Continuous Multi-Way Joins over Distributed Hash Tables[*]

Stratos Idreos
CWI, Amsterdam
The Netherlands
s.idreos@cwi.nl

Erietta Liarou
CWI, Amsterdam
The Netherlands
erietta@cwi.nl

Manolis Koubarakis
National and Kapodistrian
University of Athens, Greece
koubarak@di.uoa.gr

ABSTRACT

This paper studies the problem of evaluating continuous multi-way joins on top of Distributed Hash Tables (DHTs). We present a novel algorithm, called recursive join (RJoin), that takes into account various parameters crucial in a distributed setting i.e., network traffic, query processing load distribution, storage load distribution etc. The key idea of RJoin is incremental evaluation: as relevant tuples arrive continuously, a given multi-way join is rewritten continuously into a join with fewer join operators, and is assigned continuously to different nodes of the network. In this way, RJoin distributes the responsibility of evaluating a continuous multi-way join to many network nodes by assigning parts of the evaluation of each binary join to a different node depending on the values of the join attributes. The actual nodes to be involved are decided by RJoin dynamically after taking into account the rate of incoming tuples with values equal to the values of the joined attributes. RJoin also supports sliding window joins which is a crucial feature, especially for long join paths, since it provides a mechanism to reduce the query processing state and thus keep the cost of handling incoming tuples stable. In addition, RJoin is able to handle message delays due to heavy network traffic. We present a detailed mathematical and experimental analysis of RJoin and study the performance tradeoffs that occur.

1. INTRODUCTION

We study the problem of evaluating continuous multi-way equi-joins on top of DHTs. Equi-join query processing over DHTs has been studied in the past in [15, 18] by exploiting hashing techniques. In [15, 16], the authors perform a study of one-time query processing over DHTs by evaluating existing distributed equi-join algorithms using the PIER query processor. These papers focus on *one-time* query processing leaving open the extension to the more challenging case of continuous queries. Thus, the most relevant previous work is [18] where we proposed and evaluated four algorithms for *two-way continuous equi-joins*, but left open the problem of the general case of multi-way continuous equi-joins. This is the main question we study in this paper, and make the following contributions.

First, we present algorithm RJoin (*recursive join*) for the evaluation of continuous multi-way joins on top of DHTs (for simplicity, in the rest of the paper the term "join" will refer to equi-join). Our paper is the first one to present a comprehensive solution to this open problem by generalizing and extending the techniques of [18]. The key idea of RJoin is to achieve good *balancing of the query processing load* by evaluating a multi-way join *incrementally.* As relevant tuples arrive continuously, RJoin rewrites a given multi-way join continuously into joins with fewer join operators and assigns these joins to different nodes of the network. The answer to the original query is the union of the answers of the rewritten queries. Thus, RJoin distributes the responsibility of evaluating a multi-way join to many network nodes by assigning parts of the evaluation of each binary join to a different node depending on the values of join attributes.

In addition to being an extension of the main ideas of [18], RJoin has a number of nice properties that are crucial for the multi-way joins case. First, RJoin is capable of evaluating sliding window joins. Our approach requires only local computations with zero overhead in terms of network traffic, and is flexible enough to allow a continuous query to define its own window duration, window type (sliding/tumbling) and finally whether the window duration is defined by the number of incoming tuples or by elapsed time. Window restrictions, in addition to more expressive queries, provide the means for efficient garbage collection/status reduction. This is of great importance, especially in join queries with long join paths, to keep resource usage at a steady level. RJoin can also handle message delays, e.g., due to heavy network traffic, gracefully without losing answers. Furthermore, it can support both set and bag semantics.

We present a formal analysis of the properties of RJoin. We show that RJoin is sound, enjoys a property of "eventual completeness" (a property similar to eventual consistency as known in distributed systems) under certain assumptions regarding the dynamism of the network, and does not generate any duplicate answers. Note that such mathematical analysis is absent from related papers [15, 18] where "best effort" query processing is assumed.

Finally, we show how to take into account appropriate environment parameters in order to choose suitable distributed query plans. The actual nodes to be involved in query pro-

[*]M. Koubarakis was supported in part by the European project Ontogrid

EDBT'08, March 25–30, 2008, Nantes, France.
Copyright 2008 ACM 978-1-59593-926-5/08/0003 ...$5.00.

cessing are decided by RJoin on-line after taking into account the rate of incoming tuples with values equal to the values of the joined attributes. In this way, a suitable query plan is selected that minimizes network traffic. This is very important in the case of multi-way joins since a bad plan (especially for long join paths) can lead to huge amounts of network traffic and query processing load. We present a detailed experimental analysis of the techniques used by RJoin and a discussion of the various tradeoffs that occur.

The rest of the paper is organized as follows. In Section 2, we present our assumptions regarding the network model we will use, and define an appropriate notion of continuous relational query evaluation under this model. In Section 3, we introduce algorithm RJoin and discuss its formal properties in Section 4. In Section 5, we show how RJoin supports sliding window joins. In Section 6, we discuss how RJoin takes into account the rates of incoming tuples to improve its query plans while Section 7 presents optimization techniques for minimizing network traffic. Section 8 presents a detailed experimental analysis. Finally, Section 9 discusses related work and Section 10 concludes the paper.

2. SYSTEM MODEL AND DATA MODEL

System Model. We assume an overlay network where nodes are organized according to a DHT protocol. Nodes can insert data and pose continuous queries. In our examples and experiments we use Chord [25] due to its simplicity and popularity. However, our proposal is DHT-agnostic, i.e., it can be applied on top of any DHT protocol since we only use the *lookup* API supported by all DHTs.

Let us briefly describe the Chord protocol. Each node n in Chord owns a unique key, denoted by $Key(n)$. Each item i also has a key, $Key(i)$. Consistent hashing is used to map keys to identifiers. Nodes and items are assigned m-bit identifiers (large enough to avoid collisions). Identifiers are created by hashing keys using a cryptographic hash function, such as SHA-1 or MD5. Function $Hash(k)$ returns the identifier of a key k. Identifiers are ordered in an *identifier circle (ring)* modulo 2^m i.e., from 0 to $2^m - 1$. A key k with identifier $id = Hash(k)$ is assigned to the first node n which is equal or follows id clockwise in the identifier space. Node n is the *successor* node of id, denoted by $Successor(id)$. We will often say that n is *responsible* for k. In a network of N nodes, a query for locating the node responsible for a key can be done in $O(\log N)$ steps with high probability [25].

Chord is able to deal gracefully with network dynamism i.e., nodes joining, failing or leaving voluntarily. Under certain assumptions stated explicitly in [25], the Chord protocols guarantee that a Chord network is robust in the presence of node joins, failures or voluntary leaves, and it will eventually reach stability so that each node will be able to contact any other node by using successor pointers. We make use of these assumptions of Chord in Section 4 to characterize formally the behavior of RJoin.

As in [6], we assume a *relaxed asynchronous model* of distributed systems where there are known upper bounds on process execution speeds, message transmission delays and clock drift rates. Thus, there is a universal maximum delay δ such that a message at time t at a node n will reach an alive destination node n' by time $t + \delta$.

RJoin uses the following API (originally used in [18]) for delivering messages through the network. Function *send(msg,id)*, where *msg* is a message and *id* is an identifier, delivers *msg* from any node to $Successor(id)$ in $O(logN)$ hops. Function *multiSend(msg,I)*, where I is a set of $d > 1$ identifiers $I_1, ..., I_d$ delivers *msg* to nodes $P_1, P_2, ..., P_d$ such that $P_j = Successor(I_j)$, where $1 < j \leq d$. This cost is $d * O(logN)$ hops. Function *multiSend()* is also used as, *multiSend(M,I)*, where M is a set of h messages and I is a set of h identifiers. For each I_j, message M_j is delivered to $Successor(I_j)$ in $h * O(logN)$ hops. Finally, $sendDirect(msg, addr)$ delivers *msg* to a known IP address *addr* in one hop. Our API uses only the *lookup* operation of a DHT thus it is applicable to any DHT protocol. A detailed description and evaluation of this API on top of Chord can be found in [17].

Like the related papers [15, 18], RJoin builds on top of existing DHTs since they offer very good support for the main functionality that we need (i.e., indexing). We assume that low level issues like node churn, node heterogeneity etc. are handled by the DHT layer and are offered via APIs to RJoin. This layered design approach offers more flexibility, rapid design and development as explained in [3] where the authors propose a multi-level API for P2P system design. RJoin belongs to the application layer of the multi-layered architecture of [3], and it is designed in such a way that it can make maximum use of the knowledge offered by data items (queries and tuples) to nicely utilize available network resources. Any DHT functionality offered at a lower level, e.g., identifier load balancing [19], communication APIs [17], replication and so on can be directly used together with RJoin to improve performance. In the experiments section we give a simple example involving load balancing.

Data Model. We use the *relational data model*. Data is inserted in the network in the form of tuples. As in [15, 18], different schemas can co-exist but schema mappings are not supported. Continuous queries are formed using the SQL query language and we study the case of multi-way equijoins. As in Tapestry [26], our relations are *append-only*, thus in this paper we do not consider any other operations beyond insertion of tuples. Updates are left for future work.

For each tuple t, the *publication time* of t, denoted by $pubT(t)$, is the time that the tuple was inserted into the network by some node. Similarly, for each query q, the *insertion time* of q, denoted by $insT(q)$, is the time that q was submitted to the network by some node. Each query q will be associated with a unique key, denoted by $Key(q)$, that is created from the key of the node n that poses it, by concatenating a positive integer to $Key(n)$.

The following definition captures the semantics of query answering in our model.

Definition 1. *Let q be a query submitted 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 tuples 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 SQL query q over $DB_{t'}$ at every time instant $T_0 \leq t' < t$.*

As it is standard, the above definition assumes bag semantics for SQL queries. In Section 4, we extend RJoin so that set semantics (i.e., duplicate elimination) are also supported.

The above definition defines the answer to a query at each time t after this query was submitted. In practice, RJoin evaluates submitted queries incrementally as new tuples arrive and tuples in the answer will be made available to the querying node as soon as possible after they are generated.

3. EVALUATING MULTI-WAY JOINS

Let us now proceed with the description of our algorithm. We first discuss the challenges one has to face and the decisions one has to make when designing such an algorithm. Then, we discuss the issue of tuple indexing, we show how we can remember tuple insertions and query status in a distributed way, we discuss query indexing issues and finally we give the details of the RJoin query processing algorithm.

For a multi-way join query that refers to n relations, we need n tuples, one from each distinct relation to form an answer tuple. According to the semantics of Section 2, tuples that satisfy a query should be published at or after the time the query has been submitted. In addition, there can be no guarantee that these tuples will arrive *simultaneously.* In fact, the temporal ordering according to which these tuples arrive is an important parameter, and may lead to different ways to evaluate a query. Finally, a single tuple can be used to compute answers for *multiple* continuous queries.

Therefore, the system has to *remember every* tuple insertion and try to *combine* it with *every future* tuple insertion in case that it can participate in an answer for *any* of the existing queries. Thus, careful handling is needed to match the correct tuples with the correct queries in an efficient way.

Evaluating continuous multi-way joins is a complicated problem in a centralized setting. Distributed computation of continuous multi-way joins is even more challenging since one has to take into account more parameters, e.g., network traffic and load distribution. RJoin evaluates the queries *continuously* (partially or fully) *while* new tuples are coming. Each new tuple triggers a series of events that may lead to the creation of one or more answers for one or more queries.

Tuple indexing. One of the key points when designing an in-network processing algorithm is where new tuples are indexed, i.e., in which node or nodes a new tuple will be stored (or try to find relevant queries). The strategy of tuple indexing determines/restricts the way we choose to evaluate queries. In RJoin, each node will index new tuples in the network using information found in the tuples, e.g., the name of the involved relation, attribute names, values etc. Such a strategy has the advantage that when looking for a data item, we do not have to contact all nodes of the network or have prior/global knowledge of where this data item is. Here the choice of using DHTs becomes useful, i.e., we can index data items and then find them fast if we know exactly what we are looking for. We will describe the exact protocol later in this section.

Remembering events. Continuing our high level discussion, we will see now how we can *remember*, i.e., keep state in our distributed setting. As we already discussed, it is necessary to remember each tuple insertion to use it in the future when more tuples arrive and try to create answers for relevant queries. Otherwise, we have to periodically try and evaluate all existing queries for all past tuples which would create much more network traffic and force network nodes to evaluate the same queries for the same data repeatedly.

Here we follow an approach that allows us to keep state for each query and each relevant data item in a distributed way. *Each* new tuple t will force all relevant queries to be *rewritten* into queries with *fewer joins* that reflect the fact that t has arrived. Let us give an example of this rewriting step. Assume the following query q_1:

(q_1) select R.B, S.B from R,S,P where R.A=S.A and S.B=P.B

Assume now an incoming tuple t of R where $t = (3, 5)$. A new rewritten query q_2 is created by replacing the attributes of R in q_1 with their corresponding value in t and simplifying the where clause if necessary. Then, q_2 is as follows:

(q_2) select 5, S.B from S,P where 3=S.A and S.B=P.B

Future tuples can safely be combined with rewritten queries to create new ones and so on, until the where clause of a query becomes equivalent to "true" and an answer to the original query can be created. In order to distribute the query processing load in the network and exploit available resources, each time a new rewritten query is created, it will be assigned to a different node where relevant data might arrive (and/or might already be there).

Query indexing. Thus, the next issue is how we decide where to index the queries or, in other words, decide the criteria for distributing the query processing load. We can distinguish between two kinds of queries. First, we have the *input queries*, i.e., the queries that were submitted to the network. Second, we have the queries that we create either by rewriting an input query or a query that has already been rewritten once. We will refer to these queries as *rewritten queries.* We will see below that rewritten queries allow more flexibility regarding where to index them.

As an example, consider an input query such as q_1 in our previous example. Trying to find a good way to index this query leads us to the following decisions. Recall that our main goal is to use multiple nodes of the network to exploit available resources and distribute query processing responsibilities. In this way, we would like different queries to be evaluated by different nodes. One way to do that would be to assign queries to nodes based on the relation names used in the query. For example, we can index q_1 using the name of the relation R. Thus, the node $n = Successor(Hash(R))$ will be responsible for q_1. Then, we can ensure without global knowledge that this node n will receive all new tuples of relation R. This is easy to ensure since the required information is contained in each tuple of R. However, such a solution would not scale since only a few nodes would suffer all the query processing load, namely, as many nodes as the distinct relations in the schema or schemas used.

In order to achieve a better load distribution, a possible next step is to also use the attribute names, e.g., we can index the query to $n = Successor(Hash(R + A))$. We use the operator '+' to denote concatenation of string values. Then, a new tuple of R can reach n if indexed by $R + A$. Of course, in order to guarantee completeness of the algorithm, a tuple has to be indexed under all its attributes. At this point, we observe that there is no more information in the query that we can use for indexing, i.e., information that can be found both in the query and in the relevant tuples, so the second solution is satisfactory and will be adopted.

Now, if we observe a rewritten query (e.g., q_2 in our earlier example), we can see that there is more to use, i.e., values that replaced attribute names during the rewriting process. In our example, the rewritten query can be indexed to node $n = Successor(Hash(S + A + 3))$. In order to make a relevant new tuple meet the query, we have to index new tuples in the same way too. This is possible since the required information exists in a tuple. Thus, in order to make tuples reach all relevant input *and* rewritten queries, we index new tuples both under the concatenation of relation name and attribute name, and under the concatenation of relation name, attribute name and value. Naturally, indexing using values gives us much better properties in terms of load distribution than using only relation names and attribute names.

When an item, a query or a tuple, is indexed using the concatenation of a relation name and an attribute name, then we will say that this item is indexed at the *attribute level*. Similarly, when an item is indexed using the concatenation of a relation name, an attribute name and a value, then we will say that it is indexed at the *value level*. We will now describe in detail the exact protocols to index queries and tuples, and present the query processing algorithm RJoin.

The recursive join algorithm. The "typical" temporal order of events in our continuous query processing algorithm is as follows. First, a query is submitted and is indexed somewhere in the network waiting for matching tuples. Then, tuples arrive. Each tuple that is a match, causes the query to be rewritten and recursively reindexed at a different node waiting for new tuples. When a rewritten query has a where clause equivalent to "true", it is not reindexed any further. Instead, an answer is formed and is sent back to the node that submitted the corresponding input query. Thus, we call our algorithm the *recursive join* (RJoin) algorithm. In the following paragraphs, we describe the individual steps in detail. For the time being let us assume that events take place according to the "typical" temporal order we just discussed. In Section 4, we revisit this assumption.

Let us first see how a node x submits a query q. Node x will index q in the network under the concatenation of a relation name and an attribute name which form one of the expressions $RelName.AttName$ in the where clause of q. Since there are multiple such expressions in the where clause of a multi-way join query, we have to make a choice. This is an important choice because it affects multiple optimization parameters. For now, assume for simplicity that we choose randomly (in our examples, we always choose the first expression in the order they appear in the where clause). In Section 6, we discuss how to make this choice in a judicious way. The query is sent then to the successor node of the identifier computed by hashing the concatenation of the chosen relation and attribute name. There, the query is stored waiting for future tuples.

In Procedure 1, we show how a new tuple is indexed. A tuple is indexed *twice for each attribute* it has; once at the attribute level and once at the value level. Procedure 1 shows in detail all the messages created by the publishing node and how the API presented in Section 2 is used to route them in the network.

It should be clear now that because a new tuple t of a relation R is indexed at the attribute level, it will reach all input queries that have been indexed under the concatenation of R and any of the attribute names of R. Queries that refer to attributes of R but are indexed at the attribute level using the concatenation of another relation and any of its attribute names, will still have the chance to meet the new tuple t as rewritten queries when they need the values used in t. This is why when a node receives a new tuple t at the value level, it will always store t locally (see Procedure 2) since it may become useful in the future when a rewritten query arrives.

Procedure 1 A node x inserts a tuple t of k attributes

```
$M = \varnothing$, $I = \varnothing$
for $i = 0$ to $k$ do
    $Key1 = RelationName(t) + AttributeName(t, i)$
    $id1 = Hash(Key1)$
    $Level1 = attribute$
    $msg1 = newTuple(t, Key1, IP(x), Level1)$
    $M = M \cup \{msg1\}$
    $I = I \cup \{id1\}$
    $Key2 = RelationName(t) + AttributeName(t, i) + Value(t, i)$
    $id2 = Hash(Key2)$
    $Level2 = value$
    $msg2 = newTuple(t, Key2, IP(x), Level2)$
    $M = M \cup \{msg2\}$
    $I = I \cup \{id2\}$
end for
$multiSend(M, I)$
```

Procedure 2 A node n receives a new tuple with a message $newTuple(t, id, Key, IP(x), Level)$

```
$Q$ = list of input queries $q$ stored locally that match $t$ based on $Key$
$M = \varnothing$, $I = \varnothing$
for $i = 0$ to $sizeof(Q)$ do
    $q'_i = rewrite(q_i, t)$
    if $WhereClauseOf(q'_i) == True$ then
        $msg = CreateAnswer(q'_i)$
        $sendDirect(msg, Owner(q_i))$
        continue
    end if
    $Key = nextKey(q'_i)$
    $id = Hash(Key)$
    $msg = Eval(q'_i, key, Owner(q_i))$
    $M = M \cup \{msg\}$
    $I = I \cup \{id\}$
end for
if $Level == Value$ then
    store $t$ locally
end if
$multiSend(M, I)$
```

Let us then go through the actions taken by a node n that receives a tuple t. The exact steps are described in Procedure 2 and are almost the same independently of whether n receives t at the attribute or value level (this is signaled by parameter $Level$ in Procedure 2). In both cases, the receiver node searches locally to find which queries are triggered by t. A query q is *triggered* by a tuple $t = (v_1, \ldots, v_m)$ over a schema $R(A_1, \ldots, A_m)$ if $pub(t) \geq insT(q)$ and the where clause of q contains a conjunct either of the form $R.A_i = S.B_j$ or of the form $R.A_i = v_i$. Each triggered query q_1 will be rewritten using tuple t. If a rewritten query q_1 has a where clause equivalent to "true", then the query q from which q_1 originates has been satisfied, and an answer can be created and returned to the node that posed q. The remaining rewritten queries are reindexed to other nodes to find (or wait for) the rest of the matching tuples needed so that we can create answers given tuple t. The message msg sent contains the rewritten query, the key used to index it (so that the next node can search its local data based on that key) and the identifier of the node that submitted the input query (this is returned by the function $Owner()$). To decide where to index the rewritten query, the function $nextKey()$ is used. This function returns the relation-attribute pair or relation-attribute-value triple to be used as the key to determine the node that will receive the rewritten query. For now, assume for simplicity that $nextKey()$ makes this choice randomly (in our examples, $nextKey()$ returns the first pair or triple in the where clause of the rewritten query). The exact algorithm used by $nextKey()$ will be discussed in Section 6. If t has been received by n at the value level ($Level = value$), t is stored locally to wait for future rewritten queries.

Note that only tuples received by a node at the value level

Procedure 3 A node receives a rewritten query with a message $Eval(q, Key, I, Owner(q))$

```
store $q$ locally as a rewritten query
$T$= list of tuples $t$ stored locally that match $q$ based on $Key$
$M = \varnothing$, $I = \varnothing$
for $i = 0$ to $sizeof(T)$ do
  if $pub(t_i) >= insT(q)$ then
    $q' = rewrite(q, t_i)$
    if $WhereClauseOf(q') == True$ then
      $msg = CreateAnswer(q')$
      $sendDirect(msg, Owner(q))$
      continue
    end if
    $Key = nextKey(q')$
    $id = Hash(Key)$
    $msg = Eval(q', key, Owner(q))$
    $M = M \cup \{msg\}$
    $I = I \cup \{id\}$
  end if
end for
$multiSend(M, I)$
```

are stored locally. Tuples received by a node at the attribute level are used to trigger and rewrite stored input queries and then are discarded. We revisit this choice and point out its significance in Section 4.

Let us now describe the actions taken by a node when it receives a rewritten query (see Procedure 3). First, the query is stored locally since matching tuples might arrive in the future. In addition, there might be already some matching tuples stored locally, i.e., tuples that were published *after* the input query was submitted and arrived at this node *before* this rewritten query. For each such tuple the query is further rewritten and each new rewritten query with a where clause not equivalent to "true" is reindexed to a new node. For each new rewritten query with a where clause equivalent to "true", an answer is created and is sent to the node that submitted the corresponding input query.

Summarizing the steps of our algorithm as presented above, we can make the following observations. At each event, e.g., a tuple insertion, one or more steps may happen, e.g., a node may forward a rewritten query to another node and this node may further rewrite this query into one or more queries and forward them to more nodes. Furthermore, the distributed query plan for a query is dynamically built as new tuples arrive. Initially, the query is in its initial place in the index. Then, as tuples arrive to this node, the query is migrated to other nodes according to incoming tuple values.

Since rewritten queries are born with every tuple insertion, this leads to an ever increasing state. It is important, especially for long join paths, to have a mechanism to reduce this state. We discuss this issue in detail in Section 5.

Intuitively, nodes where queries or tuples are indexed at the attribute level incur more load than those nodes that their indexing targets at the value level regarding how often a node is required to process an incoming tuple. For example, a node responsible for $R.B$ receives more tuples to process than a node responsible for $R.B + v$, where v is a value that $R.B$ can take. This issue has been solved through replication of queries in [18] so we do not further pursue this in the current paper.

Example. RJoin is shown in operation in Figure 1 through a simple example. Each event in this figure represents an event in the network, i.e., either the arrival of a new tuple or the arrival of a new query. Events are drawn starting from the top 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 for tuples, evaluate a query etc. In Events 2 and 3, we show how a query is being rewritten and sent to another node according to the values used in an incoming tuple. Event 5 shows the case where a rewritten query that arrives to a new node is triggered by a tuple already stored there.

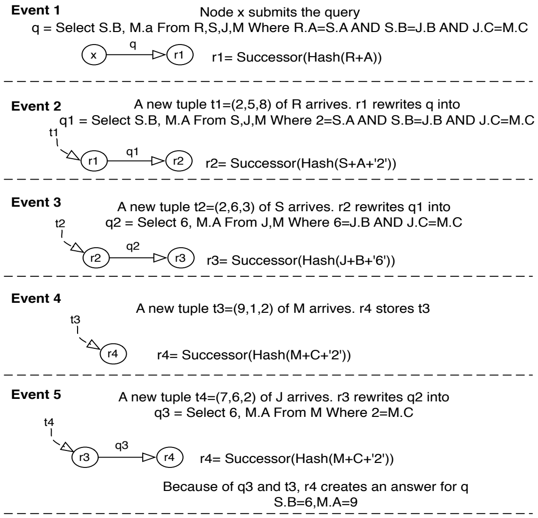

Event 1
Node x submits the query
q = Select S.B, M.a From R,S,J,M Where R.A=S.A AND S.B=J.B AND J.C=M.C
x
q
r1
r1= Successor(Hash(R+A))
Event 2
A new tuple t1=(2,5,8) of R arrives. r1 rewrites q into
q1 = Select S.B, M.A From S,J,M Where 2=S.A AND S.B=J.B AND J.C=M.C
t1
r1
q1
r2
r2= Successor(Hash(S+A+'2'))
Event 3
A new tuple t2=(2,6,3) of S arrives. r2 rewrites q1 into
q2 = Select 6, M.A From J,M Where 6=J.B AND J.C=M.C
t2
r2
q2
r3
r3= Successor(Hash(J+B+'6'))
Event 4
A new tuple t3=(9,1,2) of M arrives. r4 stores t3
t3
r4
r4= Successor(Hash(M+C+'2'))
Event 5
A new tuple t4=(7,6,2) of J arrives. r3 rewrites q2 into
q3 = Select 6, M.A From M Where 2=M.C
t4
r3
q3
r4
r4= Successor(Hash(M+C+'2'))
Because of q3 and t3, r4 creates an answer for q
S.B=6,M.A=9

Figure 1: An example

4. FORMAL PROPERTIES OF RJOIN

We will now characterize the behavior of RJoin formally.

Soundness and Completeness. It is easy to see that RJoin is a sound algorithm. RJoin might fail to be a complete algorithm (i.e., return all answers to a query) due to network dynamism (e.g., nodes failing or leaving silently) or even simple network delays. So far, we have not dealt with network dynamism and assumed that events follow their "typical" temporal order in the description of RJoin. Unfortunately, even if this "typical" temporal order of events is not followed e.g., due to network delays, RJoin may fail to produce an answer as the following example illustrates.

Example 1. *Let us consider relations $R(A_1, A_2, A_3)$ and $S(B_1, B_2, B_3)$ and the following query q*

$$\text{select } R.A_1, S.B_1 \text{ from } R, S \text{ where } R.A_2 = S.B_2$$

submitted at time T_0. If $R.A_2$ is used to index q at attribute level to node n then $n = Successor(Hash(R{+}A_2))$. Let t_q be the time that n receives q. If a tuple τ is published to relation R at time $T_1 > T_0$, τ will be sent to n as well (Procedure 1). Let t_τ be the time that n receives τ. If $t_\tau > t_q$ then τ will trigger q and RJoin will work correctly by creating and further forwarding a rewritten query q'. However, if $t_\tau < t_q$, then q and τ will not meet and a potential answer might be lost. The case where $t_\tau = t_q$ simply depends on whether n will process q or τ first. If τ is processed first, then again a potential answer might be lost.

To ensure that RJoin will never lose an answer due to message delays, we introduce the following rules (modifications to RJoin):

1. When a node receives a tuple τ at the attribute level, it stores τ in a new local table, called the *attribute level tuple table* (ALTT). τ will be removed from ALTT after time $\Delta > 0$.

2. When a node receives an input query q at the attribute level, it always searches for matching tuples in ALTT.

Naturally, Δ can be infinity (i.e., tuples are not removed from ALTT). This rather extreme solution will be useful if we want to support one-time queries in the same framework. However, to reduce the storage load per node at the attribute level, a garbage collection approach may be used. Then, Δ can be a system parameter or even be defined by each node separately given its system resources. It is possible to estimate an appropriate value of Δ so that we do not lose completeness of RJoin. Since each node in a Chord network can contact another node using *lookup* in $O(\log N)$ hops, and we assume that it takes at most δ time units to reach another node (see Section 2), then each node running our protocol can estimate the number of nodes in the network (e.g., using the techniques of [14]) and then compute an overestimate of Δ: an upper bound on the number of time units it takes for a message to be transmitted by one node to another node in a Chord network using *lookup*.

Given this extension, the following definition and theorem show that RJoin enjoys the useful property of *"eventual completeness"* (similar to eventual consistency as known in distributed systems) even in the presence of network delays.

Definition 2. *Let q be a continuous SQL query submitted at time T_0. The set of* new answers *to q at time instant $t > T_0$, denoted by $new(q,t)$, is the set of tuples τ such that τ is in the answer of instantaneous SQL query q over DB_t, but it is not in the answer of instantaneous SQL query q over $DB_{t'}$ for all $T_0 < t' < t$.*

Theorem 1. *Let q be a multi-way join submitted by node n_q at time T_0. Let $t > T_0$ be a time instant and τ a tuple with the same arity as q. Under the assumption that nodes can always compute the function $Successor()$ correctly, if $\tau \in new(q,t)$ then τ will eventually be delivered to node n_q by RJoin at some time $t' > t$.*

Proof. Let q be an m-way join of the form

$$\mathsf{select} \ * \ \mathsf{from} \ R_1, \dots, R_m \ \mathsf{where} \ \phi$$

that is processed recursively at some step of RJoin. We assume that each relation R_i has schema $\{A_{i1}, \dots, A_{if_i}\}$.

Let $\tau \in new(q,t)$. Then, there are tuples $\tau_1, \dots, \tau_m$ such that τ_i is in the instance of R_i at time t, ϕ is satisfied when values from the τ_i's are substituted for attribute expressions, and $\tau = \tau_1 \bowtie \cdots \bowtie \tau_m$. Without loss of generality, let us assume that $\tau_1, \dots, \tau_m$ were published in the network at times $t_1, \dots, t_m$ respectively and $t_1 < \cdots < t_m$. Then, obviously, $T_0 \le t_1 < \cdots < t_m = t$ from the definition of $new()$. We can now distinguish two cases:

1. q is indexed by RJoin at attribute level e.g., using relation R_k and one of its attributes A_{kl}. In this case, q is an input query (but See Section 6 for a generalization of RJoin where q can be a rewritten query; this case makes sure that the proof goes through for this generalization too). Therefore, RJoin forwards q to node $n_k = Successor(H(R_k + A_{kl}))$ where it is stored from time s onwards.

   Let $t'_k \ge T_0$ be the time tuple τ_k arrives and gets stored at node n_k. Let t''_k be the time τ_k is garbage collected from n_k by RJoin. If $t'_k > s$ then τ_k will arrive at node n_k after query q and will trigger it (Procedure 2). If $t'_k < s$ then τ_k will still be at node n_k when q arrives, and would have not been garbage collected yet because $s - t'_k < \Delta$ and $t''_k - t'_k \ge \Delta$ (it is $s - t'_k < \Delta$ because $T_0 \le t'_k$ and $s - T_0 < \Delta$ since $s - T_0$ is the time it takes for the rewritten query to be transmitted from node n_q to node n_k where it is stored). Thus, tuple τ_k will be used to rewrite q into a new query q'. If $t'_k = s$, then even if τ_k is processed first and q second, τ_k will still be available and q will be triggered because of our assumptions about Δ.

2. q is indexed by RJoin at value level e.g., using relation R_k, one of its attributes A_{kl} and value v. In this case q is a rewritten query. q will be forwarded by RJoin to node $n_k = Successor(H(R_k + A_{kl} + v))$ where it is stored from time s onwards (Procedure 3). According to Procedure 3, n_k first stores τ locally from time s onwards, and then searches for locally stored tuples that match q.

   Let t'_k be the time tuple τ_k arrives and gets stored at node n_k. If $t'_k > s$, q will be triggered. q will be triggered even if $t_k < s$ since when n_k receives q it searches for locally stored matching tuples. Finally, for the same reasons, q will be triggered even if $t_k = s$, independently of whether n_k processes q or τ first.

In both of the above cases, tuple τ_k will be used to rewrite q into a new query q' and RJoin will continue in a similar way after being invoked recursively on q'. Thus, eventually tuple τ will be formed and delivered to node n_q using the underlying network infrastructure. Throughout the proof, the assumption regarding network nodes allow us to conclude that RJoin messages will eventually be delivered to their right recipients using our protocol since the function $Successor()$ will be computed correctly. □

If $m = 2$, the above theorem shows that algorithm SAI of [18] shares the same eventual completeness property with RJoin if it is modified as we have suggested above for RJoin.

One might say that the above assumption regarding the computation of $Sucessor()$ is rather strong, given the typical assumptions for network dynamism in DHTs. Nevertheless, as new DHT protocols are designed and further properties of Chord-like DHTs are proven, it is beneficial to have a clear formal analysis for algorithms built on top of DHTs.

Duplicate Elimination. The following example shows that RJoin can generate duplicate answers to a given query.

Example 2. *Let us consider relations $R(A_1, A_2, A_3)$ and $S(B_1, B_2, B_3)$ and the query*

$$\mathsf{select} \ R.A_1, S.B_1 \ \mathsf{from} \ R, S \ \mathsf{where} \ R.A_2 = S.B_2$$

submitted at time 0. If tuples $(1,2,3), (b,2,c), (b,2,e)$ are published to relations R, S and S at times 1, 2 and 3 respectively, then RJoin will return the answer $(1,b)$ twice: first, after $(b,2,c)$ is published to S and, again, after $(b,2,e)$ is published to S.

We modify RJoin as follows so that no duplicate answers are returned and query evaluation assumes a semantics based on sets instead of bags. Our solution is based on local computations only. If an input query q requests distinct answers using the keyword DISTINCT, then each node n that receives and stores a rewritten query q' originating from q does the following. Let us assume that n receives at value level a tuple τ of relation R that triggers q'. Let $A_1, \ldots, A_k$ be the attributes of R in the select or where clause of q'. The new rule is that n keeps track of the projection $\pi_{A_1,\ldots,A_k}(\tau)$ and allows a new tuple τ' to trigger q' only if its projection on attributes $A_1, \ldots, A_k$ has not occurred in the past in one of the tuples that have already triggered q'.

The duplicate answers above were due to the nature of the submitted query and the published tuples. The following theorem shows that the continuous rewriting of queries in RJoin cannot lead to "accidental" duplicates. The proof of the theorem is by contradiction and is omitted due to space.

THEOREM 2. *Let q be a multi-way join in SQL involving relations $R_1, \ldots, R_m$. Let $\tau_1, \ldots, \tau_m$ be tuples of appropriate arities that satisfy the* where *clause of q and produce the answer τ. Under the same assumptions as in Theorem 1, τ will be delivered to node n_q exactly once if n_q submits continuous query q to the network at time t and each $\tau_i, 1 \leq i \leq m$ is published to the network at time $t' \geq t$.*

5. SLIDING WINDOW MULTI-WAY JOINS

In the previous sections, we described the main concepts of the RJoin algorithm. Here, we extend RJoin to support *sliding window multi-way joins* over data streams: given m streams $S_1, \ldots, S_m$ of data tuples published to the network and m sliding windows, we would like to evaluate the join of all m windows [12]. In sliding window joins, a window "slides", i.e., if we have a time window of duration T, then a tuple inserted at time t_1 can be combined only with tuples that arrive between t_1 and $t_1 + T$.

Such time restrictions are often very useful in network applications [7]. Another important usage of window queries is the fact that they limit the amount of data processed. In a streaming/continuous query environment this of great importance. For example, for the case of join queries, if there are no window restrictions, then every new tuple has to be considered together will all past tuples for all existing relevant queries. Both in a centralized and in a distributed setting this leads in an ever increasing usage of system resources. In the case of RJoin, the effect is the increasing number of stored rewritten queries which in turn trigger incoming tuples and so on, leading to a continuous increase in the amount of network traffic, query processing load and storage load required to handle incoming tuples. Window restrictions act as a garbage collection mechanism, allowing the system to discard past (possibly expired) events, focus on the current parts of the data and in this way keep a constant cost for query processing/event handling.

In the rest of this section we show how RJoin supports *time-based* sliding windows and *tuple-based* sliding windows as defined in [1]. In the former case, the duration of a window is defined in terms of time, while in the latter case it is defined in terms of the number of tuples that arrive in a window. In RJoin we can support sliding window joins with simple extensions that require only local computations. Recall that a rewritten query q_1 reflects the fact that certain tuples have arrived in the past and led to its creation. If these tuples are no longer in the relevant window, all we have to do is to forget q_1, i.e., delete it, and thus stop creating rewritten queries and eventually answers by rewriting and reindexing q_1. In this way, we are neglecting the combination of the tuples that originally led to q_1 and this is exactly what we need to do to support sliding windows.

In our extension of RJoin, each query has the following parameters: a Boolean parameter $useWindows$ to indicate if this is a window join or not, a numeric parameter $window$ that defines the duration of the window and, finally, a numeric parameter $start$ that denotes the beginning of the window. For simplicity, we assume that all windows are the same; the extension to the multiple-window case is obvious.

RJoin can handle time-based sliding window joins if it is extended with the following rules:

- When a tuple τ is indexed at the attribute level and triggers an input query q, then each rewritten query q_i created, inherits parameters $useWindows$ and $window$ from q. Furthermore, we have $start(q_i) = pubT(\tau)$.
- When a rewritten query q_1 is triggered at a node n by an incoming tuple τ, n checks if q_1 is still valid, i.e., if $|start(q_1) - pubT(\tau)| + 1 \leq window(q_1)$. If this is not true, then n deletes q_1. If it is true, then the generated rewritten query q_2 inherits parameters $useWindows$, $window$ and $start$ from q_1.
- Similarly, when a rewritten query q_1 is indexed at a node n, it is triggered by an already stored tuple τ at n only if $|start(q_1) - pubT(\tau)| + 1 \leq window(q_1)$. The generated rewritten query q_2 inherits parameters $useWindows$ and $window$ from q_1 while $start(q_2) = max(start(q_1), pubT(\tau))$.

With these simple rules that require only local actions RJoin supports sliding window joins. Tuple-based windows are supported in a similar way. The only difference is that a node has to monitor how many tuples have arrived in the current window for a given relation that may trigger a rewritten query. Other types of sliding windows e.g., time-based tumbling windows [7] can also be easily supported by RJoin using similar extensions. Proofs of formal properties of these algorithms are similar to the ones for the basic version of RJoin and are omitted due to available space.

6. DECIDING WHERE TO INDEX QUERIES

Let us now discuss the important issue of how we choose the relation name and attribute name we use to index a query (an input query or a rewritten one). So far, for simplicity, we described the algorithm RJoin by evaluating the joins of a query in the order they appear in the where clause. This can be highly inefficient. The critical parameter is the *rate of incoming tuples* of each relation involved in the query. We would like to evaluate the joins in such an order so that the number of intermediate results (i.e., rewritten queries) transferred is the minimum possible. In this way, we improve network traffic and also minimize the query processing load since by transferring fewer queries further in the query plan, less work has to be done. The critical question is which queries are waiting for which tuples. If a query is waiting for tuples of a relation that arrive at a very high rate, then a lot of traffic and query processing load is generated since *each new tuple* leads to rewriting and reindexing of the query. Thus, we would like to index an input query under the concatenation of a relation name and one of its attribute names such that the tuples of the relation arrive at a low rate. Similarly with rewritten queries; we would like to avoid indexing a query under a relation-attribute-value triple (R, A, v) if tuples of R with $A = v$ arrive very often.

In the one-time query scenario, this problem is equivalent to choosing the appropriate join order in a query plan based on *selectivity estimation*. In the continuous query scenario, we have to make a *prediction*. We have to choose a relation (and any of its attributes) or a relation, one of its attributes and a value such that the corresponding rate of tuple publications is predicted to be low. Whether this choice is right or wrong will be judged by future tuple insertions.

Collecting RIC information. Our solution is that before indexing a query (input or rewritten) a node *asks* each candidate node for information about the rate of incoming tuples (we will refer to this kind of information as *RIC information*). Then, the node takes a decision.

The above approach costs only a few extra messages and leads to significant improvements. The cost to index an input query without requesting RIC information is $O(logN)$, i.e., we pick one attribute randomly and index the query using the name of this attribute. Now the cost becomes $O(k * logN)$ where k is the number of distinct relations in the where clause of the query. In the implementation, we use the $multiSend(M, I)$ primitive of our API that exploits grouping of messages in the network (see Section 2). Assume a node n that wants to index a query q and there are 3 candidate nodes, n_1, n_2 and n_3. If (n_1, n_2, n_3) is the optimal order to contact these nodes, first a message *msg* goes from n to n_1 requesting RIC information. Then, *msg* is forwarded from n_1 to n_2 by piggy-backing the RIC information of n_1 and the address IP(n_1). Similarly, n_2 forwards *msg* to n_3 including RIC information about tuples it receives and the address IP(n_2). Finally, n_3 sends all RIC information and IPs to n in one hop since its address is included in the request. Then, after having seen all answers and taken a decision, the query can be indexed with one extra hop since all answers contain the IP address of the candidate nodes. This leads to a total cost of $O(k * logN) + 2$ messages. Observe that this is a cost we pay *only once* for each input query while the benefits of avoiding large amounts of network traffic are applicable to every tuple insertion.

In our experiments, in order to investigate the cost and benefits of utilizing RIC information to make indexing decisions, we do the following: we observe what has happened during the last time window and assume a similar behavior for the future. Of course, there are many more cases that one can consider. For example, some values might have a periodic behavior or others might be extreme cases, outliers that occur only once or only for one period and so on. What is a good model to make this prediction is out of the scope of this paper. Here we point out that our distributed query processing algorithm, RJoin, can take into account such issues in a viable way. More sophisticated criteria to make this prediction can be directly plugged-in since this is a local decision of the node that indexes the query after having seen relevant data from candidate nodes.

There is a final point that we wish to make. In Section 3, a rewritten query was always indexed using a relation-attribute-value triple since this intuitively gives more indexing possibilities. The discussion of this section forces us to change this part of RJoin as follows. To index a rewritten query q, we consider the following indexing possibilities and choose the most appropriate one depending on the RIC information received from relevant nodes: (a) all relation-attribute pairs that appear in a join condition in the where clause of q, (b) all relation-attribute-value triples that appear explicitly as a selection condition in the where clause of q and (c) all relation-attribute-value triples such that the selection condition *relation.attribute = value is logically implied* by the where clause of q.

7. MINIMIZING NETWORK TRAFFIC

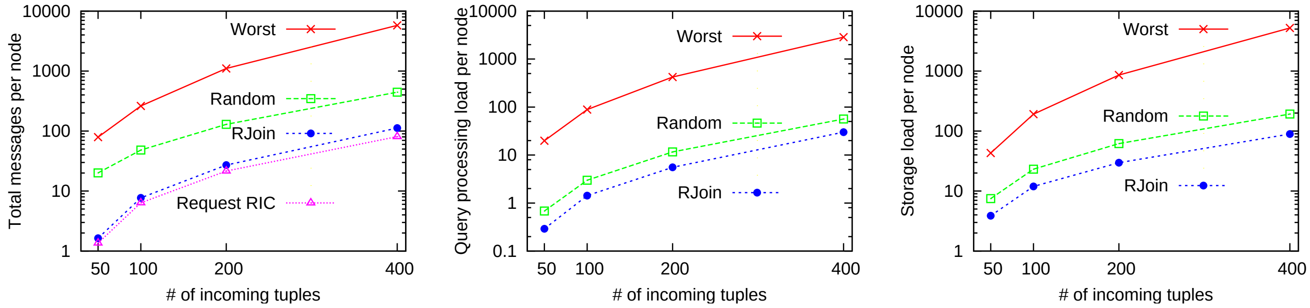
In this section, we introduce optimizations to RJoin that help reduce the network traffic requirements by efficiently reusing RIC information whenever possible.

Assume a node n_2 that receives a query q_1 from a node n_1 and creates a query q_2 by rewriting q_1 (q_1 can be an input query or a rewritten one). Node n_2 has to reindex q_2 to another node. For this reason, n_2 needs to know the RIC information for the relation-attribute pairs and relation-attribute-value triples that appear in the where clause of q_2. Observe now that since q_2 was created by rewriting q_1, these expressions in q_2 are almost the same as in q_1. The difference is that n_2 has probably introduced one or more new *relation.attribute = value* expressions in the where clause of q_2 compared to that of q_1. According to RJoin, when n_1 indexed q_1, it asked for RIC information regarding all expressions in the where clause of q_1. This information can be passed from n_1 to n_2 along with the message that transfers q_1. Then, n_2 needs to ask for RIC information only the candidate nodes of the new *relation.attribute = value* expressions which the rewriting step introduced in the where clause of q_2. Thus, we choose to always pack the IPs of available candidate nodes with a rewritten query. In this way, a new rewritten query asks for RIC information *only the new candidate* nodes. As a result, rewritten queries become very cheap to index, i.e., $k * O(logN) + 1$ hops where k is the number of new candidates. Typically, k will be equal to 1.

In addition, for each node n_1 that a node n_2 contacts for RIC information, n_2 stores locally the IP address of n_1. Locally, a node n groups all this information in a hash table, called *candidate table* (CT) so that it can exploit available IP addresses collected for all queries. Assume that n creates a rewritten query q_1 and asks node $x = Successor(Hash(R + A + v))$ for RIC information regarding expression $R.A = v$ of q_1. If r is the returned RIC information and T_r is a timestamp that denotes *when* r was learned, then n will store the pair (r, T_r) in CT using key $R{+}A{+}v$. In addition, n might receive a rewritten query that carries RIC information for the expression $R.A = v$. Then n will keep in its local CT the *most recent* RIC information. In this way, by using CT, future queries that need RIC information for $R.A = v$ can access it without having to contact x. Thus, a node n can avoid locating in $O(logN)$ hops a candidate x for a query, if x has been contacted in the past (even by a different query and from a different node). If the stored RIC information is not considered valid, e.g., if it was acquired a long time ago, then we can contact x in one hop and ask for an update.

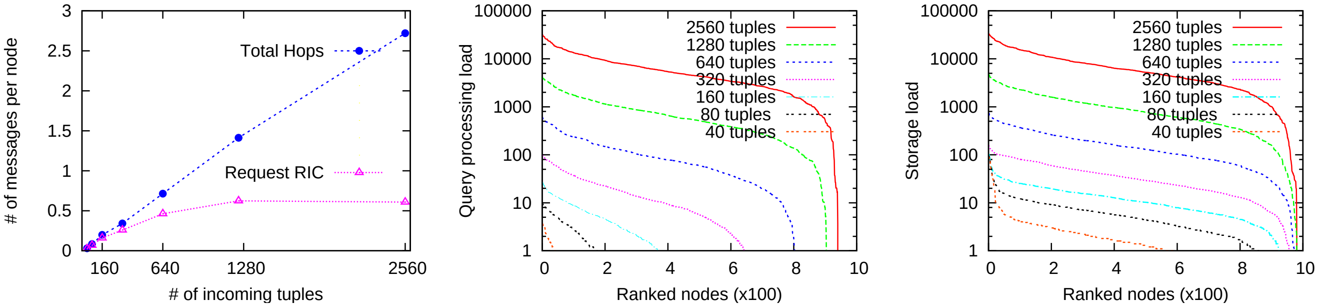
8. EXPERIMENTAL ANALYSIS

In this section, we experimentally evaluate the performance of our algorithm. Our experiments are based on a Java implementation that allows us to run multiple Chord



(a) Traffic cost (b) Query processing load (c) Storage load

Figure 2: Effect of taking into account RIC information



(a) Traffic cost per tuple (b) Query processing load distribution (c) Storage load distribution

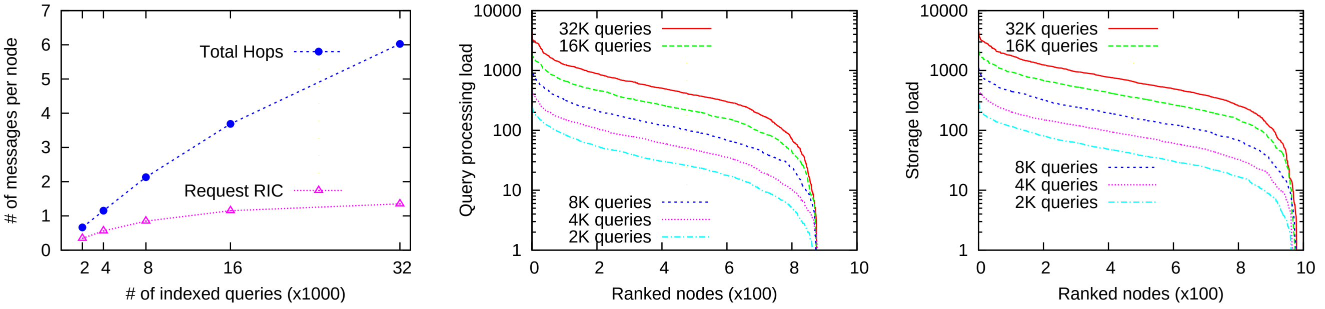
Figure 3: Effect of increasing the number of incoming tuples

nodes in one machine. Our workload is created as follows. We have a schema of 10 relations, each one with 10 attributes. Each attribute has a value range of 100 values. In order to create a new tuple, we choose one relation using a Zipf distribution, and then we assign values to all its attributes again using a Zipf distribution. Unless explicitly mentioned in an experiment, the default parameter used for our Zipf distribution is $\theta = 0.9$, thus our experimental data is highly skewed. In most cases, we experiment with 4-way joins with a where clause of the following form: $R.A = S.B$ *and* $S.C = J.F$ *and* $J.C = K.D$. An experiment that shows the effect of query complexity is also provided where we experiment with 4-way, 6-way and 8-way joins. The only restriction in the above syntax is that adjacent joins must have a common relation. Relations and attributes are chosen randomly for each query. Most of our experiments are for queries without windows since this kind of queries is the most challenging one forcing nodes to consider all past tuples and combine them with future ones. The effect (and benefits due to garbage collection) of having a "bigger" or "smaller" window on RJoin can be seen in a single experiment where the techniques of Section 5 are evaluated.

We define as *network traffic* the number of messages that a node n has to send. This includes both the messages that n creates due to RJoin, e.g., index a rewritten query to a new node, and also the messages that n has to route due to the DHT routing protocols. In most of the experiments, we measure this cost at a per tuple basis. Messages are small, since only rewritten queries are transferred, so the cost to send a message is mainly due to the initialization phase, i.e., establishing the connection with the remote node. Thus, a message has always weight 1. The *query processing load* that a node n incurs is defined as the sum of the number of rewritten queries that n receives in order to search for locally stored tuples, plus the number of tuples that n receives in order to search for matching locally stored queries. The *storage load* that a node n incurs is defined as the sum of the number of rewritten queries plus the number of tuples that n has to store locally.

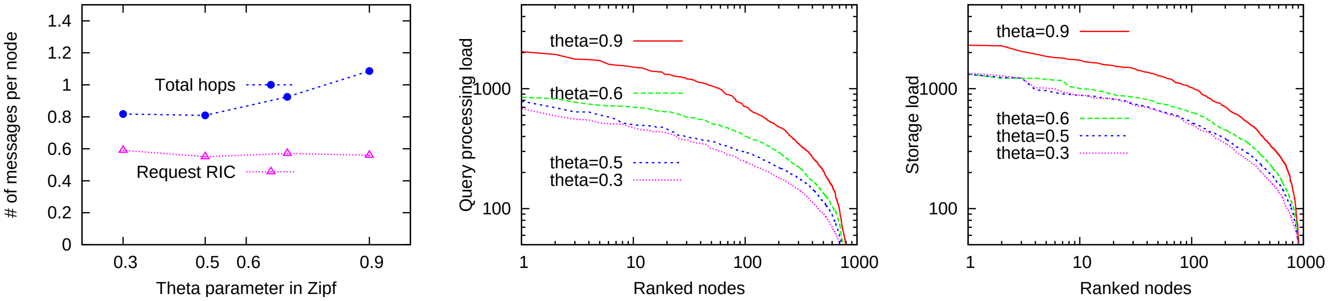
Effect of taking into account RIC information. The first experiment demonstrates how important it is to take into account the RIC information and make the right decisions while processing multi-way joins in a distributed environment. We created two variations of our query processing algorithm to compare with RJoin. The first one simulates the worst case scenario, i.e., what happens if we always make the worst choice regarding where we decide to index a query. The second one makes random choices. In a network of 10^3 nodes we insert $2 * 10^4$ queries. In Figure 2, we show what happens after 50, 100, 200 and 400 tuples have been inserted.

For all the parameters assessed, there is a clear advantage of taking into account the RIC information. For example, in Figure 2(a) we present the total traffic cost per node. The total cost of RJoin includes the cost of requesting RIC information. We also show this cost separately to give a feeling of how expensive requesting RIC information is. This cost is mainly dominated by the DHT routing of messages. Observe that the remaining cost, i.e., the cost to actually index the input/rewritten queries is small. Moreover, the fact that RIC information is packed with rewritten queries, helps us to keep the cost of requesting RIC information at low levels as more tuples are arriving. Depending on the number of incoming tuples, the difference between making the worst decision and RJoin can be many orders of magnitude (one to two in Figure 2(a)). In addition, RJoin is also 4-5 times cheaper in terms of network traffic compared to the random strategy. Indexing queries by a relation-attribute or a relation-attribute-value combination that is less often trig-



(a) Traffic cost per tuple (b) Query processing load distribution (c) Storage load distribution

Figure 4: Effect of increasing the number of indexed queries



(a) Traffic cost per tuple (b) Query processing load distribution (c) Storage load distribution

Figure 5: Effect of skewed data

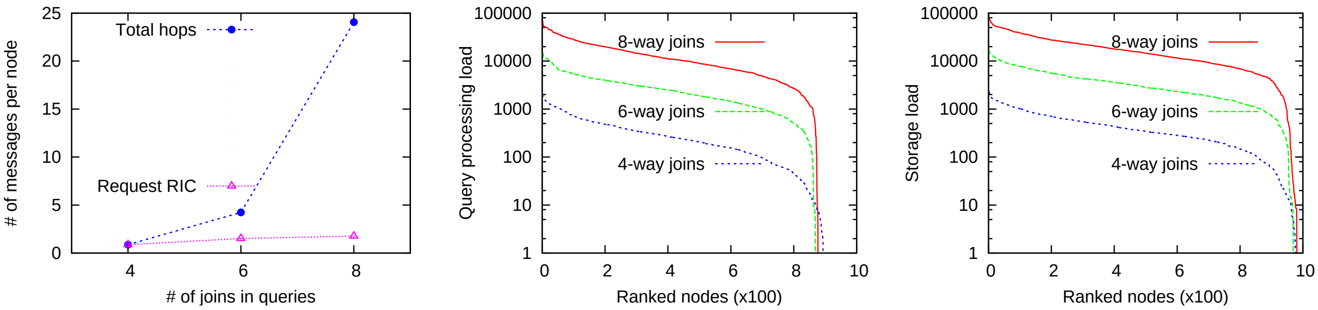
gered by incoming tuples allows us to avoid creating a huge amount of traffic because fewer queries have to be forwarded through the network to be assigned to other nodes. For the same reason, in Figures 2(b) and 2(c), we see that RJoin generates significantly less query processing and storage load.

Effect of increasing the rate of incoming tuples. Here we discuss what happens as more tuples arrive. In a network of 10^3 nodes, we insert $2 * 10^4$ queries, like in the previous experiment but now we consider a higher number of incoming tuples. Figure 3 shows the results.

Naturally, as more tuples arrive, the messages needed to handle a new tuple are increased since more queries are triggered and for different values used in the tuples, more rewritten queries are created. Initially, the network traffic for requesting RIC information increases at a high rate, but as more tuples arrive the rate of increase becomes significantly lower since RJoin can reuse knowledge about incoming tuple rates obtained in the past. Since more and more queries are triggered, the total cost is increased but due to efficient usage and caching of RIC information this increase is linear allowing RJoin to scale well with increasing number of tuples. In Figure 3(b), we see that the total query processing load is increased as more tuples arrive for the same reason as before. However, observe that although initially the load distribution is uneven, gradually, as more tuples arrive, more nodes participate to share the query processing load. Finally, after 2500 tuples we observe that almost all nodes (e.g., 940 nodes) participate in query processing. This is because more values are being used and rewritten queries spread throughout the network. Similarly, as we see in Figure 3(c), the storage load is nicely distributed.

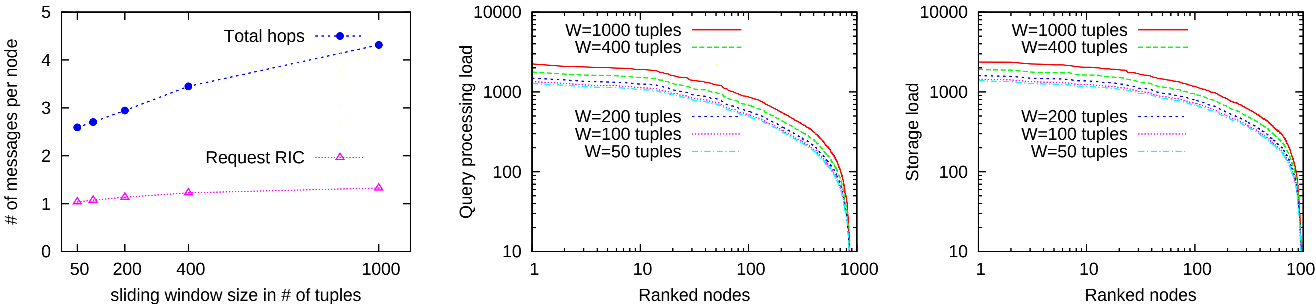
Effect of increasing the number of indexed queries. Let us now demonstrate the effect of the number of queries waiting for tuples. We set up a network of 10^3 nodes and we insert $Q = 2 * 10^3$ queries. Then, we insert 10^3 tuples and let the algorithm work. The experiment is repeated for $Q = 4 * 10^3, 8 * 10^3, 16 * 10^3, 32 * 10^3$. Figure 4 shows the results. The network traffic is of course increased, since more queries will be triggered by incoming tuples. However, as more queries are indexed, the rate of increase is smaller due to the fact that similar queries can be nicely grouped together to avoid creating extra network traffic, and RIC information is being reused. For the same reasons, we see that both query processing and storage load are increased as it is natural, but the pattern of distribution remains the same, so the extra load is again shared by multiple nodes.

Varying the skew of the data distribution. As before we use a network of 10^3 nodes where we insert $2 * 10^4$ queries. Then, we create 10^3 tuples using a Zipf distribution with $\theta = 0.3$ both for choosing the next relation and for choosing which value to assign to a new attribute. We repeat the experiment for $\theta = 0.5$, $\theta = 0.7$ and $\theta = 0.9$. The results are shown in Figure 5. We observe that the more skewed the workload is (i.e., as θ is growing), the more work is being done, i.e., all metrics are increased. This is reasonable to happen since the possibility of having joined tuples is higher if their value range is more skewed. Thus, more queries are triggered and processed. In addition, in Figures 5(b) and (c), we see that the highest loaded node incurs more load as θ is growing. This is natural since the more skewed the data distribution is, the fewer options there are to distribute the rewritten queries. However, even with a very skewed workload, RJoin manages to use a large portion of the network and balance the load well, i.e., the extra load created due to skew is nicely distributed to the nodes with the same pattern as with the less skewed workloads. Regarding network traffic, while the cost of requesting RIC information is decreasing, the total cost slightly increases with the skewness



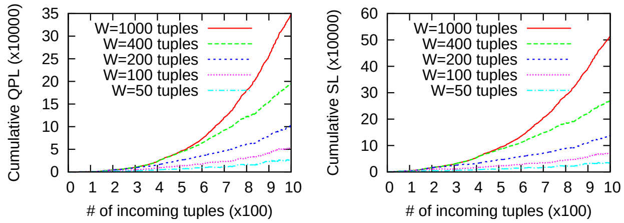
(a) Traffic cost per tuple (b) Query processing load distribution (c) Storage load distribution

Figure 6: Effect of having more complex queries



(a) Traffic cost per tuple (b) Query processing load distribution (c) Storage load distribution

Figure 7: Effect of sliding window size (W)



(a) Query processing load (b) Storage load

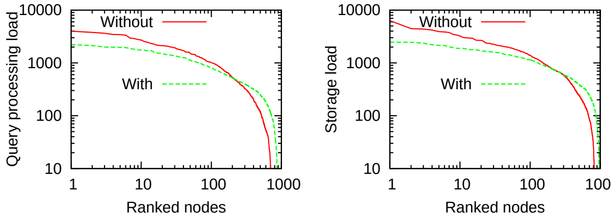
Figure 8: Cumulative load created with each new tuple depending on window size (W)

of distribution. This is due to the fact that by using more often the skewed values, more queries are triggered, rewritten and forwarded to other nodes. However, since the same values are used more often, nodes do not need to acquire new RIC information and thus this cost is decreased.

Number of joins. Here, we present the effect of the query complexity by varying the number of joins that queries contain. In a network of 10^3 nodes, we insert $2 * 10^4$ k-way join queries. Then, we insert 10^3 tuples. We repeat the experiment for $k = 4$, 6 and 8. In Figure 6, we see that naturally, more complex queries require more network traffic, processing and storage space to be evaluated. However, while queries get more complex, the extra load is shared in a similar way among the network nodes.

Sliding window size. Here we present the effect of the sliding window size W. In a network of 10^3 nodes, we insert $2 * 10^4$ 4-way join queries. For clarity of results the sliding size W is the same for all queries. Figures 7 and 8 show results for various window sizes. Figure 8 shows the total cumulative query processing and storage load created for each sliding window size as the number of tuples increases from 1 to 10^3 . With each new tuple arrival we measure the amount of query processing and storage load needed to handle this tuple. For clarity of presentation, the graphs in Figure 8 are cumulative, i.e., for each new tuple we add the load created due to this tuple to the total load (per window size). As expected, as the window size grows, more query processing and storage load is created since there are more combinations of tuples to consider within each window. With smaller window sizes rewritten queries are dropped often and thus the nodes have significantly less data to store and process. For the same reasons, we observe in Figure 7(a) an increase in the network traffic required for larger windows since more rewritten queries will be triggered and reindexed at different nodes. Finally, observe in Figures 7 (b) and (c) that the load is nicely distributed among the nodes maintaining the same pattern with all window sizes.

Using lower level interfaces. As mentioned earlier, RJoin respects the standard DHT APIs. Thus, it can take advantage of all existing (and possibly future) DHT techniques that implement various low-level optimizations. As an example, we will use in this experiment the load balancing technique of [19] which is based on allowing a node to change its position on the identifier circle. In this way, a node can also choose for which identifiers it is responsible for. This is an elegant way to offer low-level load balancing functionality which is powerful enough to also take into account individual node characteristics and cope with node heterogeneity since a node can decide for how many identifiers it will be responsible for, depending on its system resources. In this experiment, we use the technique of [19] to equally balance the responsibility for rewritten queries and tuples among the nodes. In a network of 10^3 nodes, we insert $2 * 10^4$ 4-way join queries and then a 10^3 tuples. In Figure 9, we see that id movement offers a significant improvement by removing load from the higher loaded nodes (i.e., the highest loaded



(a) QPL distribution (b) SL distribution

Figure 9: Effect of Id movement (with and without)

node had originally $4 * 10^3$ hits whereas with id movement it has only $2 * 10^3$) and allowing more nodes to participate in query processing (900 nodes participating when id movement is enabled instead of only 700). This example, nicely demonstrates that RJoin can directly use any low-level technique offered by a general DHT optimization.

9. RELATED WORK

Let us now briefly survey related work. The most related previous work is [18], where we study two-way equi-joins in DHTs. Compared to this work, we have investigated the general problem of multi-way joins and introduced algorithm RJoin with various properties not present in the proposal of [18], i.e., window joins, indexing based on environment conditions, handle network delays and duplicate answers.

Continuous queries for databases first appeared in [26]. OpenCQ [21] and NiagaraCQ [9] are examples of centralized systems that support continuous queries. Our work is also related with work on monitoring and stream processing with various centralized [22, 8, 23] and distributed proposals [11, 10, 2, 4, 5, 24, 13]. In addition, distributed and parallel databases [20] are highly related.

PeerCQ [11] is a system for continuous queries on top of DHTs. PeerCQ does not concentrate on the relational data model and the SQL query language, and assumes that data is not stored in a DHT but is kept locally at external data sources (e.g., web sources). One of the main contributions of PeerCQ is that peers are considered *heterogeneous* and a sophisticated model of peer capabilities is used to distribute the evaluation of continuous queries.

[5] considers distributed equi-join evaluation in wide-area networks consisting of many heterogeneous hosts concentrating on *network locality* and *data locality* trying to optimize the delay of output tuples. In a similar manner, [4] shows the benefits of using the locality-aware DHT Tapestry [27] to implement distributed operator placement for continuous query processing of data streams. [24] is another recent paper that considers distributed query optimization in stream overlay networks and points out differences with distributed query optimization. Finally, [13] is another recent paper that makes the case for distributed triggers in the context of wide-area monitoring applications.

10. CONCLUSIONS AND FUTURE WORK

We presented the RJoin algorithm for the evaluation of continuous multi-way joins on top of DHTs. RJoin exploits the resources of multiple DHT nodes to distribute the query processing load by dynamically creating query plans according to values used in incoming tuples. It takes into account the rate of incoming tuples to create plans that lead to a limited amount of network traffic and query processing and allow RJoin to keep distributing the load nicely among the nodes even when the workload is increased.

Our future plans regarding RJoin include developing techniques to eliminate the effect of skewed values. We are currently investigating strategies based on self-organization to further balance the load. In addition, we are working on techniques to make RJoin sensitive to environment changes, e.g., workload changes by on-line adaptation of the existing distributed query plans (by query migration) which is a way to handle updates also. Finally, we are working on lower level techniques to further optimize network traffic by allowing a node n to perform batch routing of messages that need to be routed within the same time window.

11. REFERENCES

[1] S. B. A. Arasu and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *VLDB Journal*, 15(2):121–142, June 2006.

[2] D. Abadi et al. The Design of the Borealis Stream Processing Engine. CIDR '05.

[3] K. Aberer et al. The essence of P2P: A reference architecture for overlay networks. Peer-to-Peer Computing, '05.

[4] Y. Ahmad and U. Centinemel. Network-Aware Query Processing for Stream-based Applications. VLDB '04.

[5] Y. Ahmad et al. Locality-Aware Networked Join Evaluation. NETDB '05.

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

[7] D. Carney et al. Monitoring streams - a new class of data management applications. VLDB 2002.

[8] S. Chandrasekaran and M. J. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB Journal*, 12:140–156, 2003.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD '02.

[10] M. Cherniack et al. Scalable Distributed Stream Processing. CIDR '03.

[11] B. Gedik and L. Liu. PeerCQ: A Decentralized and Self-Configuring Peer-to-Peer Information Monitoring System. ICDCS '03.

[12] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. VLDB 2003.

[13] J. M. Hellerstein et al. A Wakeup Call for Internet Monitoring Systems: The Case for Distributed Triggers. HotNets-III '04.

[14] K. Horowitz and D. Malkhi. Estimating network size from local information. *Inf. Process. Lett.*, 88(5):237–243, 2003.

[15] R. Huebsch et al. Querying the Internet with PIER. VLDB '02.

[16] R. Huebsch et al. The Architecture of PIER: an Internet-Scale Query Processor. CIDR '05.

[17] S. Idreos. Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. Master thesis. Technical University of Crete. September, 2005.

[18] S. Idreos et al. Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks. ICDE 2006.

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

[20] D. Kossman. The State of the art in Distributed Query Processing. *ACM Comput. Surveys*, 32(4):422–469, Sep 2000.

[21] L. Liu et al. Continual Queries for Internet Scale Event-Driven Information Delivery. *TKDE*, 11(4):610–628, 1999.

[22] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. SIGMOD '02.

[23] S. Chandrasekharan et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. CIDR '03.

[24] J. Shneidman et al. A Cost-Space Approach to Distributed Query Optimization in Stream Based Overlays. NETDB '05.

[25] I. Stoica et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. SIGCOMM '01.

[26] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. SIGMOD '92.

[27] B.-Y. Zhao et al. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.