Distributed Processing of Continous Join Queries using DHT Networks

Wenceslao Palma, Reza Akbarinia, Esther Pacitti, Patrick Valduriez

Distributed Processing of Continuous Join Queries using DHT Networks

Wenceslao Palma
INRIA and LINA
University of Nantes, France
wenceslao.palma@univ-nantes.fr

Reza Akbarinia
INRIA and LINA
Nantes, France
rakbarin@cs.uwaterloo.ca

Esther Pacitti
INRIA and LINA
University of Nantes, France
esther.pacitti@univ-nantes.fr

Patrick Valduriez
INRIA and LINA
Nantes, France
patrick.valduriez@inria.fr

ABSTRACT

This paper addresses the problem of computing approximate answers to continuous join queries. We present a new method, called DHTJoin, which combines hash-based placement of tuples in a Distributed Hash Table (DHT) and dissemination of queries exploiting the trees formed by the underlying DHT links. DHTJoin distributes the query workload across multiple DHT nodes and provides a mechanism that avoids indexing tuples that cannot contribute to join results. We provide a performance evaluation which shows that DHTJoin can achieve significant performance gains in terms of network traffic.

1. INTRODUCTION

Recent years have witnessed major research interest in data stream management systems (DSMS), which can manage continuous and unbounded sequences of data items. There are many applications that generate data streams including financial applications [7], network monitoring [24], telecommunication data management [6], sensor networks [4], etc. Processing a query over a data stream involves running the query continuously over the data stream and generating a new answer each time a new data item arrives. However, the unbounded nature of data streams makes it impossible to store the data entirely in bounded memory. This makes difficult the processing of queries that need to compare each new arriving data with past ones. For example, real data traces of IP packets from an AT&T data source [11] show an average data rate of approximately 400 Mbits/sec, which makes it hard to keep pace for a DSMS. Moreover, a DSMS may have to process hundreds of user queries over multiple data sources. For most distributed streaming applications, the naive solution of collecting all data at a single site is simply not viable [8]. Therefore, we are interested in techniques for processing continuous queries over collections of distributed data streams. This setting imposes high processing and memory requirements. However, approximate answers are often sufficient when the goal of a query is to understand trends and making decisions about measurements or utilizations patterns. An example of such queries is join queries which are very important for many applications. For example, consider a network monitoring application that needs to issue a join query over traffic traces from various links, in order to monitor the total traffic that passes through three routers (R_1, R_2 and R_3) and has the same destination host within the last 10 minutes. Data collected from the routers generate streams S_1,S_2 and S_3. The content of each stream tuple contains a packet destination, the packet size and possibly other information. This query can be posed using a declarative language such as CQL [2], a relational query language for data streams, as follows:

q_1: Select sum (S_1.size)
From S_1[range 10 min], S_2[range 10 min], S_3[range 10 min]
Where S_1.dest=S_2.dest and S_2.dest=S_3.dest

A common solution to the problem of processing join queries over data streams is to execute the query over a sliding window [12] that maintains a restricted number of recent data items. To improve performance and scalability, distributed processing of data streams is a well accepted approach [25][8]. However, even in a distributed setting high stream arrival rates and cost-intensive query operations may cause a DSMS to run out of resources. For example, when the memory allocated to maintain the state of a query is not sufficient to keep the window size entirely, the completeness, i.e., the fraction of results produced by an query operator over the total results (which could be produced under perfect conditions) is reduced.

A major problem in the distributed processing of data streams is the occurrence of node failures. For example, consider the join query q_1 of the above example. A possible query plan that consists of two join operations is ($S_1 \bowtie S_2$) $\bowtie$ S_3 where each join operation can be allocated to two different nodes and streams tuples come from different nodes. If during the processing of the continuous query q_1 the node that processes S_3 fails, the node that processes $S_1 \bowtie S_2$ can generate partial results irrespective of whether

they produce join query results. Thus, if no matching tuple of S_3 appears in the node that processes $(S_1 \bowtie S_2) \bowtie S_3$, the resources involved in sending, processing and storing $S_1 \bowtie S_2$ tuples are wasted and the results may be incomplete. Furthermore, this produces unnecessary intermediate join results and reduces completeness of results due to the lack of matching tuples caused by the failure of a node.

In this paper, we address the problem of computing approximate answers to windowed stream joins over data streams. We propose a method, called DHTJoin, which deals with efficient processing of join queries over all data items which are stored in a DHT network. We provide an efficient processing of join queries, in terms of network traffic, by avoiding the indexing of tuples that are not involved by the target continuous query and by addressing node failures that can produce unnecessary intermediate join results. We evaluated the performance of DHTJoin through simulation. The results show the effectiveness of our solution compared with previous work.

This paper is an extended version of [19] with the following added value. First, we present a dissemination system (Section 3.2) based on the trees formed by DHT links that uses $O(n-1)$ messages. This yields an important reduction of network traffic compared with the $O(\frac{nlogn}{2})$ messages generated by the dissemination system proposed in our previous work. We also propose (Section 3.4) and validate (Section 5) an optimization that avoids the sending of unnecessary intermediate results, i.e. tuples that cannot contribute to join results, when a node fails. For example, considering the query q_1 of the above example and the query plan $(S_1 \bowtie S_2) \bowtie S_3$, when the node that processes $(S_1 \bowtie S_2) \bowtie S_3$ finds out that cannot generate join results due to the lack of S_3 matching tuples, it sends a feedback message to node that processes $S_1 \bowtie S_2$ which immediately suspends the sending of unnecessary intermediate results. Finally, we show analytically in Section 4 that DHTJoin can scale up the processing of continuous join queries using multiple peers and improves the completeness of join results linearly as memory capacity is increased.

The rest of this paper is organized as follows. In Section 2, we introduce our system model and define the problem. In Section 3, we describe DHTJoin. In Section 4, we provide an analysis of result completeness of our algorithms which relates memory constraints, stream arrival rates and result completeness. In Section 5, we provide a performance evaluation of our solution through simulation using Java. In Section 6, we discuss related work. Finally, Section 7 concludes.

2. SYSTEM MODEL AND PROBLEM DEFINITION

In this section, we introduce a general system model for processing data streams over DHTs, with a DHT model and a stream processing model. Then, we state the problem.

2.1 DHT Model

In our system, the nodes of the overlay network are organized using a DHT protocol. While there are significant implementation differences between DHTs [20] [23], they all map a given key k onto a node p using a hash function and can lookup p efficiently, usually in $O(logn)$ routing hops where n is the number of nodes. DHTs typically provide two basic operations : *put(k, data)* stores a key k and its associated *data* in the DHT using some hash function; *get(k)* retrieves the data associated with k in the DHT. Tuples are originated at certains nodes and continuous queries are originated at any node of the network. Nodes insert data in the form of relational tuples and queries are represented in SQL. Tuples and queries are timestamped to represent the time that they are inserted in the network by some node. We assume that data and query sources are equipped with well-synchronized clocks by using the public domain Network Time Protocol (NTP) designed to work over packet-switched and variable latency data networks and already tested in a distributed DSMS as Borealis [25]. Each tuple has a unique key generated using the name of the node that inserts it, the name of the relation to which it belongs and its timestamp. Additionally, each query is associated with a unique key q_{id} used to identify it in query processing, optimization tasks and to relate it to the node that submitted it. We identify two types of queries depending on the attributes involved. If the join attribute is the same in all the relations of the query (e.g. query q_1 of section 1) we say that it is a query of type 1, otherwise we say that the query is of type 2, i.e. the join attributes are different.

2.2 Stream Processing Model

A data stream S_i is a sequence of tuples ordered by an increasing timestamp where $i \in [1..m]$ and $m \geq 2$ denotes the number of input streams. At each time unit, a number of tuples of average size l_i arrives to stream S_i. We use λ_i to denote the average arrival rate of a stream S_i in terms of tuples per second. Many applications are interested in making decisions over recently observed tuples of the streams. This is why we maintain each tuple only for a limited time. This leads to a sliding window $S[W_i]$ over S_i that is defined as follows. Let W_i denotes the size of $S[W_i]$ in terms of seconds, i.e. the maximum time that a tuple is maintained in $S[W_i]$. Let $TS(s)$ be a function that denotes the arrival time of a tuple s and t be current time. Then $S[W_i]$ is defined as $S[W_i] = \{s | s \in S_i \wedge (t - TS(s) \leq W_i\}$. Tuples continuously arrive at each instant and expire after W_i time steps (time units). Thus, the tuples under consideration change over time as new tuples get added and old tuples get deleted. In practice, when arrival rates are high and the memory dedicated to the sliding window is limited, it becomes full rapidly and many tuples must be dropped before they naturally expire. In this case, we need to decide whether to admit or discard the arriving tuples and if admitted, which of the existing tuples to discard. This kind of decision is made using a load shedding strategy [22][26] which yields that only a fraction of the complete result will be produced.

2.3 Problem Definition

In this paper, we address the problem of processing join queries over data streams. We view a data stream as a sequence of tuples ordered by monotonically increasing timestamps. The nodes are assumed to synchronize their clocks using the public domain Network Time Protocol (NTP), thus achieving accuracies within milliseconds [3]. Each tuple and query have a timestamp that may be either implicit, i.e. generated by the system at arrival time, or explicit, i.e. inserted by the source at creation time. Formally, the problem can be defined as follows. Let $S = \{S_1, S_2, ..., S_m\}$ be

a set of data streams, and $Q = \{Q_1, Q_2, ..., Q_n\}$ be a set of join queries specified on these data streams. Our goal is to provide an efficient method to execute Q over S in a way that minimizes network traffic.

3. DHTJOIN METHOD

In this section, we describe our solution, DHTJoin, for processing continuous join query processing using DHTs. The main issues for processing continuous queries in DHTs are the following: how to route data and queries to nodes in an efficient way; how to provide a data storage mechanism for storing relational data; and how to provide a good approximate answer to join queries.

DHTJoin has two steps: indexing of tuples and dissemination of queries. A tuple inserted by a node is indexed, i.e., stored at another node using DHT primitives and a query is disseminated using the embedded trees inherents to DHTs networks. However, a node indexes a tuple only if there is a query that contains an attribute of the arriving tuple in its where clause. To this end, a node stores locally a disseminated query and once it receives a tuple it checks for already disseminated queries that contain an attribute of the arriving tuple in its where clause.

We describe the design of DHTJoin based on Chord which is a simple and very popular DHT. However, the techniques used here can be adaptable to others DHTs such as Pastry [21] and Tapestry [28]. To process a query, we consider different kinds of nodes. The first kind is Stream Reception Peers (SRP) for indexing tuples to the second kind of nodes, the Stream Query Peers (SQP). In Figure 1(b), nodes 3, 6 and 7 correspond to SRP because they receive tuples belonging to streams z, y, and x respectively. SQP are responsible for executing query predicates over the arriving tuples using their local sliding windows, and sending the results to the third kind of node(s), the User Query Peers (UQP). In Figure 1(b), nodes 1 and 4 are SQP because node 1 computes the join predicate $X.A = Y.A$ of query q_2 (submitted at node 0) and node 4 performs the join predicate $Y.B = Z.B$ of q_2. In addition, node 0 is a UQP because query q_2 was submitted at this node.

To support dissemination of queries, a node must be a dissemination node (i.e. executes a dissemination protocol) while to index tuples, a node must be a DHT peer. Note that the difference between SRP, SQP and UQP is functional and the same node can support all these functionalities.

3.1 Indexing tuples

Let us describe our DHTJoin method for streams $S = \{S_1, S_2, ..., S_m\}$. Let s_i be a tuple belonging to S_i. Let A be the set of attributes in s_i and $val(s_i, \alpha)$ be a function that returns the value of an attribute $\alpha \in A$ in tuple s_i. Let h be a uniform hash function that hashes S_i's name and $val(s_i, \alpha)$ into a DHT key, i.e. a number which can be mapped to a node *id*. Let $S[W_i]$ denote a sliding window on stream S_i. Recall that we use time-based sliding windows where W_i is the size of the window in time units. At time t, a tuple belongs to $S[W_i]$ if it has arrived in the time interval $[t - W_i, t]$.

For indexing a tuple s_i that arrives at a SRP, each tuple obtains an index key computed as $key = h(S_i, val(s_i, \alpha))$. The attribute α in s_i is chosen by searching locally for queries that contains α in its where clause. Then s_i is indexed to a SQP, by performing $put(key, s_i)$. Thus, tuples of different streams having the same *key* are put in the same SQP node and are stored in sliding windows where they are processed to produce the result of a specific join predicate.

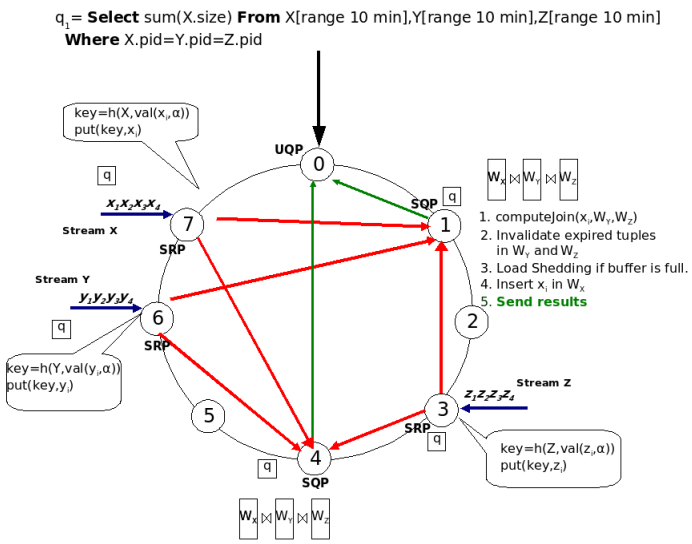
3.2 Disseminating queries

Query dissemination is important to distribute query load so that many DHT nodes may participate in query processing tasks or optimization decisions. A query can originate at any of the nodes and is disseminated using a tree [5]. To disseminate a query, DHTJoin dynamically builds a dissemination tree as proposed in [9]. The root of the tree is the node that submits the query (an UQP node). The query is disseminated from the root node to all nodes of the DHT using a divide-and-conquer approach. A dissemination message contains q_{id} and a range of dissemination. For example, using a fully-populated Chord ring with 8 nodes, each one contains a routing table of $log(n)$ entries called fingers. The i^{th} entry in the table at node n contains the identity of the first node that succeeds or equal $n + 2^i$. A dissemination message initiated at node 0 is sent to finger nodes 1, 2 and 4 (see Figure 2) giving them the disseminations limits [1,2), [2,4) and [4,0) respectively. The disseminations limits are used to restrict the forwarding space of a node and they are constructed using as a upper bound the finger $i + 1$. Each node applies the same principle reducing the search scope. When node 2 receives the dissemination message with limits [2,4) it examines the routing table and sends the message to node 3. Once node 4 receives the dissemination message it examines the routing table and sends the message to nodes 5 and 6 with limits [5 ,6) and [6,0) respectively. In the same way, node 5 does not continues with the dissemination process (since there are no nodes between [5,6)) and node 6 disseminates the message to node 7. This forwarding process assures a network coverage of 100%, generates $n - 1$ messages and a tree of depth $log(n)$, which fixes the latency of query dissemination.

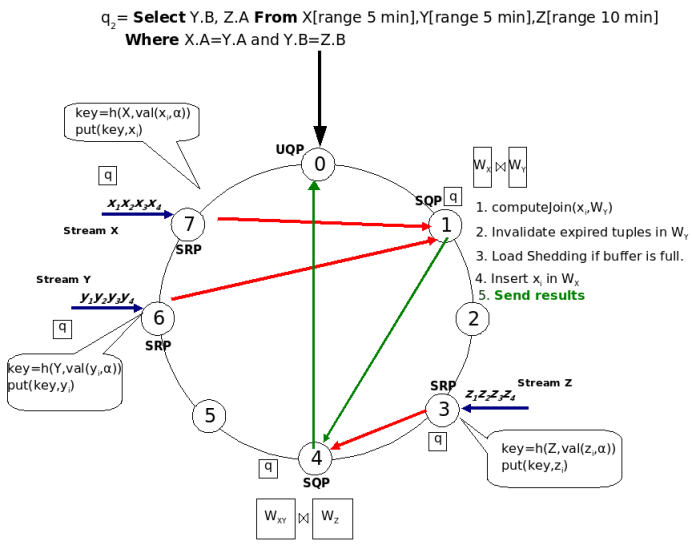
3.2.1 Adaptability to other DHTs

The design of DHTJoin is based on Chord which is a simple and very popular DHT. However, the dissemination technique used can be adapted to others DHTs such as Pastry [21] and Tapestry [28]. Recall that the basic idea in the dissemination of queries (using Chord) is to consider a *lookup* operation as a binary search in spite of its ring geometry. In the case of Pastry and Tapestry are both extended versions of PRR trees that support dynamic node membership.

We here take Pastry as an example and demonstrate how to apply the dissemination of queries into Pastry using the mechanism proposed in [5]. Assuming a network of size n each Pastry node maintains a routing table of $log_{2^b} n$ rows with 2^b entries each. For the purposes of routing the identifier of a node and keys can be thought of as a sequence of L digits in base 2^b. The mechanism to route a message is prefix-based, i.e. the routing is achieved by forwarding the message to a node that shares a common prefix by at least one more digit. Pastry can route a message to any node in $log_{2^b} n$ hops. For easy of explanation we use $b = 1$, $L = 3$ and a network of 8 nodes. A dissemination message initiated a node 000 contains the query id q_{id} and the message is sent to the 3 nodes of its routing table 100, 010 and 001 adding the routing table row r of each node. When a node receives a dissemination message it searches in the routing table all the nodes located in rows greater than r (if any) and dissem-



(a) A join example of query type 1



(b) A join example of query type 2

Figure 1: Query Processing in DHTJoin

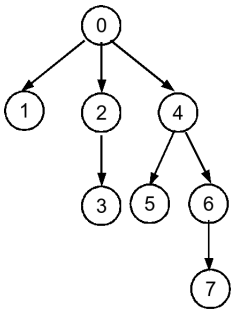


Figure 2: A dissemination tree formed using DHT links of a 8-node Chord ring

inates them the message. This process is repeated at each node receiving the message generating a dissemination tree of depth $log(n)$.

3.3 Example of DHTJoin query processing

Let us illustrate how DHTJoin performs query processing using the following query of type 2:

q_2: Select Y.B, Z.A
From X[range 5 min], Y[range 5 min], Z[range 5 min]
Where X.A=Y.A and Y.B=Z.B

This query specifies an equijoin among X, Y and Z streams over the last 5 minutes. The query q_2 is submitted at node 0 and disseminated over the entire network as soon as it is submitted. Thus, after a while, all nodes will know the existence of this query and be able to index the incoming streams (tuples). Once an X-, Y- or Z-tuple arrives at nodes 7, 6 and 3 respectively, each node checks locally whether the query q_2 contains in its where clause an attribute α of the arriving tuple (see Figure 1(b)). If yes, nodes 7, 6 and 3 execute the task of a SRP. For instance, in our example, node 7 indexes x_i by performing two operations: $key = h(X, val(x_i, \alpha))$ and straight afterwards $put(key, x_i)$. A join predicate with respect to q_2 is evaluated at a SQP (node 1) only with tuples that arrive in the system after the query.

DHTJoin manages the sliding windows at each SQP node as follows. Upon each indexed tuple arrival at a SQP node, tuples that have expired are invalidated from the sliding window of a SQP. For example, at node 1 in Figure 1(b), tuples expired in $S[W_Y]$ are invalidated on the arrival of X-tuples. The load shedding procedure is executed over $S[W_X]$'s buffer if there is not enough memory space to insert the arriving tuple. In our example, the intermediate results produced by SQP node 1 are sent to SQP 4 using the value of B attribute belonging to the Y-tuple using $key = h(Y, val(y_i, B))$ and straight afterwards $put(key, \{x_i, y_j\})$. The join result tuples produced by SQP node 4 are immediately sent to the appropriate UQP node (whose address is provided when starting query dissemination).

For queries of type 1 (see Figure 1(a)), we apply the same method. However, as the join attribute is the same in all the relations of query q_1, all the tuples having the same attribute value and the join operators are located in the same node without producing intermediate results. In our example nodes 7,6 and 3 indexes x_i, y_i and z_i tuples in SQP nodes 1 and 4 depending of the join attribute value. The results produced by SQP nodes 1 and 4 are sent to the UQP node.

3.4 Optimization

DHTJoin distributes the query workload across multiple DHT nodes and provides a mechanism that avoids indexing tuples using attributes not contained in the where clause of a query. However, when a source of data streams interrupts its operation, an SQP node can generate partial results irrespective of whether they produce join query results. In this section, we address the problem of indexing tuples that never contribute to generate join results.

Let us consider a query of type 2 where there are nodes connected by a producer-consumer relationship, whereby a pro-

Table 1: Variables and functions used in our analysis

n	number of nodes
m	number of streams in query Q
$S = \{S_1, S_2, ..., S_m\}$	set of streams
λ_i	arrival rate of stream i in tuples/sec
W_i	window size of stream i in seconds
sel	join selectivity $\in$ [0..1]
$m(S_i)$	functions that returns the memory assigned to S_i tuples

ducer node generates tuples to be processed by a consumer node [27]. For example, in query q_2 (see Figure 1(b)), node 1 is the producer of join tuples between X and Y ($X \bowtie Y$) that node 4 joins with Z-tuples. If the Z source (see node 3 in Figure 1(b)) leaves the DHT, the indexing of Z-tuples is stopped, thus yielding no join results because of no matching tuples in node 4. Furthermore, if no matching tuples of Z appears in node 4 before expiration of $X \bowtie Y$ tuples, the resources involved in sending, processing and storing these tuples are wasted.

Motivated by the above situation, we propose the following optimization. Once a node (e.g. node 4) detects that it is not possible to generate join results, due to the absence of matching tuples of Z, it sends a message to node 1 alerting that is not necessary send tuples. This tuple stays stored at node 1 as long as it fits in the window size. Once the communication with the source of Z-tuples is reestablished, node 4 sends a message to node 1 alerting that the tuples can be sent. In this way, during the period that node 4 does not receive Z-tuples, some tuples stored in node 1 are deleted due to window constraints, thus avoiding to send tuples that never contribute to generate join results.

By eliminating unnecessary intermediate results, this optimization yields an important reduction of network traffic and a better utilization of local resources. In sensor networks where the nodes are generally powered by batteries and power preservation is a major objective, this optimization is particularly useful as reducing network traffic yields reduced power consumption.

4. ANALYSIS OF RESULT COMPLETENESS

The notion of result completeness is important in distributed and P2P databases since partial (incomplete) query answers are often only possible[18][16]. Result completeness is thus defined as the fraction of results actually produced over the total results (which could be produced under perfect conditions). In data streaming applications,the potential high arrival rates of streams impose high processsing and memory requirements. However, approximate answers are often sufficient when the goal of a query is to understand trends and making decisions about measurement or utilization patterns. Query approximation can be done by limiting the size of states maintained for queries [15]. In our analysis we focus in the case where the memory allocated to maintain the state of a query is not sufficient to keep the window size entirely, thus reducing the received join results and completeness. DHTJoin provides more memory to store tuples, but we consider that determining the number of computing resources necessaries to achieve a certain degree of completeness for a given query is an important aspect in the setup phase of DHTJoin.

In this section, we propose formulas which relate peer memory constraints, stream arrival rates, and result completeness. We will use these formulas in our performance evaluation and they could be useful to a DHTJoin user (e.g. an application developer) to define and tune a DHT network for specific application requirements. We provide the necessary equations to calculate the completeness in a 2-way join and afterwards we generalize our results for a m-way join.

For ease of analysis, we make simplifying assumptions: the tuples are uniformly distributed across the DHT network; the memory assigned to store tuples is the same at each peer; we use the average rate to characterize the rate of arrivals of incoming tuples and stream tuples arrive in monotonically increasing order of their timestamps. We use the notations specified in Table 1. In order to illustrate our analysis, let us consider the following join query over two streams S_1 and S_2:

Q: Select *
from S_1[range 5 min], S_2[range 5 min]
where $S_1.x = S_2.x$

The expected tuple arrival rate of streams S_1 and S_2 at each node of the DHT is $\frac{\lambda_1}{n}$ and $\frac{\lambda_2}{n}$ respectively. Thus, the expected number of join tuples generated by $S_1 \bowtie S_2$ per unit time at each node can be estimated as

$$T(S_1, S_2) = sel \times (\frac{W_1\lambda_1}{n}) \times (\frac{W_2\lambda_2}{n}) \quad (1)$$

Each node needs a memory space for storing tuples in its local sliding window equivalent to $\frac{W_1\lambda_1}{n}$ and $\frac{W_2\lambda_2}{n}$. In general, if ($\frac{W_i\lambda_i}{n} > m(S_i)$) we have a loss rate (Lr) to store tuples equivalent to:

$$Lr(S_i) = \begin{cases} 0, & \frac{W_i\lambda_i}{n} \leq m(S_i) \\ \frac{W_i\lambda_i}{n} - m(S_i), & otherwise \end{cases} \quad (2)$$

Assuming that memory is insufficient to retain all the tuples in W_1 and W_2, the loss of join tuples L of S_1 and S_2 is:

$$L(S_1) = sel \times Lr(S_1) \times (\frac{W_2\lambda_2}{n}) \quad (3)$$

$$L(S_2) = sel \times Lr(S_2) \times (\frac{W_1\lambda_1}{n}) \quad (4)$$

Let α_i be the S_i-tuples stored in the memory space $m(S_i)$ and β_i be the S_i-tuples not stored due to memory constraints (see Figure 3). We can rewrite equations (3) and (4) as:

$$L(S_1) = sel \times \beta_1 \times (\alpha_2 + \beta_2) = (sel \times \alpha_2 \times \beta_1) + (sel \times \beta_1 \times \beta_2)$$

$$L(S_2) = sel \times \beta_2 \times (\alpha_1 + \beta_1) = (sel \times \alpha_1 \times \beta_2) + (sel \times \beta_1 \times \beta_2)$$

Notice that the tuples related to expression ($sel \times \beta_1 \times \beta_2$) are counted in both $L(S_1)$ and $L(S_2)$. This expression can be rewritten as: ($sel \times Lr(S_1) \times Lr(S_2)$). The total loss of join tuples TL of $S_1 \bowtie S_2$ is the sum of the loss of join tuples $L(S_1)$ and $L(S_2)$ minus the tuples counted twice:

$$TL(S_1, S_2) = L(S_1) + L(S_2) - (sel \times Lr(S_1) \times Lr(S_2)) \quad (5)$$

The completeness C of a $S_1 \bowtie S_2$ join query is the fraction of total results $T(S_1, S_2)$ minus the loss of tuples $TL(S_1, S_2)$ and total results $T(S_1, S_2)$, using equation (1) and equation (5) C is:

$$C = \frac{T(S_1, S_2) - TL(S_1, S_2)}{T(S_1, S_2)} \quad (6)$$

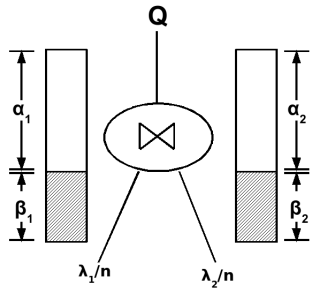


Figure 3: A join state including stored and non stored tuples

Developing expressions in (6) allows us to simplify C to:

$$C = \frac{n^2 \times m(S_1) \times m(S_2)}{W_1\lambda_1 \times W_2\lambda_2} \tag{7}$$

Moreover, we can write (7) as:

$$n = \sqrt{\frac{C \times (W_1\lambda_1) \times (W_2\lambda_2)}{m(S_1) \times m(S_2)}} \tag{8}$$

This equation allows us to evaluate how many peers are necessary to evaluate a 2-way join query.
Now we generalize our analysis to m-way joins as following. Recall that the total loss of join tuples TL is the sum of the loss of join tuples minus the tuples counted more than one time. The sum of the loss of join tuples can be easily extended to an m-way join as $\sum_{i=1}^{m} L(S_i)$. However, the expression that represents the tuples counted more than one time is more difficult to generalize. We use the same method of rewriting (3) and (4) to find the expression that represents the case of tuples counted more than one time. Thus in a $S_1 \bowtie S_2 \bowtie S_3$ join we rewrite $L(S_1)$,$L(S_2)$ and $L(S_3)$, discovering that $(sel^2 \times \beta_1 \times \beta_2 \times \alpha_3)$, $(sel^2 \times \beta_1 \times \beta_3 \times \alpha_2)$ and $(sel^2 \times \beta_2 \times \beta_3 \times \alpha_1)$ are counted twice and $(sel^2 \times \beta_1 \times \beta_2 \times \beta_3)$ is counted triple. Rewriting α_i and β_i we arrive at the following expression:

$$sel^2 Lr(S_1)Lr(S_2)m(S_3) + sel^2 Lr(S_1)Lr(S_3)m(S_2) + sel^2 Lr(S_2)Lr(S_3)m(S_1) + 2sel^2 Lr(S_1)Lr(S_2)Lr(S_3).$$

Repeating the same method with m-way joins ($m \geq 4$) and analyzing the resulting expressions, we arrive at the following general expression for a $S_1 \bowtie S_2 \bowtie .... \bowtie S_m$ join:

$$\sum_{k=2}^{m} \sum_{\substack{S' \subseteq S \\ |S'|=k}} \sum_{\substack{S'' \subseteq S \\ |S''|=m-k \\ S'' \cap S' = \emptyset}} (sel^{m-1}(k-1) \prod_{a \in S'} Lr(a) \prod_{b \in S''} m(b))$$

Now, the general case of (5) can be expressed as:

$$TL(S_1, S_2, ..., S_m) = \sum_{i=1}^{m} L(S_i) - \sum_{k=2}^{m} \sum_{\substack{S' \subseteq S \\ |S'|=k}} \sum_{\substack{S'' \subseteq S \\ |S''|=m-k \\ S'' \cap S' = \emptyset}} (sel^{m-1}(k-1) \prod_{a \in S'} Lr(a) \prod_{b \in S''} m(b)) \tag{9}$$

The completeness C of a $S_1 \bowtie S_2 \bowtie .... \bowtie S_m$ join query, using the general form of (1) and equation (9) is:

$$C = \frac{T(S_1, S_2, ..., S_m) - TL(S_1, S_2, ..., S_m)}{T(S_1, S_2, ..., S_m)} \tag{10}$$

Developing expressions in (10) allows us to simplify C to:

$$C = \frac{n^m \prod_{i=1}^{m} m(S_i)}{\prod_{i=1}^{m} W_i\lambda_i} \tag{11}$$

and to obtain

$$n = \sqrt[m]{\frac{C \times \prod_{i=1}^{m} W_i\lambda_i}{\prod_{i=1}^{m} m(S_i)}} \tag{12}$$

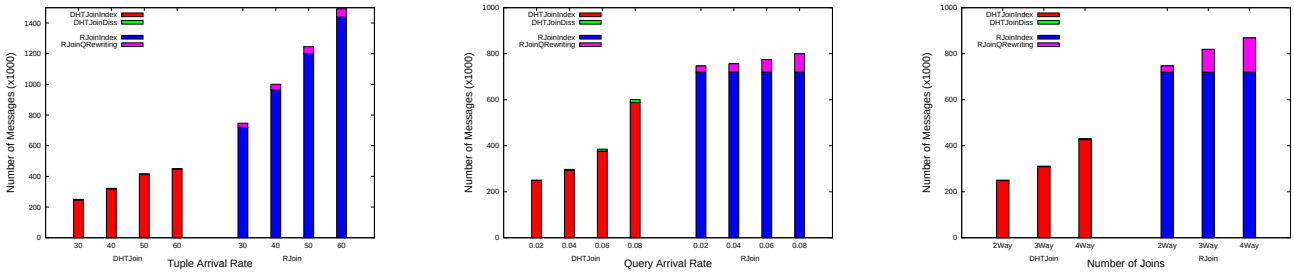
It is clear from our analysis that (11) is independent of selectivity which is reasonable in the context of continuous join queries.
As our analysis shows, DHTJoin can scale up the processing of continuous join queries using multiple peers and improve the completeness of join results. Using (12) a DHTJoin user can adjust the size of the network by evaluating how many peers are necessary to process a continuous join query for given stream arrival rates and a desired result completeness.

5. PERFORMANCE EVALUATION

To test our DHTJoin method, we built a Java-based simulator. Our simulator is based on Chord which is a simple and efficient DHT. We generate arbitrary input data streams consisting of synthetic asynchronous data items with no tuple-level semantics. We have a schema of 4 relations, each one with 10 attributes. In order to create a new tuple we choose a relation and assign values to all its attributes using a Zipf distribution with a default parameter of 0.9. The max value of the domain of the join attribute is fixed to 1000. Tuples on streams are generated at a constant rate of 30 tuples per second. Queries are generated with a mean arrival rate of 0.02, i.e., a query arrives to the system every 50 seconds on average. The network size is set to 1024 nodes. In all experiments, we use time-based sliding windows of 50 seconds and the method of dissemination of queries proposed in Section 3. For each of our tests, we run the simulator for 300 seconds. In order to assess our approach, we compare the network traffic of DHTJoin against a complete implementation of RJoin [14] which is the most relevant related work (see Section 6). RJoin uses incremental evaluation based on tuple indexing and query rewriting over distributed hash tables. The network traffic is the total number of messages needed to index tuples and disseminate a query in DHTJoin or to index tuples and perform query rewriting in RJoin. In the rest of this section, we evaluate network traffic and the effectiveness of optimization proposed in Section 3.

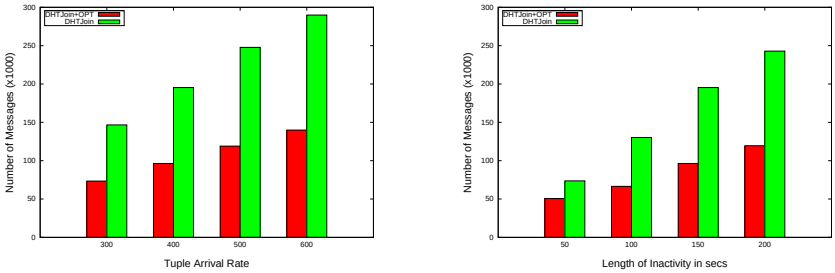
5.1 Network traffic

In this section, we investigate the effect of tuples' arrival rate, query's arrival rate and number of joins on the network traffic. The network traffic of RJoin and DHTJoin grows as the tuples' arrival rate grows. In RJoin, as more tuples arrive, the number of messages related to the indexing of tuples and query rewriting increases (see Figure 4(a)). As expected, DHTJoin generates significantly less messages because a high tuples' arrival rate does not mean more indexing of tuples. The reason is that before indexing a tuple, DHTJoin checks for the existence of a query that requires



(a) Effect of tuple arrival rate (b) Effect of query arrival rate (c) Effect of number of joins

Figure 4: Effect of tuple/query arrival rates and number of joins on network traffic



(a) Effect of tuple arrival rate (b) Effect of duration of inactivity

Figure 5: Reduction of intermediate results and its impact on network traffic

it. In Figure 4(b), we show that, as more queries arrive, RJoin generates the same number of messages when indexing tuples. However, more query rewriting messages are generated when more queries arrive. DHTJoin generates more messages because new queries related to new different values used in the tuples arrive in the system. The number of messages generated by the query dissemination increases but this increase is lower to that of the query rewriting messages generated by RJoin. Figure 4(c) shows that more join require more network traffic. RJoin generates more query rewriting when there are more joins in the queries. However, in DHTJoin the network traffic increases only if the arriving queries require of attributes not present in the already disseminated queries. The reason is that with the dissemination of queries, DHTJoin can avoid the unnecessary indexing of tuples that are not required by the queries.
In summary, due to the integration of query dissemination and hash-based placement of tuples our approach avoids the excessive traffic generated by RJoin which is due to its method of indexing tuples.

5.2 Dealing with stream source failures

In this section, we investigate the effect of optimization proposed in Section 3 in order to reduce the traffic the tuples that never contribute to generate join results due to stream source failures. In this experiment we replicate the same scenario of Figure 1(b) with $\lambda_i = 400$tuples/sec. In Figure 5(a), we show that, as more tuples arrive and considering that the source of Z-tuples leaves the DHT, the proposed optimization reduces the network traffic considerably. In Figure 5(b), we show that, as more long is the period of inactivity (time between leave and join) of a stream source, the generation of tuples that never contribute to generate join results increases. However, by eliminating unnecessary intermediate results, this optimization yields an important reduction of network traffic.

6. RELATED WORK

A DHT can serve as the hash table that underlies many parallel hash-based join algorithms. However, our approach provides Internet-wide scalability. Our work is related to many studies in the field of centralized and distributed continuous query processing [13][10][26][6][17]. In PIER [13], a query processor is used on top of a DHT to process one-time join queries. Recent work on PIER has been developed to process only continuous aggregation queries. PeerCQ [10] was developed to process continuous queries on top of a DHT, However, PeerCQ does not consider SQL queries and the data is not stored in the DHT. Borealis [26], TelegraphCQ [6] and DCAPE [17] have been developed to process continuous queries in a cluster setting and many of their techniques for load-shedding and load balancing are orthogonal to our work. The most relevant previous work regarding the utilization of a DHT network is [14] which proposes RJoin. In RJoin, a new tuple is indexed twice for each attribute it has; wrt the attribute name and wrt the attribute value. A query is indexed waiting for matching tuples. Each arriving tuple that is a match causes the query to be rewritten and reindexed at a different node. This incremental evaluation is based on tuple indexing and query rewriting

over distributed hash tables. A major difference in our work differs is that DHTJoin avoids indexing tuples that cannot contribute to generate join results.
A solution to estimate the completeness has been proposed in [16]. Completeness is calculated on a peer level using the notion of routing graphs. The routing graphs trace the routes that a one time query and its sub-queries take through the network. Our work instead considers continuous queries and completeness is calculated on a data level.

7. CONCLUSION

In this paper, we proposed a new method, called DHTJoin, for processing continuous join queries using DHTs. DHTJoin combines hash-based placement of tuples and dissemination of queries using the trees formed by the underlying DHT links. DHTJoin takes advantage of the indexing power of DHT protocols and dissemination of queries to avoid the indexing of tuples that cannot contribute to generate join results. We show analytically that DHTJoin can scale up the processing of continuous join queries using multiple peers and improves the completeness of join results significantly. To validate our contribution, we implemented DHTJoin as well as RJoin which is the most relevant state of the art solution in the context of processing continuous join queries. Our performance evaluation shows that DHTJoin yields significant performance gains due to the mechanism of indexing tuples and the elimination of unnecessary intermediate results. Our results also demonstrate that the total number of messages of DHTJoin is always less than that of RJoin wrt tuple arrival rate, query arrival rate and number of joins.
As future work, we plan to address the problem of efficient execution of top-k join queries over data streams using DHTs, taking advantage of the best position algorithms [1] which can be used in many distributed and P2P systems for efficient processing of top-k queries.

8. REFERENCES

[1] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.

[2] A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3):6–12, 2004.

[3] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD Conference*, pages 515–526, 2004.

[4] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.

[5] M. Castro et al. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *INFOCOM*, 2003.

[6] S. Chandrasekaran et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD Conference*, pages 379–390, 2000.

[8] G. Cormode and M. N. Garofalakis. Streaming in a connected world: querying and tracking distributed data streams. In *SIGMOD Conference*, pages 1178–1181, 2007.

[9] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi. Efficient broadcast in structured p2p networks. In *IPTPS*, pages 304–314, 2003.

[10] B. Gedik and L. Liu. Peercq: A decentralized and self-configuring peer-to-peer information monitoring system. In *ICDCS*, pages 490–499, 2003.

[11] L. Golab et al. Optimizing away joins on data streams. In *SSPS*, pages 48–57, 2008.

[12] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[13] R. Huebsch et al. Querying the internet with pier. In *VLDB*, pages 321–332, 2003.

[14] S. Idreos, E. Liarou, and M. Koubarakis. Continuous multi-way joins over distributed hash tables. In *EDBT*, pages 594–605, 2008.

[15] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[16] M. Karnstedt et al. Estimating the number of answers with guarantees for structured queries in p2p databases. In *CIKM*, pages 1407–1408, 2008.

[17] B. Liu et al. A dynamically adaptive distributed system for processing complex continuous queries. In *VLDB*, pages 1338–1341, 2005.

[18] F. Naumann, J. C. Freytag, and U. Leser. Completeness of integrated information sources. *Inf. Syst.*, 29(7):583–615, 2004.

[19] W. Palma, R. Akbarinia, E. Pacitti, and P. Valduriez. Efficient processing of continuous join queries using distributed hash tables. In *Euro-Par*, pages 632–641, 2008.

[20] S. Ratnasamy et al. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[21] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[22] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[23] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[24] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, page 594, 1996.

[25] N. Tatbul, U. Çetintemel, and S. B. Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB*, pages 159–170, 2007.

[26] N. Tatbul and S. B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.

[27] Y. Yang and D. Papadias. Just-in-time processing of continuous queries. In *ICDE*, pages 1150–1159, 2008.

[28] B. Y. Zhao et al. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.