# Continuous RDF Query Processing over DHTs*

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

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

**Abstract.** We study the continuous evaluation of conjunctive triple pattern queries over RDF data stored in distributed hash tables. In a continuous query scenario network nodes subscribe with long-standing queries and receive answers whenever RDF triples satisfying their queries are published. We present two novel query processing algorithms for this scenario and analyze their properties formally. Our performance goal is to have algorithms that scale to large amounts of RDF data, distribute the storage and query processing load evenly and incur as little network traffic as possible. We discuss the various performance tradeoffs that occur through a detailed experimental evaluation of the proposed algorithms.

## 1 Introduction

Continuous querying of RDF data has been studied only by a few Semantic Web researchers, although it is an important component of many Semantic Web applications [6, 5, 15, 14, 4, 13]. In a continuous query scenario, users post long-standing queries expressing their interests and are notified whenever new data matching their queries become available (e.g., "Notify me when a new article by John Smith becomes available"). Prominent examples of applications based on continuous querying of RDF data are the following: alert systems for Web resource management systems with resources annotated by RDF metadata (e.g., e-learning systems like ELENA [19], semantic blogging systems [10], RSS feeds etc.), notification mechanisms for Semantic Grid software [7], data coordination in heterogeneous P2P databases [2] based on RDF and so on.

In this work, we study the evaluation of *continuous conjunctive queries* composed of *triple patterns* over RDF data stored in *distributed hash tables (DHTs)*. Conjunctions of triple patterns are core constructs of some RDF query languages (e.g., RDQL [18] and SPARQL [16]) and used implicitly in all others (e.g., in the generalized path expressions of RQL [11]). 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 [1]. DHTs have recently been used for instantaneous querying of RDF data by [4, 12] and continuous querying of RDF data by [4, 13]. Unfortunately, both papers [4, 13] use a rather limited query language allowing only *atomic* triple patterns or conjunctions of triple patterns with the *same* variable or constant subject and possibly different *constant* predicates (the so-called conjunctive multi-predicate queries). Thus, the problem of evaluating arbitrary continuous conjunctive queries composed of triple patterns is left open by [4, 13].

* E. Liarou and M. Koubarakis were supported in part by the European project Ontogrid.

In this paper, we solve this problem by presenting two novel algorithms (the *continuous query chain* algorithm - CQC and the *continuous spread-by-value* algorithm - CSBV) for the continuous evaluation of conjunctive triple pattern queries on top of DHTs. The core ideas of CQC and CSBV come from the algorithms QC and SBV of [12] that solve the corresponding query processing problem for one-time queries. The first contribution of the present paper is to show that the main ideas of QC and SBV are powerful enough to be applicable in a continuous query setting, and to develop the required machinery of the new algorithms CQC and CSBV. The second contribution is a detailed experimental evaluation of CQC and CSBV. We focus on two critical parameters in a distributed setting; load distribution and network traffic. Both algorithms are designed in such a way that they try to involve as many nodes as possible in the query processing procedure, while taking into account the amount of traffic they create. This involves a tradeoff and we clearly demonstrate that one algorithm can sacrifice some nice load distribution properties to keep network traffic at a lower level or vice versa.

For a continuous conjunctive query of $k$ triple patterns, we may need $k$ different triples and each triple may participate in more than one answer sets for a given query. These triples may arrive asynchronously. This means that when a triple $t$ arrives and we detect that it can be used to generate an answer for a query $q$, we should "remember" this information to use it in the future, when the rest of the triples that are necessary to create an answer for $q$ (together with $t$) arrive. To avoid recomputing queries each time new matching data arrive, we keep a distributed state of already received triples. We achieve this by careful assignment of intermediate results to the proper nodes (where future matching data might arrive) and by rewriting queries into ones with fewer conjuncts that reflect the fact that certain triples have arrived. Another key point is that our algorithms split the responsibility of handling events at the triple pattern level. When a query $q$ is inserted in the network, it is not assigned to a single node. Instead, different nodes are responsible for different triple patterns of $q$ which allows for better load distribution.

We present a large number of experimental results. For example, we study the effect of varying the number of indexed queries. The larger the number of continuous queries indexed in the network waiting for data, the harder it is to find answers each time new data arrives. We show that our algorithms manage to spread the extra load created to a large part of the network by maintaining their nice load distribution and limited network traffic creation properties. Another important parameter we study is the rate of triple publication in the network. If RDF triples arrive more frequently, we also have to perform query processing operations more often. We demonstrate that our algorithms manage to keep distributing the extra load without creating heavy network traffic and without overloading a restricted set of nodes. We also show that if more resources are available they can nicely be used by our algorithms, e.g., when more nodes connect in the network, they will be assigned parts of the current query processing operations to remove load from existing nodes.

The rest of the paper is organized as follows. Section 2 presents our assumption regarding the system architecture, the data and query model. Sections 3 and 4 present the alternative indexing and query processing algorithms. In Section 5, we present a detailed experimental evaluation and comparison under various parameters. Finally, Section 6 presents related work and Section 7 concludes the paper.

## 2 System model and data model

In this section, we introduce the system and data model. Essentially, we extend the models of [12] to deal with the continuous query scenario of this paper.

**System model.** We assume an overlay network where all nodes are equal, they run the same software and they have the same rights and responsibilities. Nodes are organized according to the Chord DHT protocol [20] and are assumed to have synchronized clocks. In practice, nodes will run a protocol such as NTP and achieve accuracies within few milliseconds [3]. Each node can insert data and pose continuous queries. Each node $n$ has a unique key, denoted by $key(n)$ and each data item (RDF triple or query) has a key denoted by $key(i)$. Chord uses consistent hashing to map keys to identifiers. Each node and item is assigned an $m$-bit identifier using function $Hash(k)$ (e.g., SHA-1, MD5) that returns the $m$-bit identifier of key $k$. 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 $Successor(Hash(k))$. A query for locating the node responsible for a key $k$ can be done in $O(\log N)$ steps with high probability [20], where $N$ is the number of nodes in the network.

Our algorithms use the API defined in [9, 8] that provides two functionalities not given by the standard DHT protocols: (i) send the same message to multiple nodes and (ii) send $d$ messages to $b$ nodes where each node receives one or more messages. 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. This function is also used as $multiSend(M, I)$, where $M$ is a set of $d$ messages and $I$ is a set of $d$ identifiers ($b$ distinct ones). If more than one messages, say $j$, have the same receiver node $n$, then the identifier of $n$ will appear $j$ times in the set $I$. For each $I_j$, message $M_j$ is delivered to $Successor(I_j)$ in $O(b \log N)$ hops.

**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 *publishing* metadata in the form of *RDF triples*. In addition, each node can *subscribe with continuous queries* that describe resources that this node wants to receive *answers* about. 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. 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 [18], 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 of the following form: $?x_1, \ldots, ?x_n : (s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge \cdots \wedge (s_n, p_n, o_n)$

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 $?x_1, \ldots, ?x_n$ are called *answer variables* distinguishing them from other variables of the query. A query with a single conjunct is called *atomic*.

Let us define the concept of valuation (to talk about values satisfying 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 triples.

Each triple $t$ has a time parameter called *publication time*, denoted by $pubT(t)$, that represents the time that the triple is inserted into the network. Each query $q$ has a time parameter too, called *subscription time*, denoted by $subscrT(q)$. Each triple pattern $q_i$ of a query $q$ inherits the subscription time, i.e., $subscrT(q_i) = subscrT(q)$. A triple $t$ can satisfy/trigger a triple pattern of query $q$ only if $subscrT(q) \leq pubT(t)$, i.e., only triples that are inserted after a continuous query was subscribed can participating in its satisfaction. Finally, 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 $q$.

Let us now give the semantics of query answering in our continuous query processing setting. We first deal with instantaneous queries [12], and then use their semantics to define the concept of answer to a continuous query.

An *RDF database* is a set of triples. Let $DB$ be an RDF database and $q$ an *instantaneous* 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 $i = 1, \ldots, n$.

Let $q$ be a *continuous* query submitted to the network at time $T_0$ to be evaluated continuously for the interval $[T_0, \infty]$. Let $t$ be a time instant in $[T_0, \infty]$, and $DB_t$ the set of triples that have been published in the network during the interval $[T_0, t]$. The *answer* to query $q$ *at time* $t$, denoted by $ans(q,t)$, is the bag union of the results of evaluating the instantaneous query $q$ over $DB_{t'}$ at every time instant $T_0 \leq t' < t$.

The above definition assumes bag semantics for query evaluation. This semantics is supported by the algorithms CQC and CSBV. Simple modifications to the algorithms are possible so that set semantics (i.e., duplicate elimination) is also supported.

Note also that the above definition defines the answer to a query at each time $t$ after this query was submitted. In practice, continuous query processing algorithms such as CQC and CSBV will evaluate submitted queries incrementally i.e., triples in the answer will be made available to the querying node as soon as possible after they are generated.

## 3 The CQC algorithm

Let us now describe our first algorithm, the *continuous query chain* algorithm (CQC). In the presentation of our algorithms, it will be useful to represent a conjunctive query $q$ of the form $q_1 \wedge \cdots \wedge q_n$ in list notation i.e., as $[q_1, \ldots, q_n]$.

**Indexing a query.** Assume a node $n$ that wants to subscribe with a conjunctive query $q = [q_1, \ldots, q_k]$ with set of answer variables $V$. Node $n$ indexes each triple pattern $q_j$ to a different node $n_j$. Each node $n_j$ is responsible for query processing regarding $q_j$, and all nodes $n_1, \ldots, n_k$ will form the *query chain* of $q$. To determine the satisfaction of

$q$ for a given set of incoming triples, the nodes of a query chain have to collaborate by exchanging intermediate results.

Now let us see how a node indexes each triple pattern. For each triple pattern $q_j$ of $q$, $n$ computes an identifier $I_j$ using the parts of $q_j$ that are constant. For example, assume a triple pattern $q_j = (?s_j, p_j, ?o_j)$. Then, the identifier for $q_j$ is $I_j = Hash(pred(q_j))$ since the predicate part is the only constant part of $q_j$. This identifier is used to locate the node $n_j$ that will be responsible for $q_j$. In Chord terminology, this node will be the successor of the identifier $I_j$, namely $n_j = Successor(I_j)$. If a triple pattern has just one constant, this constant is used to compute the identifier of the node that will store the triple pattern. Otherwise, if the triple pattern has multiple constants, $n$ will heuristically prefer to use first the subject, then the object and finally the predicate to determine the node that will evaluate $q_j$. Intuitively, there will be more *distinct* subject or object values than *distinct* predicates values in an instance of a given schema. Thus, our decision helps us to achieve a better distribution of the query processing load.

So, for the query $q$, we have $k$ identifiers whose successors are the nodes that will participate in the query chain of $q$. Node $n$ has to send to each one of these nodes a message with the appropriate information notifying them that from there on, each one of them will be responsible for one of the triple patterns of $q$. The exact procedure is as follows. For simplicity, assume that triple patterns are indexed in the order they appear in the query. Thus, the first node in the query chain is responsible for the first triple pattern in the query, the second node is responsible for the second triple pattern and so on. In Section 4, we revisit this issue. For each triple pattern $q_1, \ldots, q_k$, $n$ creates a message $IndexTriplePattern(q_j, V, key(q), I_{j+1}, First)$ to be delivered to nodes $n_1, \ldots, n_k$ respectively. Identifier $I_{j+1}$ allows node $n_j$ to be able to contact the next node in the query chain $n_{j+1}$. When the message is sent to the last node $n_k$ in a query chain, this argument takes the value $key(n)$ so that $n_k$ will be able to deliver results back to the node $n$ that submitted $q$. Parameter $First$ is a Boolean that indicates whether $n_j$ will be the first node in the query chain of $q$ or not. After having created a collection of $k$ messages (one for each triple pattern), $n$ uses function *multiSend()* to deliver the messages. Thus, $q$ is indexed in $k * O(logN)$ overlay hops, where $N$ is the size of the network.

When a node $n_j$ receives a message $IndexTriplePattern()$, it stores all its parameters in the local *query table* ($QT$) and waits for triples to trigger the triple pattern.

**Indexing a new triple.** Let us now proceed with the next logical step in the sequence of events in a continuous query system. We have explained so far how a query is indexed. We will now see how an incoming triple is indexed. We have to make sure that a triple will meet all relevant triple patterns so that our algorithm will not miss any answers. Looking back to how a triple pattern is indexed, we see that we always use the constant parts of a triple pattern. Thus, we have to index a new triple in the same way. Therefore, a new triple $t = (s, p, o)$ has to reach the successor nodes of identifiers $I_1 = Hash(s)$, $I_2 = Hash(p)$ and $I_3 = Hash(o)$. The node that inserts $t$ will use the $multiSend()$ function to index $t$ to these 3 nodes in $O(logN)$ overlay hops. In the next paragraph we discuss how a node reacts upon receiving a new triple.

**Receiving a new triple.** Assume a node $n_j$ that receives a new triple $t$. $n_j$ has to determine if $t$ is relevant to any already indexed queries so $n_j$ searches its local $QT$ for triple patterns that match $t$. Assume that a matching triple pattern $q_j$ belonging to

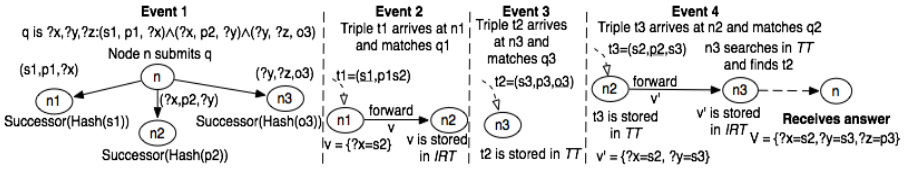


**Fig. 1.** The algorithm CQC in operation

query $q$ is found, i.e., there is a valuation $v$ over the variables of $q_j$ such that $v(q_j) = t$. According to the position/order of $n_j$ in the query chain of $q$, $n_j$ now acts differently. We will distinguish between two cases: (a) when $n_j$ is the first node in the query chain of $q$ and (b) when $n_j$ is any other node but the first one. For ease of presentation we point out now that in the second case, a node always stores the new triple in its *triple table* ($TT$). Later on, we will come back to this case to explain the rest of the steps.

If $n_j$ is the first node in the query chain of a query $q$. $n_j$ forwards valuation $v$ to the next node $n_{j+1}$ in the chain. Valuation $v$ holds a partial answer to $q$. Thus, from here on, we will call such valuations *intermediate results*. In the implemented system, $n_j$ creates the following message $FwdValuation(v, key(q))$ that has to be delivered to $n_{j+1} = Successor(Hash(I_{j+1}))$. So, for all $l$ queries in $QT$ whose triple patterns have been triggered in $n_j$ by $t$, $n_j$ will perform the operations we just described and use the $multiSend()$ function to forward the various intermediate results to the appropriate nodes in query chains. This will cost $l * O(logN)$ overlay hops.

**Receiving intermediate results.** Let us now see how a node $n_j$ reacts upon receiving an intermediate result i.e., a valuation $w$. First, $n_j$ applies $w$ to $q_j$, the triple pattern it is responsible for, to compute $q'_j = w(q_j)$. Then $n_j$ tries to find if triples matching $q'_j$ have already arrived. So, $n_j$ searches its $TT$ and for *each* triple $t \in TT$ that matches $q'_j$ (i.e., there is a valuation $v$ over the variables of $q'_j$ such that $v(q'_j) = t$), $n_j$ produces a new intermediate result, the valuation $w' = w \cup v$. Then, $n_j$ forwards the new intermediate results to the next node $n_{j+1}$ in the query chain of $q$ in a single message using the $Send()$ function with a cost of $O(logN)$ hops. In addition, $n_j$ will store the intermediate result $w$ locally in its *intermediate results table* ($IRT$) to use it whenever new triples arrive. Node $n_{j+1}$ that receives the set of new intermediate results will react in exactly the same way for each member of the set and so on. When the last node in the chain for a query $q$ (i.e., $n_k$) receives a set of intermediate results, stored triples in $TT$ are checked for satisfaction against each $q'_k$, and for each successful triple, an *answer* to the query $q$ is generated using each valuation $w'$ and is returned to the node that originally posed $q$.

Now we come back to finish the discussion on what happens when a node $n_j$ receives a new triple $t$ that triggers a triple pattern $q_j$ and $n_j$ is not the first node in the query chain of $q$. So far, we have only said that $n_j$ will store $t$ in its $TT$. In addition, $n_j$ has to search its $IRT$ table to see whether the evaluation of a query that has been suspended can now continue due to $t$ that has just arrived. For each intermediate result $w$ found in $IRT$ that can be used to compute $q'_j = w(q_j)$ that matches $t$, $n_j$ produces a new intermediate result (i.e., a valuation $w' = w \cup v$ where $t = v(q'_j)$) and forwards it to the next node $n_{j+1}$ in the query chain of $q$ with a $FwdValuation()$ message.

**Example.** CQC is shown in operation in Figure 1. Each *event* represents an event in the network, i.e., the arrival of a new triple or query. 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 and ease of presentation in each event we draw only the nodes that do something due to this event, i.e., rewrite a query, search or store queries or triples etc.

## 4 The CSBV algorithm

Let us now proceed with the description of our second algorithm, the *continuous spread-by-value* algorithm (CSBV). CSBV extends the ideas of CQC to achieve a better distribution of the query processing load. In CQC, a query chain for a query $q$ is created at the time that $q$ is submitted and leads to a query plan with a *fixed* number of participating nodes (one node per triple pattern in the absence of collisions in the DHT). Notice that whenever the first node $n_1$ in the chain for query $q = [q_1, \ldots, q_k]$ creates a new intermediate result (i.e., a valuation $v$), the triple pattern $q_1$ corresponding to $n_1$ is satisfied by $v$ and $q$ has been partially evaluated. The motivation for CSBV is that node $n$ can now use valuation $v$ to *rewrite* $q$ into a new query with *fewer* conjuncts $q' = [v(q_2), \ldots, v(q_n)]$ and decide *on the fly* the next node of the network to undertake the query processing of $q'$. Because $q'$ is conjunctive like $q$, its processing can proceed in a similar manner. The answer bag of $q$ can then be computed by the union of the answer bags of queries $q'$ combined with valuations $v$. In this way, a node $n_i$ in the chain of CQC for a query $q$ can be said to have multiple next nodes in CSBV depending on the triples that trigger $q_i$. Thus, the responsibility of evaluating the next triple pattern of $q$ is distributed to multiple nodes (depending on values used) compared to just one in CQC, leading to a much better query processing load distribution. If we generalize this to all nodes participating in a query plan, it is easy to see that a query plan in CSBV does not have the shape of a chain (as in CQC) but rather that of a tree. We will describe CSBV by pointing to the different actions that are taken comparing with CQC in each step.

**Indexing a query.** In CQC, when a query $q = [q_1, \ldots, q_k]$ is inserted, we immediately create a query chain of $k$ nodes. In CSBV, no query chain is created. Instead, $q$ is indexed *only to one node* that will be responsible for one of the triple patterns of $q$. Thus, a query is indexed with only $O(logN)$ hops. For now assume that, as in the description of CQC, triple patterns are handled in the order they appear in the query i.e., $q_1$ is used to index $q$ to node $n_1$.

In CSBV, we follow the same indexing heuristics as in CQC, when there is just one constant part in a triple pattern. But in case that there are multiple constants, we use the *combination* of *all* constant parts to index the query. For example, if $q_j = (?s_j, p_j, o_j)$, we have $I_j = Hash(pred(q_j) + obj(q_j))$. We use the operator $+$ to denote the *concatenation* of string values. Multiple constants typically occur in triple patterns where variables have been substituted by values of incoming triples (see discussion below in Paragraph "Receiving a new triple"). Using these combinations, a node in CSBV can direct intermediate results towards different branches of the distributed query plan tree (or dynamically create a new branch) depending on values used in incoming triples.

**Indexing a new triple.** As we discussed, CSBV uses the combination of constant parts in a triple pattern to index a query. Thus, in order not to miss possible answers, a new triple $t = (s, p, o)$ has to reach the successor nodes of identifiers $I_1 = Hash(s)$,

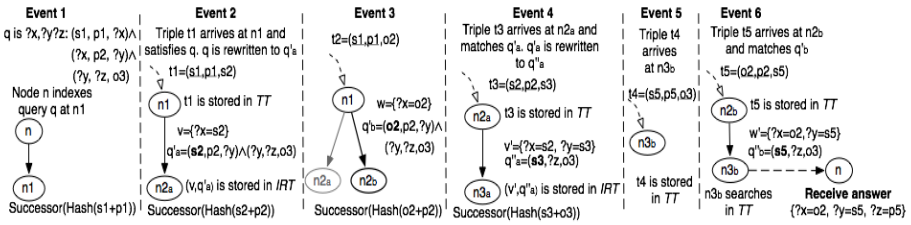


**Fig. 2.** The algorithm CSBV in operation

$I_2 = Hash(p)$, $I_3 = Hash(o)$, $I_4 = Hash(s+p)$, $I_5 = Hash(s+o)$, $I_6 = Hash(p+o)$ and $I_7 = Hash(s+p+o)$. Thus, a node $n_1$ that inserts $t$ will use the *multiSend*() function to index $t$ to these 7 nodes in $7 * O(logN)$ overlay hops.

**Receiving a new triple.** As in CQC, when a node $n_j$ receives a new triple $t$, first it has to find if $t$ triggers any local query $q$ (possibly in combination with some valuation $v$). If it does, $n_j$ rewrites $q$ using $t$ and $v$, and new intermediate results will be forwarded to the next node in the query chain. The critical difference with CQC, is how $n_j$ decides who will be the next node $n_{j+1}$. In CQC this information is given to each node in the chain upon insertion of the original query where the whole chain is created at once. Thus, in CQC, $n_j$ knows that $n_{j+1}$ is always the same node no matter what the triple that arrived is. On the contrary, in CSBV, this is a dynamic procedure and node $n_{j+1}$ can be a different node for different triples that arrive in $n_j$. Nodes in CSBV use the rewritten queries that they create to decide who the next node is.

Let us see an example. Consider the query $q = [(s_1, p_1, ?x), (?x, p_2, ?y), (?y, p_3, o_3)]$ indexed at node $n_1$. If $t_1 = (s_1, p_1, s_2)$ arrives, then the new rewritten query is $q' = [(s_2, p_2, ?y), (?y, p_3, o_3)]$ and the valuation is $v = \{?x = s_2\}$. Now the intermediate result is the *pair* $(v, q')$. In CQC, $n_2$ would be $Successor(Hash(p2))$ since this has been decided upfront (using the second triple pattern of $q$). However, CSBV uses $q'$ to decide what the next node will be. It exploits the new value $s_2$ in the first triple pattern of $q'$ to decide that the next node is the $Successor(Hash(s_2 + p2))$. Assume now that another triple $t_2 = (s_1, p_1, s_3)$ arrives at $n_1$. $n_1$ rewrites $q$ again and the new rewritten query now is $q'' = [(s_3, p_2, ?y), (?y, p_3, o_3)]$, while the new valuation is $w = \{?x = s_3\}$. In CSBV, node $n_1$ will forward the pair $(w, q'')$ to a different node than before, namely to node $Successor(Hash(s_3 + p_2))$, whereas in CQC it would go again to $Successor(Hash(p_2))$.

In CQC, nodes participating in a query plan for a query $q$ have the knowledge that they are members of this plan since they receive the appropriate triple pattern to be responsible for at the time that $q$ is submitted. Thus, when a node receives a new triple $t$, it does not need to store it if no locally stored triple pattern matches $t$. In CSBV, nodes do not have such knowledge since they are becoming part of a query plan dynamically, i.e., a node is not able to know if there is a query indexed in the network that can be triggered by $t$ in the future when other triples with appropriate values arrive. Thus, a node in CSBV *always* stores locally a new triple to guarantee completeness.

**Receiving intermediate results.** Let us now see how a node $n_j$ reacts upon receiving an intermediate result $(w, p)$ where $w$ is a valuation and $p = [p_1, \ldots, p_m]$ is a conjunctive query. First, $n_j$ tries to find if relevant triples have already arrived that can contribute to the satisfaction of $p_1$, and thus to the satisfaction of the original query $q$ from which $p$ has been produced after possibly multiple rewriting steps. For this rea-

son, $n_j$ searches its local table $TT$ and for *each* triple $t \in TT$ that matches $p_1$ (i.e., there is a valuation $v$ such that $v(p_1) = t$), $n_j$ produces a new intermediate result $(v', p')$. In this case, $v'$ is the union of $w$ with $v$ and $p'$ is $[v'(p_2), \ldots, v'(p_m)]$). To decide which node $x$ will receive the new intermediate result, $n_j$ uses the first triple pattern in $p'$ using combinations of constant parts whenever possible to compute the identifier that will lead to $x$. When all matching triples have been processed, a *set* of new intermediate results has been created each one to be delivered to a possibly different node. Then, function $multiSend()$ is used to deliver each intermediate result to the appropriate node with a cost of $z * Olog(N)$ hops, where $z$ is the number of unique identifiers created while searching $TT$. In addition, $n_j$ will store the intermediate result $(w, p)$ locally in its *intermediate results table* (*IRT*) to use it whenever new triples arrive.

Each node $n_{j+1}$ that receives one of the new intermediate results will react in exactly the same way and so on. When a node $n_k$ is responsible for the last triple pattern of a query and receives a set of intermediate results of the form $(w, [q_k])$ then no intermediate results are generated. Instead, stored triples in $TT$ are checked for satisfaction against $q_k$, and for each successful triple, an *answer* to the query $q$ is generated using $w$ and is returned to the node that originally posed $q$. In Figure 2, we show an example of CSBV.

**Optimizing network traffic.** To further optimize network traffic we use the *IP cache* (*IPC*) routing table we proposed in [12]. In both algorithms, each time a node $n_j$ forwards intermediate results to the next node $n_{j+1}$ in a query plan, we pay $O(logN)$ overlay hops. With the *IPC* after the first time that $n_j$ has sent a message to $n_{j+1}$, $n_j$ keeps track of the IP address of $n_{j+1}$ and uses it in the future when forwarding intermediate results through this query chain. Then, $n_j$ can send a message to $n_{j+1}$ in just 1 hop. Similarly, if a new triple initiates a new rewritten query $q$ in the root of a query chain of $k$ nodes, then $q$ will need $k * O(logN)$ hops to reach the end of the query chain. With *IPC*, it will need just $k$ hops. The cost for the maintenance of the *IPC* is only local.

**Optimizing a query chain.** It is important to find a *good order* of nodes in the query chain, so as to achieve the least possible network traffic and the least possible total load. A simple but powerful idea is to take into account the rate of published triples that trigger the triple patters of the query (e.g., the rate in the last time window). We place *early* in a query chain nodes that are responsible for triple patterns that are triggered *very rarely*, while nodes that are responsible for triple patterns that are triggered *more frequently* are placed towards the end. An easy way to do this at the expense of bookkeeping by individual nodes and some extra messages upon query indexing is to ask all nodes that will participate in the query chain for the rate of incoming triples related to the triple pattern that they are going to be assigned. Thus, a node needs $3k * O(logN)$ messages to index a query instead of $k * O(logN)$ . For example, in CQC, when a node $n$ wants to submit a query $q$ of $k$ triple patterns, it splits $q$ to the triple pattern it consists of and assigns each $q_i$ at a different node $n_i$. Before sending the triple patterns, $n$ sends a message $getRates(q_i)$ to each node $n_i$. When all answers return, $n$ decides the order of the nodes having the most frequently accessed triple pattern towards the end of the query chain. Similarly in CSBV, when a node wants to submit a query $q$, it asks all possible candidate nodes based on the triple patterns of $q$. Only one node $n_1$ is chosen to receive the query, the one responsible for the triple pattern with the lowest rate of incoming triples. From there on, when $n_1$, or any other node in the query chain of $q$,

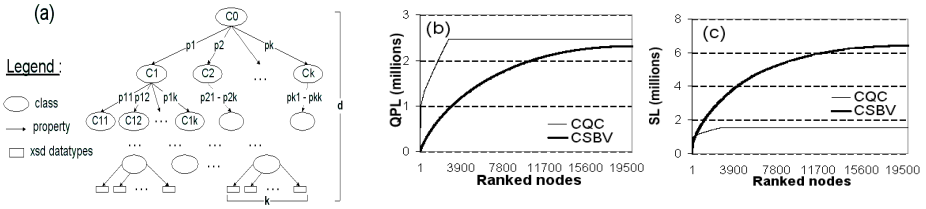


**Fig. 3.** Query processing and storage load

wants to forward intermediate results to a new node (i.e., create a new node in the query chain), it will follow the same procedure as to determine who will be this next node. These simple rules are sufficient to significantly improve network traffic in our setting at a minimal cost. The order of nodes can be periodically reevaluated and change (by migrating intermediate results through the nodes) in case the rates of incoming triples change. Due to space limitations we omit further analysis of these techniques.

## 5 Experiments

In this section, we experimentally evaluate our algorithms based on a Java implementation where we can run multiple nodes in one machine. We synthetically create a uniform workload as we did in [12]. We assume an RDFS schema of the form shown in Figure 3(a), i.e., a balanced tree of depth $d$ and branching factor $k$. Each class has a set of $k$ properties. Each property of a class $C$ at depth $l < d-1$ ranges over another class at depth $l+1$. Each class of depth $d-1$ has also $k$ properties which have values that range over XSD datatypes. To create a triple, we first randomly choose a depth. Then, we randomly choose a class $C_i$ among the classes of this depth. 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 $p$ 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 an important type of conjuctive queries, namely *path queries* of the form: $?x : (?x, p_1, ?o_1) \wedge (?o_1, p_2, ?o_2) \wedge \cdots \wedge (?o_{n-1}, p_n, o_n)$. To create a query, we randomly choose a property $p_1$ of class $C_0$. $p_1$ leads to a class $C_1$ at the next level. Then we randomly choose a property $p_2$ of $C_1$. This is repeated until we create $n$ triple patterns. For the last one, we randomly choose a value (literal) from the range of $p_n$ as $o_n$.

Our experiments use a schema with $d = 4$. The number of instances of each class is 1000, the number of properties that each one has is $k = 3$ while a literal can take up to 1000 different values. Finally, each query has 5 triple patterns.

**E1: Load distribution.** We define two types of load; the *query processing load* (QPL) and the *storage load* (SL). The QPL of node $n$ is the sum of the number of triples that $n$ receives to check against locally stored queries plus the number of intermediate results that arrive to $n$ to be compared against its locally stored triples. The SL of a node is the sum of the number of triple patterns for which it is responsible, plus the number of triples and intermediate results that it stores.

We create a network of $2 * 10^4$ nodes and insert $10^5$ queries. Then, we insert $6 * 10^5$ triples and measure the QPL and the SL of each node. In Figure 3(b), we show the QPL distribution. On the $x$-axis of this graph, nodes are ranked starting from the node

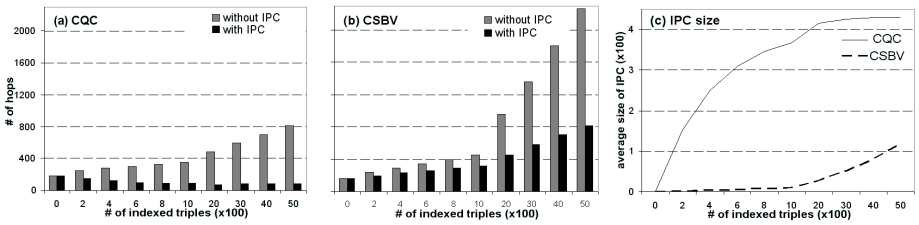


**Fig. 4.** Network traffic and *IPC* cost

with the highest load. The *y*-axis represents the *cumulative load*, i.e., each point $(a,b)$ in the graph shows that $b$ is the sum of the $a$ most loaded nodes. CSBV achieves to distribute the QPL to a significantly higher portion of nodes, i.e., in CQC, there are only 2685 nodes (out of $2*10^4$) participating in query processing, while in CSBV there are 19779 nodes. CSBV has a slightly lower total load than CQC since nodes in CSBV have more opportunities to group similar queries. Figure 3(c) shows the SL distribution. In CQC, the total SL is significantly less because in CSBV a new triple is indexed/stored four more times than in CQC, by using the combinations of the triple values. However, because of dynamic creation of query plans, this SL is nicely distributed. A higher total SL in the network is the price we pay for the better distribution of the QPL in CSBV.

**E2: Network traffic and *IPC* effect.** For this experiment, we create a network of $2*10^4$ nodes and install $10^5$ queries. Then, we train *IPCs* with a varying number of incoming triples, starting from 200 triples up to 5000. In each training phase, we insert 1000 triples and measure (a) the average number of overlay hops that are needed to *index* one triple and to *evaluate all* existing queries when using *IPCs*, (b) the size of *IPCs* at each node and (c) the same as (a) but this time we do not use *IPCs*. Finally, after each training phase, we measure how much it costs to insert a new triple.

Let us first see algorithm CQC, shown in Figure 4(a). The point 0 on the *x*-axis has the minimum cost, since it represents the cost to insert the first triple. There are no previous inserted triples so there are no partial results waiting for triples; therefore network traffic at this point is produced only because of the indexing of this triple to the network. *IPCs* are empty at this point so their use has no effect. However, in the next phases we observe a different behavior. Without *IPCs*, the network traffic required to insert a triple is increased, after each time we inserted a number of triples. This happens because each group of inserted triples results in the creation of new intermediate results. Thus, a next triple insertion has a higher probability to meet and trigger queries (and thus create more network traffic). This is why we see the gray bars in Figure 4(a) going higher after each phase. However, for the same reason, the black bars that represent the cost when using *IPCs* are going down. Triple insertions that trigger queries result in the forwarding of intermediate results. But when we use *IPCs*, these actions also fill the *IPCs* with IP addresses that can reduce subsequent forwarding actions. Thus, a next triple insertion will have a higher chance to cause forwarding of intermediate results with cost 1 instead of $O(logN)$ hops. For example, after 5000 triples, a triple insertion costs CQC 800 hops but with *IPCs* it costs only 60. Of course, this huge gain comes with a cost; in Figure 4(c) we show the average size (number of entries) of the *IPC* at each node. Naturally, this size is increased as more triples are inserted, but also observe

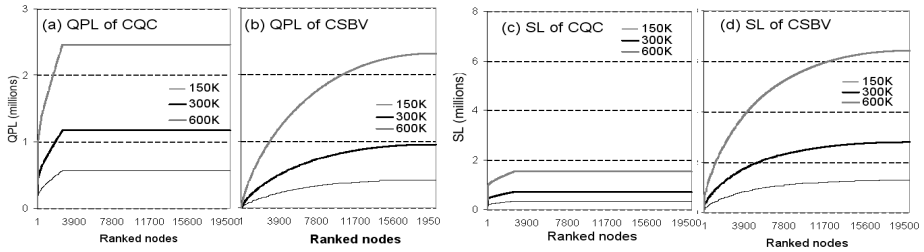


**Fig. 5.** QPL and SL when increasing the rate of incoming triples

that this is only a local storage cost at each node (there is no maintenance cost). Since even a small *IPC* size can significantly reduce network traffic (e.g., after 200 or 400 triples), we can allow each node to fill its *IPC* as long as it can handle its size.

In Figure 4(b), we show the network traffic cost for the CSBV algorithm. Results are explained with the same arguments as in CQC. The difference this time is that we see a much higher cost for CSBV both when using and when not using *IPCs*. This is due to the fact that nodes in CSBV cannot always group new intermediate results and send them with a single message to the next node in the query chain as it happens in CQC since usually there are more than one next nodes in CSBV. For the same reason, in Figure 4(c), we see that the *IPC* cost for nodes in CSBV is much smaller.

**E3: Effect of increasing the rate of incoming triples.** The base setting is a network of $2*10^4$ nodes with $10^5$ queries and $1.5*10^5$ incoming triples. We present how the two algorithms are affected when the incoming triples become $T = 3*10^5$ and $T = 6*10^5$.

In Figures 5(a) and (b), we show the cumulative query processing load distribution. In both algorithms the total load becomes higher, while the number of incoming triples is increasing since the already indexed queries are triggered by more triples. In CQC, the load distribution remains the same independently of the number of incoming triples; since a query chain is fixed at query submission, the responsible nodes remain always the same. Instead, in CSBV, query plans are formed while triples are arriving, thus, as the number of triples is increased, new responsible nodes are defined and inccur part of the QPL. Thus, the load distribution in CSBV becomes more fair as the number of triples increases. Notice, also that CQC reaches a higher total load. This is due to the creation time of the query chains. In CQC, the nodes that are responsible for the submitted queries are determined initially, so when triples are inserted they have to check for satisfied triple patterns or forwarded intermediate valuations. On the other hand, in CSBV, the nodes that incur the QPL, start when the appropriate triples that trigger the corresponding triple patterns are inserted, so they do not work in vain.

In Figures 5(c) and (d), we present the SL distribution. Naturally, in both algorithms, the total load increases with the number of inserted triples since each incoming triple is stored and creates intermediate results. As expected, the load in CSBV is higher since it indexes a triple to four more nodes than in CQC. Also, in CQC, a node stores a triple $t$ only if it is responsible for a triple pattern that is triggered by $t$ while in CSBV, a node always stores a triple it receives. Finally, CSBV uses a combination of the constant parts of a triple to index it, and thus it achieves a better load distribution than CQC.

**E4: Effect of increasing the number of queries.** The base setting for this experiment is a network of $2*10^4$ nodes with $2.5*10^4$ queries and $10^6$ triples. We increase

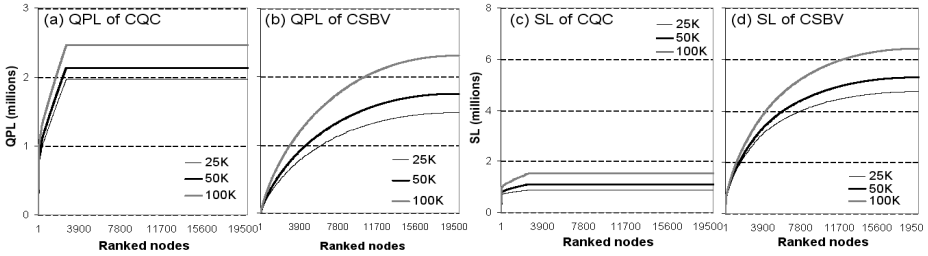


**Fig. 6.** QPL and SL when increasing the number of indexed queries

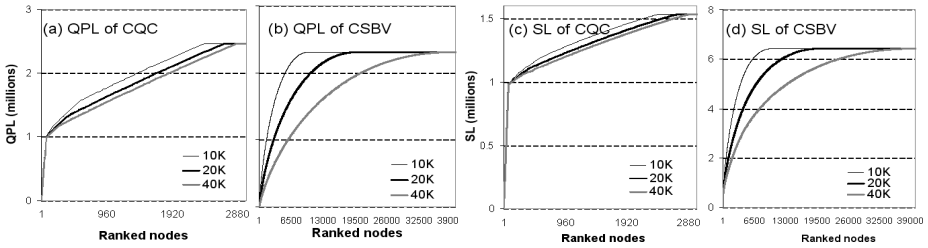


**Fig. 7.** QPL and SL when increasing the network size

the number of indexed queries to $5 * 10^4$ and $10^5$. In Figures 6 we see that the performance patterns remain the same while increasing queries, i.e., CSBV outperforms CQC in terms of load distribution (both for QPL and SL) due to the dynamic query plans. This comes at the expense of a higher SL per node.

**E5: Effect of increasing the network size.** In a network of $N = [10^4, 2*10^4, 4*10^4]$ nodes, we index $10^5$ queries and then we insert $10^6$ triples. Figure 7 shows that CQC is not able to exploit the new nodes while CSBV distributes the QPL and SL to almost as many nodes are available by dynamically creating query plans.

## 6 Related work

This paper extends the one-time query processing algorithms QC and SBV of [12] to a continuous query processing environment. Historically, the study of continuous querying of RDF data in P2P networks was initiated in [6]. [6, 5] deal with conjunctive multi-predicate queries (a subclass of the class of conjunctive triple pattern queries studied in this paper) and adopt HyperCup [17] as the underlying P2P infrastructure. Thus, their algorithms are not directly comparable with the ones of this paper.

[4] introduced publish/subscribe in the system RDF-Peers. The query language of RDF-Peers supports conjunctive multi-predicate queries, disjunctions of such conjunctions and range queries. RDF-Peers is built on top of an extension to Chord that supports order-preserving hashing so that range queries can be implemented easily. [4] concentrates mainly on one-time queries and the publish/subscribe subsystem of RDF-Peers is only briefly presented. Recently, [13] implemented and evaluated the algorithms QC and MQC for continuous conjunctive multi-predicate queries on top of Chord. QC is essentially the algorithm sketched (but not implemented or evaluated) in [4] while MQC is the algorithm that has motivated us to develop CSBV. It is not difficult to extend our algorithms to deal with disjunctions or range queries. For the former type of queries the

extension is straightforward; for the latter, we could rely on an order-preserving hashing extension of Chord such as the one of [4].

Finally, [21, 15, 14] are some other recent papers on continuous querying of RDF data. In these papers, graph-based RDF queries are supported using centralized indices.

From the area of relational databases, [9] is the paper most closely related to our work. In [9], we have discussed algorithms for continuous two-way equi-join queries.

## 7 Conclusions and future work

We introduced and compared two novel algorithms for the evaluation of continuous conjunctive triple pattern queries over RDF data stored in a DHT. The algorithms manage to distribute the query processing load to a large part of the network and keep network traffic low. Our future work plans are to design techniques for handling skewed workload efficiently and to take into account physical network proximity. We also plan to support RDFS reasoning. Since RDFS triples can be handled similarly with RDF triples, the main challenge is how to support the inference of *new* RDFS triples using the RDFS inference rules in a compatible way with our query processing framework. This can be done in a forward chaining manner by extending CSBV. The algorithms of this paper have recently been implemented in our system Atlas available at http://atlas.di.uoa.gr. We are currently evaluating Atlas on PlanetLab.

## References


[1] K. Aberer et al. The essence of P2P: A reference architecture for overlay networks. In *IEEE P2P 2005*.
[2] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion Project: From Data Integration to Data Coordination. *SIGMOD Record*, 32(3):53–58, September 2003.
[3] M. Bawa et al. The Price of Validity in Dynamic Networks. In *SIGMOD 2004*.
[4] 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.
[5] P. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Designing semantic publish/subscribe networks using super-peers. In *Semantic Web and Peer-to-Peer*. Steffen Staab and Heiner Stuckenschmidt (editors). Springer 2006.
[6] P. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/Subscribe for RDF-based P2P Networks. In *ESWS 2004*.
[7] O. Corcho, P. Alper, I. Kotsiopoulos, P. Missier, S. Bechhofer, and C. Goble. An overview of S-OGSA: a Reference Semantic Grid Architecture. *Journal of Web Semantics*, 4(2):102–115, June 2006.
[8] S. Idreos. Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. Master Thesis. Technical University of Crete, September 2005.
[9] S. Idreos, C. Tryfonopoulos, and M. Koubarakis. Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. In *ICDE 2006*.
[10] D. R. Karger and D. Quan. What would it mean to blog on the semantic web? *Journal of Web Semantics*, 3(2-3):147–157, October 2005.
[11] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *WWW 2002*.
[12] E. Liarou, S. Idreos, and M. Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. In *ISWC 2006*.
[13] E. Liarou, S. Idreos, and M. Koubarakis. Publish-Subscribe with RDF Data over Large Structured Overlay Networks. In *DBISP2P 2005*.
[14] H. Liu, M. Petrovic, and H.-A. Jacobsen. Efficient and scalable filtering of graph-based metadata. *Journal of Web Semantics*, 3(4):294–310, December 2005.
[15] M. Petrovic, H. Liu, and H.-A. Jacobsen. G-ToPSS - fast filtering of graph-based metadata. In *WWW 2005*.
[16] E. Prud'hommeaux and A. Seaborn. SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.
[17] M. T. Schlosser et al. HyperCuP - Hypercubes, Ontologies, and Efficient Search on P2P Networks. In *AP2PC 2002*.
[18] A. Seaborne. RDQL - A Query Language for RDF. W3C Member Submission, 2004.
[19] B. Simon et al. Smart Space for Learning:A Mediation Infrastructure for Learning Services. In *WWW 2003*.
[20] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM 2001*.
[21] J. Wang, B. Jin, and J. Li. An Ontology-Based Publish/Subscribe System. In *Middleware 2004*.