Distributed Evaluation of Continuous Equi-join Queries over Large Structured Overlay Networks[*]

Stratos Idreos[†] Christos Tryfonopoulos Manolis Koubarakis

CWI
Amsterdam, The Netherlands
S.Idreos@cwi.nl

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

Abstract

We study the problem of continuous relational query processing in Internet-scale overlay networks realized by distributed hash tables. We concentrate on the case of continuous two-way equi-join queries. Joins are hard to evaluate in a distributed continuous query environment because data from more than one relations is needed, and this data is inserted in the network asynchronously. Each time a new tuple is inserted, the network nodes have to cooperate to check if this tuple can contribute to the satisfaction of a query when combined with previously inserted tuples. We propose a series of algorithms that initially index queries at network nodes using hashing. Then, they exploit the values of join attributes in incoming tuples to rewrite the given queries into simpler ones, and reindex them in the network where they might be satisfied by existing or future tuples. We present a detailed experimental evaluation in a simulated environment and we show that our algorithms are scalable, balance the storage and query processing load and keep the network traffic low.

1 Introduction

We are interested in the problem of *relational query processing* in wide-area networks such as the Internet and the Web. This is an important research area with applications to e-learning [30], P2P databases [20], monitoring [19] and stream processing [28]. We envision large P2P overlay networks where information is inserted and stored in the form of relational tuples and is queried with SQL queries. Each node keeps a fraction of the total data tuples. Tuples of a given relation can be distributed among various nodes. In this paper we concentrate on *continuous* relational query processing and present algorithms for continuous *two-way equi-join* queries. Join queries have traditionally been the study of many query optimization efforts. Distributed evaluation of join queries is very challenging, mainly due to the fact that data from different parts of the network have to be combined. We consider this paper to be our first step in contributing to the vision of relational P2P databases: a set of unified protocols for full support of SQL in P2P networks.

Current work on continuous relational queries has mostly emphasized system design and query evaluation for the centralized case [39, 26, 9, 28, 33]. Recent papers [16, 19] and the present paper study continuous relational query processing in its natural habitat dictated by target applications: distributed, Internet-scale environments realized by technologies building on *distributed hash tables (DHTs)* [36]. The only system so far that implements join algorithms on top of DHTs is PIER [20] but this is done only for the case of *one-time* queries. The case of continuous join queries is a different one and cannot be captured by the algorithms presented in [20]. PeerCQ is another interesting system proposed for continuous queries over DHTs [16]. PeerCQ does not consider the relational data model and the SQL query language, and assumes that data is not stored in a DHT but is kept locally in external data sources. In PeerCQ, the DHT infrastructure is nicely utilized to achieve a good distribution of the responsibilities for monitoring external data sources and evaluating queries. To the best of our knowledge, our paper is the first one that presents algorithms for continuous relational join queries on top of DHTs where DHT nodes are fully utilized to store data tuples and run collaborative query processing protocols.

The contributions of this paper are the following. We present four distributed algorithms for evaluating continuous two-way equi-join queries over DHTs. In our algorithms, when a node poses a continuous query, the query is indexed somewhere in the network and waits for incoming tuples. As new tuples are inserted, the network nodes cooperate to deliver a *notification* to the node that posed the

[*]This work was supported in part by the European Commission project Evergrow. Christos Tryfonopoulos is partially supported by a Ph.D. fellowship from the program Heraclitus of the Greek Ministry of Education.

[†]Work performed while the author was with the Department of Electronic and Computer Engineering, Technical University of Crete.

query. All algorithms in the paper use a *two-level indexing* mechanism to index queries and tuples. In the first level, nodes use attribute names prefixed by relation names to index a query or a tuple. In the second level, nodes utilize attribute values in order to achieve a better load distribution. The two-level indexing mechanism is exploited by a two-phase query evaluation algorithm.

The emphasis of our algorithms is twofold. We try to *distribute the load* of evaluating continuous join queries to as many nodes as possible and, at the same time, keep the cost in terms of *overlay hops* low. We show the tradeoff between achieving load distribution and performing query evaluation with as little network traffic as possible. Each algorithm we studied offers a particular way to resolve this tradeoff, and it might be appropriate for applications with relevant characteristics. One of our technical contributions is the introduction of appropriate metrics for capturing individual node load and total system load in our environment.

The challenges presented by continuous query processing should not be underestimated. When the number of installed queries increases, the total query processing load and network traffic created by incoming tuples increases as well. Similarly, when the rate of incoming tuples in a given time window increases, a higher amount of installed queries will be triggered, leading to a higher query processing load and network traffic. Our simulations show that our algorithms scale well when such events take place. Another aspect of scalability is also clearly demonstrated in our work. When the overlay network grows, query processing becomes easier since new nodes relieve other nodes by taking a portion of the existing workload.

The experiments we present use Chord [36] as the underlying DHT, due to its relative simplicity, and appropriateness for equi-join queries. However, our ideas are *DHT-agnostic:* they will work with any DHT extended with the APIs we define. Recent distributed data structure proposals such as [12, 32, 22] that can handle equality queries and range queries efficiently can also be extended to handle two-way join queries (with equality or other comparison operators) in a straightforward way using our approach.

The organization of the paper is as follows. Section 2 presents Chord and gives our assumptions regarding system and data model. Sections 3, 4, 5 and 6 discuss alternative indexing and query evaluation algorithms. In Section 7 we discuss optimizations. Section 8 presents a detailed experimental evaluation. Section 9 discusses related work and finally in Section 10 we conclude the paper.

2 System model and data model

We assume an overlay network where all nodes are equal, as they run the same software and they have the same rights and responsibilities. Nodes are organized according to the Chord DHT protocol and are assumed to have synchronized clocks. In practice, nodes will run a protocol such as NTP [6] and achieve accuracies within few milliseconds. Each node can insert data and pose continuous queries. Let us now give a short description of Chord. Each node n and item i in the network owns a unique key, denoted by $Key(n)$ and $Key(i)$ respectively. Chord uses consistent hashing to map keys to identifiers. Each node and item is assigned an m-bit identifier, that should be large enough to avoid collisions. A cryptographic hash function, such as SHA-1 or MD5 is used: function $Hash(k)$ returns the m-bit identifier of key k. Identifiers are ordered in an *identifier circle (ring)* modulo 2^m i.e., from 0 to $2^m - 1$. Key k is assigned to the first node which is equal or follows $Hash(k)$ clockwise in the identifier space. This node is called the *successor* node of identifier $Hash(k)$ and is denoted by $Successor(Hash(k))$. We will often say that this node is *responsible* for key k. A query for locating the node responsible for a key k can be done in $O(\log N)$ steps with high probability [36], where N is the number of nodes in the network. Chord is described in more detail in [36].

For this work, we have developed a simple API on top of Chord [21]. 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. Moreover, 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 happens in $d * O(logN)$ hops. Function *multiSend()* can also be 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. A detailed description and evaluation of this API can be found in [21].

In this paper data is described using the *relational data model* and is inserted in the system in the form of data tuples. As in PIER [20], different schemas can co-exist but schema mappings are not supported. Continuous queries are formed using the SQL query language. We consider the case of *two-way equi-joins* i.e., SQL queries of the form:

Select $R.A_1, \ldots, R.A_\kappa, S.B_1, \ldots, S.B_\lambda$
From R, S
Where $\alpha = \beta$

where R and S are relations with schemas $R(A_1, \ldots, A_\nu)$ and $S(B_1, \ldots, B_\mu)$, $1 \leq \kappa \leq \nu$, $1 \leq \lambda \leq \mu$ and α is an expression (e.g., arithmetic, string) involving only attributes of R and possibly constants, and β is an expression involving only attributes of S and possibly constants.

We distinguish two types of queries depending on the form of α and β. If α and β involve a single attribute of R and S (e.g., A_i and B_j respectively) and equality $\alpha = \beta$ has a unique solution over $dom(A_i) \times dom(B_j)$ then we say q is of *type* T_1. If any of α or β involve more than one attributes of R and S then we say q is of *type* T_2. We show how to evaluate such queries without first transforming them to simple equi-joins using generalized projection.

Each tuple t has a time parameter called *publication time*, denoted by $pubT(t)$, representing the time that the tuple was inserted into the system. In addition, each query q has a time parameter, called *insertion time*, denoted by $insT(q)$ that shows the creation time of q. A tuple t can trigger a query q iff $pubT(t) \geq insT(q)$ i.e., only tuples inserted after a query was posed can trigger it. Whenever the Where clause of a query is satisfied, an answer is computed and this is the *notification* sent to the query subscriber. Each query q has 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 $Key(n)$. Like [20], we assume a "best-effort" semantics for query evaluation and leave all the handling of failures, partitions etc. to the underlying DHT.

Example. Consider an e-learning network such as EDUTELLA where nodes join the network for the purposes of sharing learning material [11]. Let us assume the learning material consists of research papers that are inserted in the overlay once they are published. Each paper can be described by a set of tuples using the following simple schema:

$Document(Id, Title, Conference, AuthorId)$,
$Authors(Id, Name, Surname)$

The following query asks that its subscriber be notified whenever author Smith publishes a new paper:

Select $D.Title$, $D.Conference$
From $Document$ as D, $Authors$ as A
Where $D.AuthorId = A.Id$ and $A.Surname = Smith$

3 Two-level indexing

One of the main challenges when designing a distributed query processing algorithm is to generate as little load as possible in the network and to distribute this load fairly among existing nodes. Assume a continuous two-way join query with the join condition $R.B = S.E$. The goal is to index the query in such a way, so that when new tuples are inserted, the query and the tuples will meet to create notifications. Indexing a query amounts to storing the query at one or more nodes of the overlay. We could index queries to a globally known node or set of nodes, but this would eventually overload these nodes. In a P2P environment we want as many nodes as possible to contribute some of their resources (storage, cpu, bandwidth, etc.) for achieving the overall network functionality. The resource contribution of each node will obviously depend on its capabilities, its gains from participating in the network etc. In this paper we make the simplifying assumption that all nodes are equivalent and can contribute to query evaluation in identical ways.

We choose to index a query using identifiers that are *related* to the query. This is a useful property since a tuple that should trigger a query q, is also related to the query q, for example, they both refer to the same relation. In this way, it is easy to make an incoming tuple meet the appropriate queries without any global knowledge or broadcasting.

The difficulty with join queries such as the above is that a join condition, like the one in our example, gives us little flexibility. For example, let us consider the simpler case of continuous select-project queries with a Where clause of the form $R.B = value$. In this case, we can simply assign the query to the successor x of $Hash(R + B + value)$. We use the operator $+$ to denote the *concatenation* of string values. Relevant tuples will arrive at x in the same way, and we have to worry only for skewed values regarding load distribution. With this solution to select-project queries in mind, how do we index a query with a join condition like $R.B = S.E$? One way could be to index the query to the successor nodes x_1 and x_2 of $Hash(R)$ and $Hash(S)$ respectively. Incoming tuples could then be indexed according to their relation name, and some kind of communication is required between x_1 and x_2 to create notifications. The problem with such a solution is that the query processing load is gathered to a small subset of the set of network nodes, i.e., to as many nodes as the number of distinct relations in the schema. This means that as the network size grows, the network utilization (i.e., the percentage of nodes participating in query processing) drops. The next logical step is to also use the attribute names in the indexing scheme i.e., x_1, x_2 can be the successor nodes of $Hash(R + B)$ and $Hash(S + E)$ respectively. Now we can expect a better distribution of the query processing load but again the total number of nodes contributing to query processing is limited (bounded by the total number of attributes in the schema).

Another approach would be to index a join query according to an expression combining the two join attributes, i.e., to the successor node of $Hash(R.B + S.E)$ for our example. However, new tuples would have to reach *all* pair combinations of the attributes of different relations of the schema, to guarantee completeness. Although evaluating locally a query is now very easy since we have the two relations in one node, the main disadvantage of this method is again the fact that the number of nodes that are responsible for query processing is bounded; this time by the possible join pairs.

All the previous solutions have the disadvantage that only a subset of the set of network nodes sustain the total query processing load. As with select-project queries we would like to use the various values that the join attributes can take in order to distribute this load. However, these values are not known at the time that the query is inserted but are revealed to us as tuples arrive. Our algorithms exploit this fact by using the values in incoming tuples that trigger a query in order to distribute the query processing load.

The four algorithms we will present are based on a *two-level indexing* mechanism to index queries and tuples. In the first level (*attribute level*) nodes use the names of attributes prefixed by their relation names to index a query or a tuple. In the case of a query, those attributes are among the ones involved in the join condition. In the second level of indexing (*value level*), nodes utilize attribute values in order to achieve a better load distribution. A high level description of the indexing and query processing algorithms we present

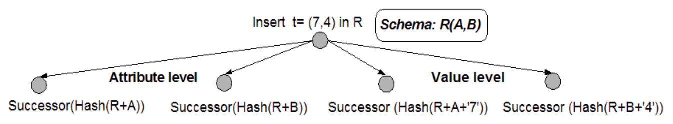

Figure 1. Inserting a tuple of a binary relation

is as follows. To pose a query q, a node indexes q at the attribute level where q is stored waiting for tuples to trigger it. When a node wants to insert a tuple, it indexes the tuple both at the attribute and at the value level. As tuples of the involved relations are inserted at the attribute level, the indexed queries are *triggered*, *rewritten* and *reindexed* at the value level according to the *values* of their join attributes in the incoming tuples. More precisely, one of the two join attributes is replaced in the join condition by its value in the incoming tuple. In this way, the join query is reduced to a simple select-project query that enters the network (reindexing) and waits to be triggered. Thus, a single join query q is evaluated by multiple nodes that share the query processing load at the value level by evaluating the multiple select-project queries that have been created from different values of the join attributes. Our algorithms result in the allocation of two *roles* to network nodes: the role of query *rewriters* and the role of query *evaluators*. A node can play both, one or none of these roles depending on the queries and tuples that are present in the network, and the node's position in the identifier space.

We continue our presentation by explaining how tuples are indexed in our proposal. Sections 4 and 5 discuss how join queries are indexed and how nodes react upon receiving a new tuple in order to trigger the appropriate queries.

3.1 Tuple indexing

Our tuple indexing protocol is a variation of *hash partitioning*. Assume a relation R over h attributes and a node x_1 that wants to insert a new tuple t. Let $\{A_1, A_2, ..., A_h\}$ be the attributes in t with values $\{v_1, v_2, ..., v_h\}$ respectively. For each A_i, x_1 computes two identifiers: $AIndex_i = Hash(R + A_i)$ and $VIndex_i = Hash(R + A_i + v_i)$. When the value of an attribute is numeric (e.g., an integer), this value is treated as a string by $+$. For each A_i, tuple t will be *indexed twice*: once according to $AIndex_i$ at the attribute level, and once according to $VIndex_i$ at the value level. Thus a set I of $2h$ identifiers is created by node x_1. For each $AIndex_i$, x creates a message AL-INDEX(t, A_i). Similarly for each $VIndex_i$, x_1 creates a message VL-INDEX(t, A_i). Attribute A_i is included in the messages so that node x_2 that receives t can tell which attribute was used to index t to x_2 (used for local processing); this attribute will be denoted by $IndexA(t)$. Finally, a set M of $2h$ messages is created and x_1 calls *multiSend(M,I)* to index t in $2h * O(logN)$ overlay hops. A complete example of inserting a tuple is shown in Figure 1.

The way a node reacts, upon receiving a tuple, depends on the algorithm and on the indexing level that the tuple was received as we will see in the following sections.

4 The single-attribute index algorithm

Let us now describe our first algorithm, the single-attribute index algorithm (SAI). To pose a query q of type T_1, a node n indexes q by one of the two join attributes at the attribute level. Node x that receives q, stores it and when tuples that trigger q arrive, x rewrites and reindexes q to nodes that are capable to create notifications at the value level.

Indexing a query at the attribute level. Indexing a query q at the attribute level proceeds as follows. First, node n *chooses one* of the join attributes of q. For the moment, we assume that this choice is random; more detailed criteria are discussed in Section 4.1. We call this attribute the *index attribute* of q and the relation that it belongs to the *index relation* of q, and denote them by $IndexA(q)$ and $IndexR(q)$ respectively. The remaining join attribute is called the *load distributing attribute* of q and its relation the *load distributing relation* of q, denoted by $DisA(q)$ and $DisR(q)$ respectively. As we will see below, the values of $DisA(q)$ will be used to distribute the query processing load generated during the evaluation of q, hence our terminology.

Then, node n creates identifier $AIndex = Hash(IndexR(q) + IndexA(q))$ and message $msg =$ QUERY$(q, Id(n), IP(n))$. Arguments $Id(n)$ and $IP(n)$ are used when delivering notifications back to n. Finally, node n calls the function *send(msg,AIndex)* to index q at the attribute level with complexity $O(logN)$.

Node $Successor(AIndex)$ that receives msg is called the *rewriter* of q. It stores q in the local *attribute-level query table* ($ALQT$) and waits for tuples to trigger it. The role of a rewriter node is *not* to compute the join itself, but to *distribute the load* of computing joins, creating notifications and delivering them. Each query has one rewriter and all queries with the same index attribute have the same rewriter.

Handling tuple insertions at the attribute level. In SAI, an incoming tuple is indexed both at the attribute and at the value level according to the protocol of Section 3.1. Assume a node x that receives a tuple t at the attribute level with the message AL-INDEX$(t, IndexA(t))$. x searches its $ALQT$ for queries that are triggered by t. The result is a set of k join queries. For each query q_i, node x owns information *on one of the two relations* needed to compute the join, namely on $IndexR(q_i)$. This information is the new tuple t. Another node has to be contacted then, where tuples of relation $DisR(q_i)$ are stored or are expected to arrive. Since q_i is an equi-join query, the only suitable tuples are the ones where the value of $DisA(q_i)$ satisfies the join condition of q_i after $IndexA(q_i)$ has been replaced with its value in t. If $valDA(q_i, t)$ is that value, then the successor node of $VIndex(q_i) = Hash(DisR(q_i) + DisA(q_i) + valDA(q_i, t))$ has the rest of the tuples needed to evaluate the join due to how tuples are indexed at the value level (see Section 3.1).

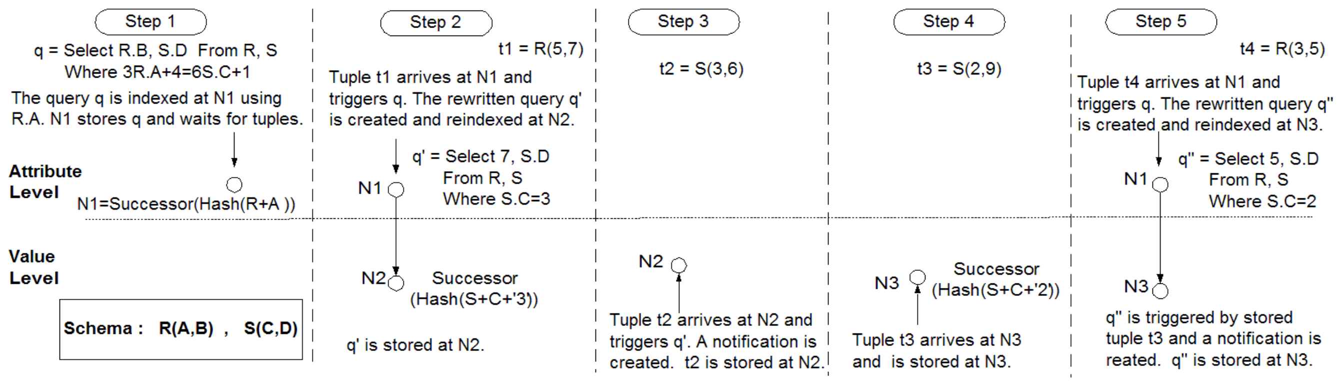
Step 1
q = Select R.B, S.D From R, S
Where 3R.A+4=6S.C+1
The query q is indexed at N1 using R.A. N1 stores q and waits for tuples.
Attribute Level
N1=Successor(Hash(R+A))
Value Level
Schema : R(A,B) , S(C,D)
Step 2
t1 = R(5,7)
Tuple t1 arrives at N1 and triggers q. The rewritten query q' is created and reindexed at N2.
N1
q' = Select 7, S.D From R, S Where S.C=3
N2
Successor (Hash(S+C+'3'))
q' is stored at N2.
Step 3
t2 = S(3,6)
N2
Tuple t2 arrives at N2 and triggers q'. A notification is created. t2 is stored at N2.
Step 4
t3 = S(2,9)
N3
Successor (Hash(S+C+'2'))
Tuple t3 arrives at N3 and is stored at N3.
Step 5
t4 = R(3,5)
Tuple t4 arrives at N1 and triggers q. The rewritten query q'' is created and reindexed at N3.
N1
q'' = Select 5, S.D From R, S Where S.C=2
N3
q'' is triggered by stored tuple t3 and a notification is reated. q'' is stored at N3.

Figure 2. An example with SAI

We call this node the *evaluator* of the query *for the value* $valDA(q_i,t)$. A query q has as many evaluators, as the *distinct values* of attribute $IndexA(q)$.

Let us now discuss what a rewriter sends to an evaluator. Each query q_i is *rewritten* according to the incoming tuple t. The resulting query q_i' will produce the same notifications, when sent to an evaluator, as if t and q_i had both arrived there. To create q_i', each attribute of $IndexR(q_i)$ in the Select and Where clause of q_i, is replaced by its corresponding value in t. Assume the query $Select\ R.A,\ S.B\ From\ R,\ S\ Where\ R.C = S.C$ which is triggered at the attribute level by a tuple $S(3,4,7)$. The rewritten query will be $Select\ R.A, 4\ From\ R\ Where\ R.C = 7$. Thus, the original query is reduced to a simple select-project query which will be send (reindexed) at the $Successor(Hash(R+C+'7'))$.

In this way, the rewriter node x rewrites all k triggered queries and for each rewritten query q_i' it creates a message JOIN(q_i'). A set M of k messages and a set I of k $VIndex$ identifiers are created. Node x calls the *multisend(M,I)* function to reindex the rewritten queries at the value level which costs $k * O(logN)$ overlay hops.

Processing rewritten queries at the value level. We will now discuss how a node x at the value level reacts upon receiving a JOIN(q') message. Assume that q' was created by query q when tuple t of relation $IndexR(q)$ arrived. First, x has to check whether it locally stores any matching tuples of $DisR(q)$, i.e., tuples that were inserted in the network after q. In addition, node x has to remember the fact that q' arrived in order to be able to create notifications in the future, when tuples of $DisR(q)$ arrive. Thus, x stores q' in its *value-level query table* ($VLQT$). This last step is necessary only if this is the first time that x receives q'. This can be easily determined using unique keys for queries [21].

An evaluator x may create one or more notifications and use either the *send()* or *multisend()* function respectively to deliver them. A notification contains the results of a triggered query, namely the appropriate tuples (projected if necessary) along with time information about when those tuples were inserted in the network. In [21] we also present techniques on how a node can retrieve its notification if it is off-line at the time that the notification is created.

Handling tuple insertions at the value level. Let us now see what happens as tuples arrive at the value level where they meet rewritten queries. Assume a node x that receives a tuple at the value level with a message VL-INDEX($t, IndexA(t)$). x checks if there is any rewritten query q' in its $VLQT$ that is triggered by the new tuple. For each triggered query a notification is created and t is also stored in the local *value-level tuple table* ($VLTT$). Storing tuples at the value level is necessary for the completeness of SAI, e.g., assume the following series of events: (a) a query q is indexed, (b) a tuple t of $DisR(q)$ is inserted and stored at node x (at the value level), and (c) a tuple of $IndexR(q)$ is inserted causing query q to be rewritten and reindexed to x. If t is not stored at x then a notification will be lost.

A complete example with SAI is shown in Figure 2. Events take place from left to right, i.e., initially query q is indexed and then tuples arrive. For readability, only the steps that affect query q are shown. Notice that while in Step 3 a notification is created by a tuple that meets a rewritten query at the value level, in Step 5 the opposite happens.

Local query indexing and grouping. Since a large number of queries are expected to be similar, i.e., reference the same relations, all queries that have equivalent join condition are *grouped* together at each node. Equivalence is easy to determine during parsing for queries of type T_1. Grouping queries is useful for minimizing the local computation cost and the network cost. Similar queries are triggered in a single step. In addition, reindexing can also be done with only one message since for the same incoming tuple all similar queries will require the same evaluator.

Locally tuples and queries are stored in hash table based data structures that are described in detail in [21]. Here we concentrate on the distributed nature of our algorithms.

4.1 Choosing the index attribute

Let us now discuss parameters that affect the choice of the index attribute. This choice determines which node will be the rewriter and which nodes will be the evaluators of a query. We can see this choice from two different perspectives with the following corresponding performance metrics that are affected: (a) the total network traffic and (b) the distribution of load among evaluator nodes.

Network traffic. A rewriter of a query q rewrites and reindexes q *each time a tuple of relation* $IndexR(q)$ *is inserted*. Thus, by indexing a query according to the attribute that belongs to the relation with the *lowest rate of tuple arrival*, we will decrease network traffic since less queries will be triggered, rewritten and reindexed. It is easy to find and maintain this information. Each node x can keep track of the total number of tuples that have arrived to x in the last time window. Then, any node can simply ask the two possible rewriter nodes before indexing a query for the rate that tuples arrive. In this way, the decision of where to index a query is *adapted* to the data already collected by the appropriate rewriters when a query is inserted. The same observation stands for queries that are highly selective, i.e., SQL queries with a Where clause which contains a join condition conjoined with a highly selective predicate (e.g., $R.A = S.B \wedge S.C = 10$). In this case, nodes should also keep track of the values of attributes as tuples arrive.

Distribution of load among rewriter nodes. A join attribute with a highly skewed value distribution will result in loading a small portion of the evaluators of the query. Thus, when distribution of load is important, the join attribute with the more uniform distribution should be chosen.

The two metrics mentioned earlier are mutually independent. In our experiments, where we assume a highly skewed distribution for all attributes, we use the first metric.

5 The double-attribute index algorithms

In this section we introduce the double-attribute index (DAI) algorithms. The motivation is to achieve an even better distribution of the query processing load. In SAI rewriter nodes distribute the query processing load by assigning rewritten queries to a multitude of evaluators. In the DAI algorithms we go even further and take advantage of the possibility of indexing a query *twice* at the attribute level, once for each join attribute. This leads to having *two* rewriters per query and thus a better load distribution than in SAI where there is only one rewriter per query.

The DAI algorithms are based on the same two-level indexing principle of SAI. But here there is a difference! If we evaluate the rewritten queries exactly as in SAI, we will end up creating *duplicate notifications* because there are two rewriters per input query. In Figure 3 we give an example of this situation. In Step 3, the same notification is created twice: once when query q'' is reindexed and once when tuple $t2$ arrives at node $N3$. Thus, to avoid creating duplicate notifications, we have a *choice* to make at the value level. Will evaluators create notifications when they receive rewritten queries or when they receive new tuples? We present two alternative algorithms (one for each option):

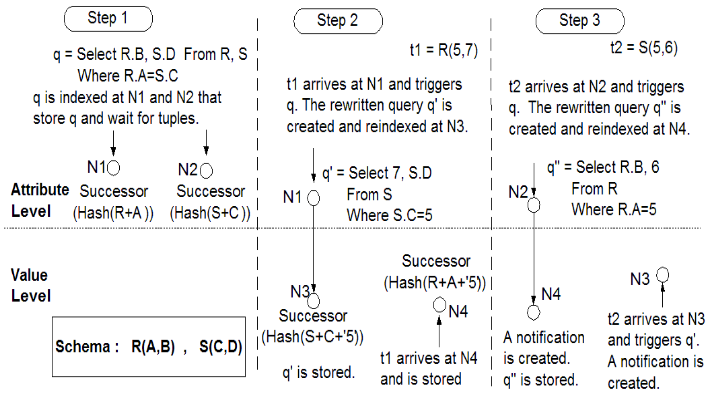
Step 1
q = Select R.B, S.D From R, S
Where R.A=S.C
q is indexed at N1 and N2 that store q and wait for tuples.
N1
N2
Attribute Level
Successor (Hash(R+A))
Successor (Hash(S+C))
Value Level
Schema : R(A,B) , S(C,D)
Step 2
t1 = R(5,7)
t1 arrives at N1 and triggers q. The rewritten query q' is created and reindexed at N3.
q' = Select 7, S.D
From S
Where S.C=5
N1
N3
Successor (Hash(S+C+'5'))
q' is stored.
Successor (Hash(R+A+'5'))
N4
t1 arrives at N4 and is stored
Step 3
t2 = S(5,6)
t2 arrives at N2 and triggers q. The rewritten query q'' is created and reindexed at N4.
q'' = Select R.B, 6
From R
Where R.A=5
N2
N4
A notification is created.
q'' is stored.
N3
t2 arrives at N3 and triggers q'. A notification is created.

Figure 3. Duplicate notifications

the DAI algorithm where notifications are created by evaluators when rewritten *queries* arrive (DAI-Q), and the DAI algorithm where notifications are created at evaluators when *tuples* arrive (DAI-T).

Common steps in all DAI algorithms. Upon insertion, a query is indexed twice at the attribute level. For example, consider a query q with the join condition $R.B = S.E$. The query is indexed once with $R.B$ and once with $S.E$ as index attribute to the successor nodes of $Hash(R+B)$ and $Hash(S+E)$ respectively. This takes place using the multi-send() function in $2*O(logN)$ hops.

We will use the notation q_L (respectively q_R) to refer to a query q when it is indexed with respect to the left (respectively right) attribute of a join condition. Using our notation, we now have the following equalities:

$$DisR(q_L) = IndexR(q_R),\ DisR(q_R) = IndexR(q_L),$$
$$DisA(q_L) = IndexA(q_R) \text{ and } DisA(q_R) = IndexA(q_L)$$

In all DAI algorithms, new tuples are indexed both at the attribute and at the value level as in SAI. Similarly, an indexed query is triggered, rewritten and has its evaluator computed at the attribute level exactly as in SAI. The rest of the query processing algorithm (i.e., how a rewritten query is processed at evaluator nodes, how evaluators react upon receiving tuples at the value level, etc.) is different for algorithms DAI-Q and DAI-T and is discussed below.

The DAI-Q algorithm. In DAI-Q, once an evaluator node receives a rewritten query, it tries to evaluate it against locally indexed data tuples and create notifications. An evaluator does *not* store the rewritten queries that it receives since incoming tuples will not try to create notifications. On the contrary, when an evaluator receives a new tuple at the value level, it *stores* it locally so that it is available when rewritten queries arrive, but it does not try to create any notifications (there are no stored rewritten queries).

The DAI-T algorithm. In DAI-T, notifications are created when evaluators receive tuples at the value level. Thus, in contrast with DAI-Q, evaluators do not need to store tuples but need to store rewritten queries. An important mo-

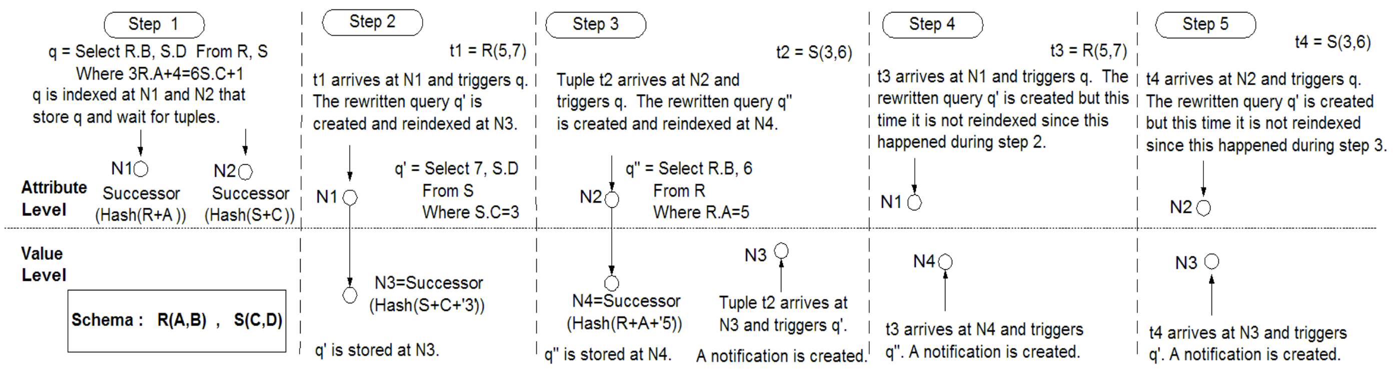
Step 1
q = Select R.B, S.D From R, S Where 3R.A+4=6S.C+1
q is indexed at N1 and N2 that store q and wait for tuples.
Attribute Level
N1 Successor (Hash(R+A))
N2 Successor (Hash(S+C))
Value Level
Schema : R(A,B) , S(C,D)
Step 2
t1 = R(5,7)
t1 arrives at N1 and triggers q. The rewritten query q' is created and reindexed at N3.
q' = Select 7, S.D From S Where S.C=3
N3=Successor (Hash(S+C+'3'))
q' is stored at N3.
Step 3
t2 = S(3,6)
Tuple t2 arrives at N2 and triggers q. The rewritten query q'' is created and reindexed at N4.
q'' = Select R.B, 6 From R Where R.A=5
N4=Successor (Hash(R+A+'5'))
q'' is stored at N4.
Tuple t2 arrives at N3 and triggers q'.
A notification is created.
Step 4
t3 = R(5,7)
t3 arrives at N1 and triggers q. The rewritten query q' is created but this time it is not reindexed since this happened during step 2.
t3 arrives at N4 and triggers q''. A notification is created.
Step 5
t4 = S(3,6)
t4 arrives at N2 and triggers q. The rewritten query q' is created but this time it is not reindexed since this happened during step 3.
t4 arrives at N3 and triggers q'. A notification is created.

Figure 4. An example with DAI-T

tivation behind DAI-T is that since rewritten queries are stored at evaluators, a rewriter does *not* need to reindex the same rewritten query *more than once* at the value level. This results in a huge performance gain for DAI-T compared to the rest of the algorithms, since after the rewritten queries (for a given input query) have been distributed to the appropriate evaluators, no intercommunication is needed between the attribute and value level. This leads not only to a *decrease in the total network traffic* but also to a significant *decrease in the total query processing load* that is created when evaluators receive and process rewritten queries.

A complete example of DAI-T in operation is shown in Figure 4. Observe that when similar tuples are inserted (after Step 3), notifications are created without extra messages except the ones used to index a tuple. Moreover, compared to SAI, the notifications are created by $N3$ and $N4$, whereas in SAI only $N3$ or only $N4$ would create the notifications depending on what index attribute has been chosen.

6 The DAI-V algorithm

The algorithms presented so far are capable of processing queries of type T_1 but not queries of type T_2. Let us see why by considering the following query q :

Select $R.A, S.D$
From R, S
Where $4*R.B+R.C+8=5*S.E+S.D*S.F$

In queries of type T_2 such as q, we have multiple candidates for the role of the index attribute. Assuming that the choice of index attribute is made randomly, let us consider what happens when q is triggered by a tuple at the attribute level. Unlike queries of type T_1, queries of type T_2 give rise to rewritten queries with an arbitrary equality in the Where clause e.g., the equality $5*S.E+S.D*S.F=25$ if a tuple t of R with values $R.B=4$ and $R.C=9$ is inserted and triggers query q. Indexing of such linear equalities can be done using geometric data structures but, in general, queries of type T_2 will contain arbitrary functions so geometric data structures is not an option we would like to consider further. Instead, we introduce a new double-attribute indexing algorithm that is different from previous DAI algorithms in how rewriters create the $VIndex$ identifiers that lead to evaluators. This algorithm has been especially designed for queries of type T_2 and covers queries of type T_1 as well. Now $VIndex$ identifier creation is based on the *value* that the left- or right-hand side of the join condition takes when a trigerring tuple arrives at the attribute level. Thus, our new algorithm is denoted by the acronym DAI-V.

Let q be a query on relations R_1 and R_2 indexed using attribute $IndexR(q_L)$ of relation R_1 and attribute $IndexR(q_R)$ of relation R_2. Rewriters x_1 and x_2 respectively receive the query. In DAI-V tuples are indexed *only* at the attribute level. When a tuple t_1 of R_1 arrives at rewriter node x_1, q_L is triggered. Then x_1 creates the identifier $VIndex(q_L) = valJC(q_L,t_1)$, where $valJC(q_L,t_1)$ is the value that is computed by substituting values from the tuple t_1 in the attribute expression appearing in the R_1 part of the join condition. For our example query, $valJC(q_L,t_1)=25$ when a tuple t_1 of R with values $R.B=4$ and $R.C=9$ is inserted. The corresponding evaluator is $x=Successor(Hash(VIndex(q_L)))$. After computing $VIndex$, a message JOIN(q'_L,t'_1) is created by x_1 and sent to the evaluator node, where q'_L is the rewritten q_L and t'_1 is the projection of t on the attributes needed for the evaluation of the join. Once an evaluator receives a JOIN message, it matches the rewritten query against the locally stored data tuples to create notifications, and then *stores* t'_1 locally. The rewritten query is *not* stored. Similarly, a future tuple t_2 of relation R_2, will arrive at x_2 where it triggers q_R and q'_R is created. The evaluator is the successor node of $VIndex(q_R)=valJC(q_R,t_2)$. When $valJC(q_R,t_2)=valJC(q_L,t_1)$, then we get to the same evaluator node where t'_1 has been stored. There, q'_R meets t'_1 and a notification is created.

DAI-V uses only values to reindex rewritten queries. Thus, we expect that the previous algorithms that use values prefixed with join attribute names will distribute better the query processing load. On the other hand, for the same reason, DAI-V is expected to create less traffic since queries can be grouped more easily.

7 Optimizations

In this section we present optimizations that enable us to decrease network traffic and achieve better load balancing. The techniques presented are applicable to all algorithms.

The join fingers routing table. We introduce the *join fingers routing* table ($JFRT$) in order to make the cost of inserting a new tuple and evaluating queries less expensive in terms of overlay hops. This cost is $c1 + c2$ for each attribute of a new tuple where $c1$ is the cost to index a tuple, namely $c1 = O(logN)$ for DAI-KV and $c1 = 2 * O(logN)$ for the other algorithms. The term $c2 = e * O(logN)$ is the cost to distribute the rewritten queries from a rewriter to their evaluators and e is the number of distinct combinations of load distributing attributes and join conditions in the triggered queries; thus this is the cost to reach the evaluators that compute the joins. $c2$ is the largest part of the cost $c1 + c2$ and we can reduce it down to e in the following way. Each time a rewriter x communicates with a new evaluator n, it saves $IP(n)$ and the $VIndex$ identifier that leads to n, in the local $JFRT$ which is a hash table that uses the $VIndex$ identifiers as keys. Each entry for an identifier id contains the IP address of the $Successor(id)$. The next time the rewriter needs to reindex a query with the same $VIndex$, it can do it in one hop. This way, the cost becomes $c1 + f + (e - f) * O(logN)$, where f are the evaluators found in $JFRT$ and can be reached in one hop. The term $(e - f) * O(logN)$ represents the cost to reach the evaluators not found in the routing table. Ideally this cost will be reduced down to $c1 + f$ if $e = f$ and will remain almost constant as the network size N grows.

Balancing the load at the attribute level. We observe that the nodes at the attribute level get more hits than those at the value level. For example, a request to index a query under $R.B$ will appear more often than a request to reindex a query under $R.B + v$, where v is a value that $R.B$ can take. For a database schema of k relations where each relation r_i has a_i attributes there will be at most $\sum_{i=1}^{k} a_i$ rewriter nodes. We can distinguish two types of load that a (rewriter) node suffers at the attribute level: the *rewriter storage* (*RS*) load and the *rewriter filtering* (*RF*) load. The *RS* load of a node n is defined as the *total number of queries* that are indexed to n. The more queries a rewriter has, the more effort it has to put into rewriting and reindexing operations. The *RF* load of a node n is defined as the *total number of tuples* that n receives at the attribute level in a time window. The more tuples a rewriter receives, the more times it has to search its $ALQT$ to trigger, rewrite and reindex queries.

We can significantly improve load distribution at the attribute level through replication of queries. We will use *DR* to denote the *degree of replication* ($DR \geq 1$). For example, if $DR = 3$ then when a node indexes a query q at the attribute level under $R.B$, instead of indexing q only according to the identifier $rid_1 = Hash(s)$, where $s = R + B$, it also indexes q according to the identifiers $rid_2 = Hash(s + s)$ and $rid_3 = Hash(s + s + s)$. rid_1, rid_2 and rid_3 are called *replication identifiers* with successor nodes n_1, n_2 and n_3 respectively. In this way, the query is replicated *DR* times and instead of having one rewriter it has *DR*. Then, when any node x wants to index a tuple t of R at the attribute level under $R.B$, instead of sending t directly to n_1, according to the protocol of Section 3.1, x chooses randomly among n_1, n_2 and n_3. Thus, n_1, n_2 and n_3 *share* the *RF* load that initially only n_1 suffered while all of them suffer the *same RS* load. In the absence of collisions there will be at most $DR * \sum_{i=1}^{k} a_i$ rewriter nodes and distinct replication identifiers. The cost we pay is more overlay hops when indexing queries at the attribute level, and more storage load at the network. In both cases costs are raised by a factor of *DR*.

A single node can become responsible for more than one replication identifiers of the same query [21]. We overcome this problem by allowing each node to be responsible *for at most z* replication identifiers in the spirit of [23], i.e., rewriters will change their identifiers, namely their position on the identifier circle. In our experiments we use $z = 1$.

8 Experiments

In this section we experimentally evaluate the performance of our algorithms. We implemented a Chord simulator in Java on top of which we developed our algorithms. We synthetically create tuples and queries as follows. We assume a database schema S that consists of 50 relations numbered from 1 to 50. This is a likely scenario in an Internet-wide setting with a multitude of information sources (having a smaller number of relations does not affect our techniques or results in any way). Each relation consists of 10 attributes. Each attribute A_j of a relation r_i takes values from the domain $\{1, 2, ..., 10^4\}$. There are two classes of relations, the *small* and the *big* ones. Big relations are used to model relations with a higher rate of tuple arrivals than small ones. Unless stated otherwise, the ratio between the arrival rate of tuples of big and small relations, denoted as *bos* (big over small) is 10. In order to create a tuple of a relation in the small class, we choose randomly a relation between 1 and 25 and we assign values to its attributes. The values of attributes are skewed with a Zipf distribution of $\theta = 0.9$. For the relations of the big class, we do the same with relations 26 to 50. In our experiments, we create queries of type T_1 as follows. We randomly select one relation from the big class and one relation from the small class. Then we randomly select two attributes, one from each relation, to be the join attributes.

E1: Network traffic and JFRT effect. In our first experiment we compare all algorithms in terms of overlay hops they need and demonstrate the effect of $JFRT$ as the network is being trained with tuple insertions. We set up this experiment as follows. We create a network of 10^4 nodes and install 10^5 queries. Then we train $JFRTs$ with

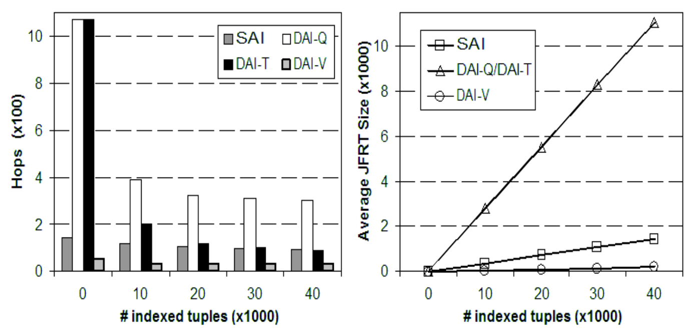

Figure 5. (E1) Traffic cost and $JFRT$ effect

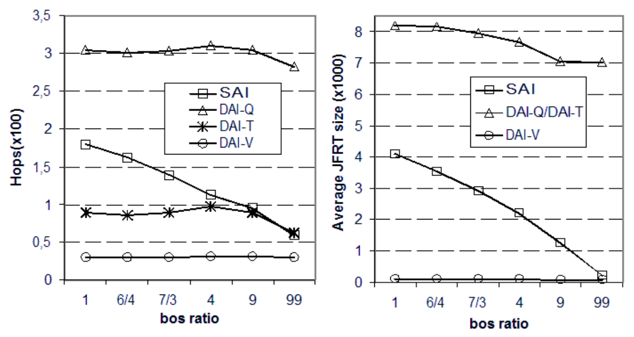

Figure 6. (E2) Effect of the bos ratio

a varying number of incoming tuples. After each training phase, we insert another 1100 tuples (100 from the small class and 1000 from the big one) and count (a) the average number of overlay hops needed to index one tuple and evaluate existing queries and (b) the average size of $JFRT$s. To count $JFRT$ size, the sum of the size of all $JFRT$s in the network is averaged by the number of rewriter nodes. For our schema, we have 500 rewriters. Note that DAI-Q and DAI-T algorithms have the same $JFRT$ size after having received the same tuples in a given network, due to having the same query indexing steps at the attribute level and that they compute the evaluators in the same way.

The results are presented in Figure 5. The number of hops is decreasing, as the number of indexed tuples increases because more queries are triggered, rewritten and reindexed, which makes the $JFRT$ on each rewriter node to store more information and be able to decrease the cost of the next tuple insertion. The point 0 on the x-axis has the highest cost, since it represents the cost to insert a tuple when the $JFRT$s are empty. We observe that the cost is reduced more quickly during the first tuple insertions, to reach a state where additional $JFRT$ training causes only a small reduction in message cost while at the same time the average $JFRT$ size keeps growing. This means that a node can *stop training* its $JFRT$ after this point and retrain it periodically. Initially, SAI has a cost lower by a factor of 7 compared to DAI-Q and DAI-T (point 0), because in SAI queries are indexed only under attributes of the small relations. But as more tuples are inserted, SAI's advantage is diminished since the $JFRT$s are trained with a smaller pace because less queries are triggered (at the attribute level). DAI-T has an advantage since the rewriters do not reindex the same rewritten query more than once. The average $JFRT$ size in SAI is a lot smaller than that in DAI-Q and DAI-T since in SAI a rewriter has less queries, and queries (at the attribute level) in SAI are triggered only by relations of the small class. Finally, algorithm DAI-V has even lower requirements regarding network traffic and $JFRT$ (by a factor of 3 compared to SAI). This is due to how rewriters in DAI-V reindex triggered queries where only the required value is used to calculate the *Vindex* identifier without using attribute names. In this way, it is possible to group more triggered queries that may not have exactly the same join conditions and use only one message to reindex them.

Experiments with uniform distribution for the values of attributes lead to similar results, except that for all algorithms the decrease rate in number of hops was smaller (i.e., we needed longer training phases) and $JFRT$s were larger by a factor of 4. Experiments where the query grouping features where not activated lead to a much higher network traffic cost for all algorithms since multiple messages where sent to the same (or towards the same destination). For example, in order to filter one tuple with algorithm DAI-Q, without grouping we needed on average 7 times more messages. Finally, experiments where we increased the number of indexed queries showed that the algorithms are scalable to such changes. For example, SAI and DAI-T need only 10 more hops on average to filter one tuple with $128K$ indexed queries than with $8K$ queries. These experiments can be found in detail in [21].

E2: Effect of the bos ratio. In this experiment we measure the effect in network traffic and in JFRT storage cost of the ratio between the number of tuples in big and small relations. We set up this experiment as follows. We create a network of 10^4 nodes where we insert 10^5 queries, and then $40K$ tuples to train the $JFRTs$. Then we count the number of hops needed to insert one tuple and evaluate all existing queries and the average size of $JFRT$s. We repeat this procedure for various bos ratios from $bos = 1$ up to $bos = 99$.

The results are shown in Figure 6. DAI-Q and DAI-V are not affected significantly as bos increases. However, when the bos ratio becomes 9 or bigger they show a decrease in hops and $JFRT$ size since similar tuples arrive all the time which means that there is no need to add new entries in the $JFRT$s while at the same time existing entries are used more often. With $bos = 1$, DAI-T has a clear advantage over SAI in terms of hops since in DAI-T the same rewritten query is not reindexed more than once. As expected, SAI reduces significantly the necessary hops as bos increases since tuples of the small relation arrive with a lower frequency so less queries are triggered. As bos becomes higher than 9, SAI and DAI-T perform similarly. However, DAI-T is far

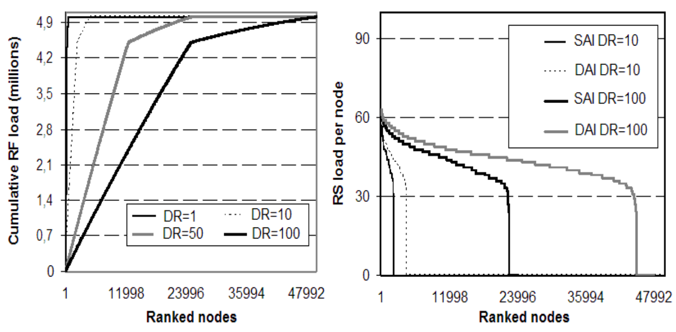

Figure 7. (E3) Effect of the replication scheme in filtering and storage load distribution

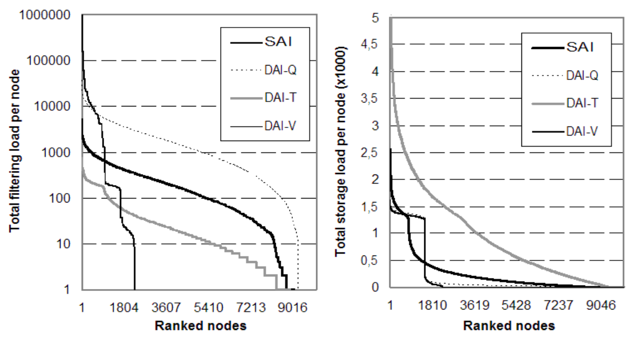

Figure 8. (E4) Total filtering and total storage load distribution

more expensive in terms of $JFRT$ needs.

E3: Evaluation of the replication scheme. In the third experiment we demonstrate the effect of our replication scheme. In a network of 10^5 nodes we insert $5 * 10^5$ tuples and count the RF and RS load per rewriter node. We do that for various DR values.

In Figure 7(a) we measure the RF load. Note that the RF load is independent of the algorithm used. On the x-axis of Figure 7(a) nodes are ranked starting from the node with the highest filtering load. The y-axis represents the cumulative filtering load, i.e, each point (a,b) in the graph represents the sum of filtering load b for the a most loaded rewriters. As DR increases, a higher number of rewriters share the same total filtering load. In addition, as DR grows the heaviest node suffers less load. Figure 7(b) shows the RS load per rewriter for $DR = 10$ and $DR = 100$. The DAI algorithms have the same RS load per rewriter, for the same set of queries, since they all index queries at the attribute level in the same way. The extra storage load created by the query replicas is distributed to new nodes, namely the ones that take part of the filtering load. The results we showed are when moving identifiers of rewriters to force them be responsible for at most one replica of the same query. If we do not do that then the benefits of replication are not that strong, e.g., for the same experiment with $DR = 100$ there were 12K less nodes sharing the filtering load [21].

Since indexing of tuples at the attribute level is not affected by the values that attributes take, these results hold for any value distribution. Also, experiments with a lower *bos* ratio naturally resulted in a better filtering load distribution without affecting the storage load distribution.

E4: Load distribution. In this experiment we compare the algorithms in terms of load distribution. First, we introduce new metrics to quantify load at the value level: the *evaluator filtering load* (or EF *load*) and the *evaluator storage load* (or ES *load*). For a node n, EF load is the sum of two quantities: the number of rewritten queries that arrive at n and are checked to see whether they match any stored tuples, plus the number of tuples that arrive at n and are checked to see whether they satisfy any stored rewritten queries. Similarly, the ES load of n is the sum of two quantities: the number of rewritten queries plus the number of tuples stored at n. We also define the *total filtering load* (or TF *load*) of a node n as the sum of the RF load and the EF load of n. Similarly, we define the *total storage load* (or TS *load*) of a node n as the sum of the RS load and the ES load of n.

We set up this experiment as follows. We create a network of 10^4 where we insert 10^5 queries. Then we insert $5 * 10^4$ tuples and we count the total TF and TS load incurred by each node for each different algorithm. The degree of replication is $DR = 10$ while $bos = 10$.

In Figure 8(a) we see the TF load distribution. DAI-V behaves a lot differently because rewritten queries are reindexed by using only the values of join attributes. The rest of the algorithms that also use the name of the load distributing attribute of a query manage to force more nodes to take part in the query processing procedure. DAI-Q and DAI-T use a similar portion of the network for query processing. However, DAI-Q loads the nodes with more load by a factor of 100. This is because in DAI-T the evaluator nodes perform filtering operations only upon receiving a tuple at the value level while in DAI-Q evaluators perform filtering operations upon receiving rewritten queries at the value level which happens more often. In SAI evaluators perform filtering operations both upon receiving a tuple and a rewritten query. However, at the attribute level queries are only triggered by tuples of the small relations so this is why SAI loads the nodes with more load than DAI-T but with less load than DAI-Q. Thus, clearly DAI-T outperforms the others by using a large portion of the overlay and loading it with less load. The behavior of the algorithms regarding TS load distribution in Figure 8(b) is explained by the same reasons. In this case DAI-Q outperforms DAI-T.

We also experimented with an alternative version of DAI-V where $VIndex(q_L) = Key(q) + valJC(q_L, t)$. This allows DAI-V to have as good TF load distribution as the rest of the algorithms. However, it also creates large amounts of

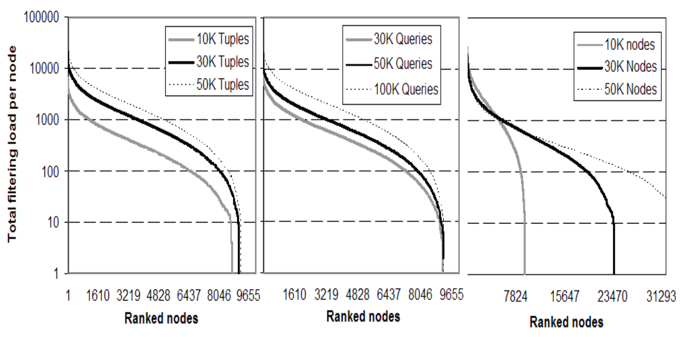

Figure 9. (E5) Effect of increasing the rate of incoming tuples, the number of indexed queries and the network size in filtering load distribution of DAI-Q

network traffic depending on the number of indexed queries, since a rewriter would have to reindex each triggered query to a different evaluator. Experiments in a 10^4 node network with 10^5 indexed queries showed an increase to network traffic approximately by a factor of 250.

E5: Parameters that affect load distribution. In this experiment we show how various parameters affect the load distribution. We set-up this experiment as the previous one and observe what happens in the load distribution as we increase the rate of incoming tuples, the number of indexed queries and the network size. The base setting is a network of 10^4 with 10^5 queries and $5 * 10^4$ incoming tuples.

For space considerations, in Figure 9 we show only the effect in the TF load distribution of DAI-Q but similar results are obtained for all algorithms (and for TS load distribution as well). These results can be found in [21]. We observe that by increasing the rate of incoming tuples more load is created in the network but this load is nicely distributed among the existing nodes and in fact even more nodes participate in query processing. The same stands for the case where we increase the number of indexed queries. Finally, a larger network size results in that the current load is distributed to more nodes.

Experiments with a uniform (or a more uniform) data distribution lead to a better load distribution for all algorithms [21]. The value range of attributes has a crucial role since a higher value range can significantly improve all kinds of load distribution especially for DAI-V.

Summary. The experimental evaluation presented above showed the strengths and weaknesses of the four algorithms. SAI outperforms the other algorithms in terms of overlay hops by taking advantage of indexing with respect to the attribute of the relation with the lowest rate of incoming tuples. DAI-T exhibits similar performance when $JFRT$ is in use. With respect to load balancing, one has to choose the algorithm that suits one's scenario, trading network hops for better load distribution. If TF load distribution is more important then DAI-T is the best algorithm while if TS load distribution is needed then DAI-Q is the best. But keep in mind that DAI-Q is a lot more expensive than DAI-T regarding network traffic. SAI offers a solution that compromises between TF and TS load distribution. DAI-V has the advantage that it creates less network traffic but on the other hand it does not utilize a great percentage of the available resource/nodes in the network.

9 Related work

Our work shares common ground with a number of research areas which we survey below.

Distributed and Parallel Databases. The database community has done a lot of work in the area of distributed and parallel databases [24, 14]. The work on hash-based join algorithms for shared-nothing [37] parallel database architectures is most relevant to our work [29], e.g., papers [34, 13] study join queries in multiprocessor environments.

In the distributed database system [25], the notion of the scalable distributed data structure (SDDS) appears which shares a lot with the underline ideas of current structured overlay networks in the sense that a SDDS is maintained even in the presence of node connections, disconnections or failures without centralized coordination.

Continuous Queries and Stream Processing. Database research on continuous queries has its origins in the paper [39] and systems OpenCQ [26] and NiagaraCQ [9]. These papers offer centralized solutions to the problem of continuous query processing. More recently, continuous queries have been studied in depth in the context of monitoring and stream processing with various centralized [28, 8, 33] and distributed proposals [16, 10, 1, 4, 5, 35, 19].

To the best of our knowledge, PeerCQ [16] is the first detailed proposal for processing continuous queries on top of DHTs that has been published before this work. PeerCQ does not concentrate on the relational data model and the SQL query language, and assumes that data are not stored in a DHT but are kept locally at external data sources. The DHT infrastructure is nicely utilized to achieve a good distribution of monitoring and evaluating responsibilities. PeerCQ assumes *heterogeneous* peers and uses a sophisticated model of peer capabilities to distribute continuous queries to evaluator peers while maintaining good load balance and system throughput. It would be interesting to extend our work with the model of peer capabilities proposed by PeerCQ to deal gracefully with peer heterogeneity.

[5] is recent paper that considers distributed equi-join evaluation in wide-area networks consisting of many heterogeneous hosts. [5] concentrates on *network locality* (i.e., proximity of hosts) and *data locality* (i.e., closeness in the data values and frequencies of these data values) with the objective of optimizing the delay of output tuples. Thus the techniques sketched in [5] are complementary to our techniques. In the DHT setting that we consider, it would make sense to investigate the applicability of locality-aware

DHTs such as Tulip [2] to tackle the questions of [5]. This is something we plan to do in the context of project Evergrow where Tulip is also been developed. In a similar manner, [4] shows the benefits of using the locality-aware DHT Tapestry [41] to implement distributed operator placement for continuous query processing of data streams.

[35] is another recent paper that considers distributed query optimization in stream overlay networks and points out differences with distributed query optimization.

Finally, [19] is a recent paper that makes the case for distributed triggers in wide-area monitoring applications. Like [5] and [35], this paper presents many interesting ideas but evaluation of these ideas is left to future work.

Pub/Sub Networks. Recently, a number of researchers have tried to implement content-based pub/sub systems on top of DHTs [40, 38, 31, 3]. The query languages of these systems are based on attribute-operator-value comparisons, thus they are not directly comparable with our work.

P2P Databases. This paper is also closely related with work in the new area of *P2P DBMS* [7, 17, 20]. Currently, one can distinguish two orthogonal research directions in this area: work that emphasizes semantic interoperability of peer databases [7, 17] and work that attempts to push the capabilities of current database query processors to new large-scale Internet-wide applications by utilizing DHTs [20]. Our work belongs to the latter direction and emphasizes the processing of continuous queries on top of DHTs. Previous work has emphasized algorithms for various kinds of queries [15, 18] or the construction and evaluation of real systems [20, 27].

10 Conclusions and future work

We studied the problem of evaluating continuous two-way equi-join queries over DHTs. We evaluated four alternative algorithms with emphasis in distributing the query processing load and minimizing network traffic. The algorithms proposed in this paper are the base for the algorithms we are currently working on for evaluating continuous multi-way join queries. In future work, we also plan to take into account network locality by exploiting locality-aware DHTs. We would also like to consider other types of continuous queries expressed in SQL.

Acknowledgements: We are grateful to Vasilis Samoladas for numerous useful discussions.

References

[1] D. Abadi, Y. Ahmad, M. Balazinska, U. Centintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. CIDR '05.

[2] I. Abraham, A. Badola, D. Bickson, D. Malkhi, S. Malook, and S. Ron. Practical Locality-Awareness for Large Scale Information Sharing. IPTPS '05.

[3] A.Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-Based Publish/Subscribe over P2P Networks. Middleware '04.

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

[5] Y. Ahmad, U. Cetintemel, J. Jannotti, and A. Zgolinski. 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] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data Management for Peer-to-Peer Computing: A Vision. WebDB '02.

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

[10] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable Distributed Stream Processing. CIDR '03.

[11] P.-A. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/Subscribe for RDF-based P2P Networks. ESWC '04.

[12] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundram. Querying Peer-to-Peer Networks using P-Trees. WebDB '04.

[13] D. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms.VLDB85.

[14] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6), 1992.

[15] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems. VLDB '04.

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

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

[18] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate range selection queries in peer-to-peer systems. CIDR '03.

[19] J. M. Hellerstein, A. Jain, S. Ratnasamy, and D. Wetherall. A Wakeup Call for Internet Monitoring Systems: The Case for Distributed Triggers. HotNets '04.

[20] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. VLDB '02.

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

[22] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. VLDB '05.

[23] D. Karger and M. Ruhl. Simple Efficient Load Balancing Algorithms for PeertoPeer Systems. SPAA '04.

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

[25] W. Litwin and M. A. Neimat ad D. A. Schneider. LH*- A Scalable Distributed Data Structure. *ACM Transactions on Database Systems*, 21(4):480–525, 1996.

[26] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *TKDE*, 11(4):610–628, 1999.

[27] B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and I. Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. VLDB '04.

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

[29] M. Mehta and J. DeWitt. Data placement in shared-nothing parallel database systems. *VLDB Journal*, 6:53–72, 1997.

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

[31] P.R. Pietzuch and J.M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. DEBS '02.

[32] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range Queries over DHTs. *Technical Report IRB-TR-03-009, Intel Corp.*, June 2003.

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

[34] D. Schneider and D. DeWitt. Tradeoffs in Processing Multi-Way Join Queries via Hashing in Multiprocessor Database Machines. VLDB '90.

[35] J. Shneidman, P. Pietzuch, M. Welsh, M. Seltzer, and M. Roussopoulos. A Cost-Space Approach to Distributed Query Optimization in Stream Based Overlays. NETDB '05.

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

[37] M. Stonebraker. The case for shared nothing. *Database Engin.*, 9(1), 1986.

[38] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A Peer-to-Peer Approach to Content-Based Publish/Subscribe. DEBS '03.

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

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

[41] B.-Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.