Gionis, A., Garcia-Molina, H., Motwani, R.: The Price of Validity in Dynamic Networks. (In: SIGMOD '04)

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

[13] Idreos, S., Tryfonopoulos, C., Koubarakis, M.: Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. (In: ICDE '06)