Adaptive Optimization of Join Trees for Multi-join Queries over Sensor Streams

Joseph Gomes

Hyeong-Ah Choi

Department of Computer Science

The George Washington University

Washington, DC, USA

Email: {joegomes,hchoi}@gwu.edu

Abstract—Data processing applications for sensor streams have to deal with multiple continuous data streams with inputs arriving at highly variable and unpredictable rates from various sources. These applications perform various operations (e.g. filter, aggregate, join etc) on incoming data streams in real-time according to predefined queries or rules. Since the data rate and data distribution fluctuate over time, an appropriate join tree for processing join queries must be adaptively maintained in response to dynamic changes to prevent rapid degradation of the system performance. In this paper, we address the problem of finding an optimal join tree that maximizes throughput for sliding window based multi-join queries over continuous data streams and prove its NP-Hardness. We present a dynamic programming algorithm, OptDP, which produces the optimal tree but runs in an exponential time in the number of input streams. We then present a polynomial time greedy algorithm, XGreedyJoin. We tested these algorithms in ARES, an adaptively re-optimizing engine for stream queries, which we developed by extending Jess¹. For almost all instances, trees from XGreedyJoin perform

close to the optimal trees from OptDP, and significantly better than common heuristics-based XJoin algorithms.

Index Terms—stream query, join query, sensor stream, sliding windows, adaptive optimization, dynamic programming

I. INTRODUCTION

Due to advancements in sensor technology and their miniature size, sensors are being used everywhere to monitor and collect data. Sensor networks are being used in a wide range of applications in fields such as military, medicine, process control and manufacturing, bio-terrorism, climate control etc. Sensors are being embedded in countless devices and deployed in locations such as highways, airports, hospitals, laboratories, shopping malls, and office buildings in an effort to bring us closer to the vision of ubiquitous computing. Sensors produce large volumes of data that need to be processed, queried, analyzed and managed in real time. However due to the dynamic and continuous nature of these sensor streams, traditional database management systems (DBMS) are

¹Jess is a popular RETE-based, forward chaining rule engine written in java

not well suited for them. Especially performing join queries online on unbounded continuous streams using a DBMS is impractical as one would need to compare tuples in one infinite stream with the tuples in the other when joining two unbounded streams. Therefore it is more sensible and useful to impose window predicates on these streams and then perform the joins on the reduced snapshots. These snapshots or windows could be *time-based*, e.g., containing tuples arriving in the last t seconds, or *count-based*, in which case only the most recent w tuples are of interest. In either case the windows are continuously sliding, i.e., as new tuples enter the windows, old ones become less important and exit.

As an alternative to DBMSs, data stream management systems (DSMS) [2], [18] have been proposed that evaluate continuous queries on sliding windows of unbounded sensor streams in a centralized manner. In a distributed approach, sensors are considered intelligent entities and form a distributed database. A query is injected into the network and the appropriate sensors respond [16]. However, the sensors themselves can only handle simple queries. In an alternative approach, the sensor data can go through a hierarchy of server nodes, as shown in Figure 1, which gradually increase in bandwidth and computing power [20]. In such a setup, the less expensive operations such as aggregation and filters are placed in server nodes that are closer to the sensors in the hierarchy or sometimes in the sensors themselves. On the other hand, the more expensive operators, e.g. joins, are placed in the nodes closer to the root or in the root. Suppose a query that requires fifteen sensor streams (15-way join) to be joined together, is posed at the root. If the root

has five child nodes then the query can be distributed in a way so that each child joins three sensor streams and forwards its result to the root. Finally, the root joins the five incoming streams from its five children and produces the final result.

Fig. 1. Sensor stream processing hierarchy

A. Background and Motivation

Let us consider an example to see what is involved in processing a sliding window based join query on sensor streams. This will also shed some light on understanding why join tree optimization is important.

Example 1 Consider three streams R_1, R_2, R_3 each with a window size of 5 tuples. The schema and the contents are shown in Figure 2 (b), where the oldest data is at the top of the list. Suppose a three-way join $R_1 \bowtie R_2 \bowtie R_3$ needs to be evaluated such that $R_1.x = R_2.x$ and $R_2.y = R_3.y$. Figure 2 (a) shows one possible join tree. The join results of the intermediate join node $R_1 \bowtie R_2$ are shown in (c). If a new tuple $\langle 1 \rangle$ arrives at R_1 then the following actions will be performed.

- 1) $\langle 1 \rangle$ will be appended to R_1 's window.
- 2) $\langle 1 \rangle$ will be compared against all the tuples in R_2 's window. All joined tuples of the form $\langle R_1.x, R_2.x, R_2.y \rangle$, in this case $\langle 1, 1, 4 \rangle$, will be

added to the intermediate result of $R_1 \bowtie R_2$. They will be propagated along the outgoing edge of the join node and compared to all the tuples in R_3 's window. All matching tuples of the form $\langle R_1.x, R_2.x, R_2.y, R_3.y \rangle$, in this case $\langle 1, 1, 4, 4 \rangle$, will be propagated as part of the result.

- 3) $\langle 3 \rangle$ will be removed from R_1 's window to maintain the window size. As a result of this, the matching tuple $\langle 3, 3, 6 \rangle$ will be removed from the intermediate join node $R_1 \bowtie R_2$.

Fig. 2. Join tree and data for example 1

If a tuple arrives at R_3 the process remains the same except that it will only be compared with the tuples in $R_1 \bowtie R_2$.

The overall processing time for a query is significantly affected by the choice of join tree. In our example, if stream R_1 has the highest arrival rate, then a join tree with the order $(R_2 \bowtie R_3) \bowtie R_1$ maybe preferred over the one shown in Figure 2 (a) since the tuples arriving at R_1 would only be matching with the tuples in $R_2 \bowtie R_3$. On the other hand in the join tree above every tuple in R_1 first has to match with those in R_2 . The matching tuples then have to be matched with the tuples in R_3 , resulting in a much higher cost. However, if the intermediate result size and output rate of $R_2 \bowtie R_3$ is very large compared to that of $R_1 \bowtie R_2$ then the tree in Figure 2 (a) may

still be the better choice.

In order to see how join queries could be useful in processing sensor data, let us look at another example.

Example 2 Consider a scenario where tens of sensors are positioned in various locations of a building for periodically (e.g. every five or ten seconds) measuring temperature, humidity and light intensity. These sensors are sending their sample measurements along with their current voltage readings and timestamps to a central processor (root) that executes continuous queries on these streams of data. However due to intermittent connectivity of the sensors, sometimes data may get lost or garbled. Different sensors may also have different sampling rates. As a result the data arrival rates from different sensors may vary substantially.

A continuous query is posed in the root processor to report the average temperature (or humidity) from a set of sensors at every possible moment. Another similar query could report an anomaly when two sensor readings at the same time differ by more than some predefined threshold. The set of sensors could all be located in a specific area or could be the set of all sensors. If the sensors were producing data in a synchronized manner, then we could just get the last reading from each sensor and find the average or the maximum difference. Since that is not the case, one way to solve the problem would be to use sliding windows for each sensor stream to hold the recent samples and perform join operation on these windows using the timestamp as the join attribute. This would produce joined tuples containing readings from each sensor taken at the same time. Then from these joined tuples, the average or the maximum difference

can be measured.

In this paper, we focus on multi-way join queries on count-based sliding windows of sensor streams. In a centralized system, our algorithms could be used in the root server to produce an optimized join tree. Similarly in a distributed system they could be used at the server nodes that perform the joins, i.e., root node or nodes close to the root .

B. Previous work

A good survey of issues with data management under streaming environment is provided in [2]. A number of different systems have been developed that use novel techniques in dealing with different aspects and issues of continuous query processing: Aurora [7] provides the user with a visual interface to design query plans, TelegraphCQ [8], [18] considers the adaptive aspect of continuous query processing, NiagaraCQ [9] addresses scalability issues in processing multiple queries, Cougar [5] explores the use of Object-Relational abstractions for querying sensor-based systems, STREAM focuses on issues such as operator scheduling [1], caching [4], filter ordering [3], adaptive plan generation, and maintaining statistics [10], ARES [15] gives cost-based online algorithms for multi-join queries. A rate-based query optimization model was proposed for continuous queries in [22]. Kang et al. proposes asymmetric joins to enhance performance and window size manipulation to cope with memory constraints in [17]. Relevant work on join ordering algorithms include *XJoin* [21], and *MJoin* [23]. *XJoin* is similar to the *RETE* match algorithm [13] as it stores intermediate join results to

avoid re-computation. *MJoin* differs from *XJoin* in the way that it has a separate query plan for each stream and it does not maintain the intermediate results at the join nodes. New insertions to a relation are joined with the other $n - 1$ relations in some order to produce the result for the n -way join. *MJoin* is a close counterpart of *TREAT*, another matching algorithm for production systems proposed by Miranker [19]. The benefits of both *TREAT* and *MJoin* stem from the low cost of deleting a tuple due to not having to maintain partial results.

The experiments in eddies [12] and stream [4] suggest that the benefits of storing the partial results outweigh the costs of maintaining them in most cases. Reference [4] considers the whole spectrum between *MJoin* and *XJoin* and selects caches using cost-benefit analysis. In their plans some join nodes cache intermediate results and some do not. But the selected caches have to meet the prefix invariant, i.e., they have to be computed as part of the regular join processing. Their experiments show that these plans are outperformed by the best left-deep *XJoin*. They extend their algorithm by considering caches that are not computed as part of regular join processing, called globally consistent caches. It was shown that these query plans can sometimes significantly outperform *XJoin* when queries involve a small number of streams (four in their experiments). Finding the right combination of globally consistent caches is NP-Hard. Babu et al. [4] uses a pruning heuristics that limits the number of candidate caches to m and uses exhaustive search over the 2^m possible combinations. When the number of streams n is large, m should be large to get enough benefit out of the additional caches. In fact, m is

in the order of n since in the second step of the algorithm they start with up to n globally-consistent caches. This makes both, the algorithm of finding the caches and maintaining the caches, very costly. Therefore when n is large, the best XJoin tree should outperform plans with globally consistent caches.

In this paper we focus on finding the best XJoin or Rete join tree. Most join algorithms and join order generators focus only on linear join trees. However, our trees do not have any restriction on their topologies (bushy); therefore the search space is much larger and produces more efficient join trees.

II. COST FUNCTION AND PROBLEM STATEMENT

We are given n streams $R_1, \dots, R_n$ such that each R_i has a set $A(i)$ of attributes. Let $a(i)$ and $w(i)$ respectively denote the arrival rate and window size of stream R_i . Each of these stream windows can be represented by a leaf node in any join tree. For a non-leaf node s that joins two nodes l and r , let $w(s)$ and $out(s)$ denote its window size (result size) and output rate. Note that l and r themselves could be either leaf nodes or join nodes. For any node s , the output rate $out(s)$ represents the number of tuples that are added to the partial results of s and passed along to the successor node of s per time-unit. Thus for a leaf node, the output rate is the same as its arrival rate.

When a new tuple $\langle q \rangle$ arrives at s from l , it would incur a matching cost of $w(r) \cdot t(s)$, where $t(s)$ is the cost for comparing 1 tuple in l with 1 tuple in r using their common join attributes. Here we assume linear search. If any special data structure is used then the matching cost would change accordingly. For example, if hash indices

are used for r then $w(r)$ would be replaced by $\frac{w(r)}{b(r)}$, where $b(r)$ is the number of hash buckets in the hash index of r .

When a match is found, the joined tuple is inserted to the list of partial results stored at s . The total matching (i.e., comparison) cost incurred at s for all tuples arriving from l and r constitutes the insertion cost $ic(s)$. Similarly, when a tuple $\langle q \rangle$ expires at l or r , all the tuples stored in s that were created from $\langle q \rangle$ have to be deleted constituting the deletion cost $dc(s)$. We use $j(s)$ to denote the join ratio(probability) at node s . Given the above notations, we calculate $ic(s)$, $w(s)$, $dc(s)$, and $out(s)$ as follows:

$$ic(s) = (out(l)f_r(w(r)) + out(r)f_l(w(l)))t(s) \quad (1)$$

$$w(s) = w(l)w(r)j(s) \quad (2)$$

$$dc(s) = (out(l)f_s(w(s)) + out(r)f_s(w(s)))t(s) \quad (3)$$

$$out(s) = (out(l)f_r(w(r)) + out(r)f_l(w(l)))j(s) \quad (4)$$

For nested loop joins $f_r(w(r))$ is just $w(r)$. In case of hash joins $f_r(w(r)) = \frac{w(r)}{b(r)}$ as mentioned above. For B^+ trees, f would be some form of log function. In any case, f is known and can be used to compute $ic(s)$. For example if l is hash-indexed and r is not indexed at all then $ic(s) = (out(l) \cdot w(r) + out(r) \cdot \frac{w(l)}{b(l)}) \cdot t(s)$. In our cost functions we ignore the actual costs of accessing, inserting and deleting a tuple, since they should be negligible compared to matching costs.

Let $S^T = \{s_i^T \mid 1 \leq i \leq n-1\}$ be the set of join nodes for a join tree T , and let $ic(s_i^T)$ and $dc(s_i^T)$ respectively be the insertion and deletion costs of a particular join node s_i^T . Our goal is to minimize the

estimated total cost incurred by a given join tree T during a unit-time period, which is defined as

$$\text{cost}(T) = \sum_{i=1}^{n-1} (ic(s_i^T) + dc(s_i^T)).$$

Optimal Stream Join Tree (OptSJT) Problem:

Given a set of streams $R_1, \dots, R_n$ with arrival rate $a(i)$, window size $w(i)$, and a set $A(i)$ of attributes for each i , the optimization problem asks for a join tree T that minimizes the estimated unit-time cost $\text{cost}(T)$. Minimizing the cost results in maximizing the number of tuples that can be processed per unit-time (i.e., maximizes throughput).

III. NP-HARDNESS RESULT

Throughout this section, the following assumptions are made. We use the window size of a node to denote the node itself, e.g. the root node of the join tree in Figure 3 has window size 2^b and will be called as such. Each input stream has a single common attribute, and for any join node s , $j(s) = 1$ (i.e., cross product) and $t(s) = 1$. This implies that $ic(s) = out(s)$ for any join node s including the root. Hence, the output rate of the query is equal to the insertion cost of the root. The deletion cost of the root is however zero. But, when computing the total cost of a subtree (as will be discussed later), we need to consider the cost incurred by the root of the subtree. So, we use two notations to denote the total cost of a join tree: $QCost(T)$ denoting the cost of T excluding the root and $SCost(T)$ denoting the cost of T including the root.

Lemma 3.1: Given 3 streams R_1, R_2, R_3 with window sizes $2^{a_1}, 2^{a_2}, 2^{a_3}$, respectively, such that (i) $a_3 \geq a_1, a_2$,

(ii) arrival rates are all set to be 1, and (iii) for any join node s , join ratio $j(s) = 1$ and $t(s) = 1$, the optimal join structure with minimal $SCost$ is $(R_1 \bowtie R_2) \bowtie R_3$ (or equivalently $R_3 \bowtie (R_1 \bowtie R_2)$).

Proof: Let $a_1 + a_2 + a_3 = b$. Using the join tree shown in Figure 3, the insertion and deletion costs for the first join node are given by $ic(2^{a_1+a_2}) = 2^{a_1} + 2^{a_2}$, and $dc(2^{a_1+a_2}) = 2^{a_1+a_2+1}$. The costs for the second join node are $ic(2^b) = (2^{a_1} + 2^{a_2})2^{a_3} + 2^{a_1+a_2} = 2^{a_1+a_2} + 2^{a_1+a_3} + 2^{a_2+a_3}$, and $dc(2^b) = (2^{a_1+a_2} + 1)2^b$. Note that $ic(2^b)$ will be the same regardless of what order of join operations is chosen. Hence in order to get the optimal order we have to minimize, $C_0 = 2^{a_1} + 2^{a_2} + 2^{a_1+a_2+1} + (2^{a_1+a_2} + 1)2^b$. It is clear to see that C_0 is minimized if and only if $a_3 \geq a_1, a_2$. ■

Fig. 3. Join tree for 3 streams

A. Transformation from 3Partition to OptSJT

To prove the NP-hardness of the OptSJT problem, we will give a polynomial time transformation from the 3PARTITION problem to the OptSJT problem.

Definition 3.1: 3PARTITION Problem:

We are given a finite set A of $3m$ elements, a bound $b \in \mathbb{Z}^+$, and a positive integer $s(a) \in A$, such that each $s(a)$ satisfies $b/4 < s(a) < b/2$ where $\sum_{a \in A} s(a) = mb$. The following decision problem is called 3PARTITION: Can

A be partitioned into m disjoint sets $S_1, S_2, \dots, S_m$ such that for $1 \leq i \leq m$, $\sum_{a \in S_i} s(a) = b$?

Note that the above constraints on the item sizes imply that every such S_i must contain exactly three elements from A .

Let $A = \{a_1, a_2, \dots, a_{3m}\}$ be a set of $3m$ elements. We assume $m = 2^k$ for some $k \in \mathbb{Z}^+$. (If $m \neq 2^k$ for any integer k , we can construct a new set A' from A such that A' has $3m'$ elements with $m' = 2^k$ and there is a solution to 3PARTITION with A if and only if there is a solution with A' . We omit the proof here.) Note that if $b < 3$, there is no solution, and if $b = 3$, the problem becomes trivial. If $b = 4$, the only possible combination of 3 integers is $(1, 1, 2)$ which does not satisfy $b/4 < a_i < b/2$. Hence, $b \geq 5$. For the rest of section, we will assume that m is a power of 2 and $b \geq 5$.

From $A = \{a_1, \dots, a_{3m}\}$ where $m = 2^k$, an instance to the OptSJT is constructed as follows. The set of input streams is defined as

$$L(T) = A_0 \cup L_0 \cup L_1 \cup \dots \cup L_{k-1},$$

where (i) $A_0 = \{2^{a_i} \mid 1 \leq i \leq 3m\}$, the window size and the arrival rate of each node 2^{a_i} are 2^{a_i} and one, respectively, and (ii) for $0 \leq i \leq k-1$, there are 2^{k-i} nodes in L_i with window size $2^{2^i mb(m+1)^i}$ and arrival rate $2^{2^{i+1} mb(m+1)^i}$. For all possible join nodes s , we set $j(s) = 1$ and $t(s) = 1$. This is a polynomial time transformation since we are creating exactly $m + \sum_{i=0}^{k-1} 2^{k-i} = m + (2m-1)$ leaf nodes. We will call this transformed OptSJT instance as $RInstance$.

Theorem 3.1: There exists a solution for the 3PARTITION problem with A if and only if the optimal solution

of the OptSJT for $RInstance$ has exactly m join nodes with size 2^b .

Proof: Clearly, if there is no solution for the 3PARTITION problem, no join tree with this property exists. Hence, it remains to prove that, if 3PARTITION has a solution, then the optimal solution for OptSJT has the above property. Suppose 3PARTITION has a solution $A = S_1 \cup \dots \cup S_m$ such that $S_i = \{a_{3i-2}, a_{3i-1}, a_{3i}\}$ for $1 \leq i \leq m = 2^k$. We then construct a join tree T^* as follows, assuming $k \geq 1$.

We recursively define a subtree T_i for $1 \leq i \leq k$ such that $3 \cdot 2^i$ nodes from A_0 are included in T_i , where T_k (which is in fact T^*) has 2^{k-i} subtrees T_i 's and each node in A_0 belongs to only one of the T_i 's. A recursive structure of T_i is given in Figure 4 using two T_{i-1} 's with additional two leaf nodes from L_{i-1} , i.e., these two leaf nodes have the window size $2^{2^{i-1} mb(m+1)^{i-1}}$ and arrival rate (i.e., their output rate) $2^{2^i mb(m+1)^{i-1}}$. As a basis case, T_1 is given in Figure 5. The following two lemmas will be used to complete the proof of the theorem.

Fig. 4. T_i for $2 \leq i \leq k$, where $h = 2(m+1)$ and $l, l' \in L_{i-1}$.

Lemma 3.2: For each i , $1 \leq i \leq k$, we have the following bounds, where $h = 2(m+1)$:

$$QCost(T_i) < 2^{(3m+1)h^{i-1}b+1} + 2^{(2m+1)h^{i-1}b+2} \quad (5)$$

$$ic(2^{h^i b}) < 2^{(3m+2)h^{i-1}b+2} \quad (6)$$

$$dc(2^{h^i b}) < 2^{(4m+3)h^{i-1}b+1} + 2^{(3m+4)h^{i-1}b} \quad (7)$$

Fig. 5. T_1

Proof: We will prove the lemma using induction on i . For $i = 1$, the tree T_1 shown in Figure 5 is clearly an optimal join tree due to Lemma 3.1. The insertion cost at node $2^{a_1+a_2}$ is maximum when (a_1, a_2, a_3) is $(1, \frac{b-1}{2}, \frac{b-1}{2})$. Of course a_1 and a_2 can be interchanged without changing any cost. Additionally, the maximum possible size of node $(2^{a_1+a_2})$ is $2^{\frac{2b}{3}}$ and occurs when $(a_1, a_2, a_3) = (\frac{b}{3}, \frac{b}{3}, \frac{b}{3})$. Consequently, we get the following inequalities.

$$ic(2^{a_1+a_2}) \leq 2^{\frac{b-1}{2}} + 2 = 2^{\frac{b}{2}-\frac{1}{2}} + 2$$

$$dc(2^{a_1+a_2}) \leq 2^{\frac{2b}{3}+1}$$

$$\begin{aligned} ic(2^b) &\leq 2^{\frac{2b}{3}} + (2^{\frac{b-1}{2}} + 2)2^{\frac{b-2}{2}} \\ &= 2^{\frac{2b}{3}} + 2^{b-\frac{3}{2}} + 2^{\frac{b}{2}} < 2^b \text{ for } b \geq 5 \end{aligned}$$

$$dc(2^b) \leq (2^{\frac{b}{2}-\frac{1}{2}} + 3)2^b < 2^{2b} + 3(2^b)$$

$$ic(2^{(m+1)b}) < 2^{2mb+b} + 2^{mb+b}$$

$$dc(2^{(m+1)b}) < 2^{3mb+b} + 2^{mb+2b}$$

The $SCost$ for the subtree rooted by node $2^{(m+1)b}$ includes the insertion and deletion cost at all the nodes in the subtree. By adding the above inequalities we get the following.

$$SCost(2^{(m+1)b}) < 2^{3mb+b} + 2^{2mb+b+1}$$

As $QCost$ for node $2^{2(m+1)b}$ is the sum of the $SCosts$ of the roots $2^{(m+1)b}$ of its two children subtrees, we have

$$\begin{aligned} QCost(2^{2(m+1)b}) &< 2(2^{3mb+b} + 2^{2mb+b+1}) \\ &= 2^{3mb+b+1} + 2^{2mb+b+2} \end{aligned}$$

$$\begin{aligned} ic(2^{2(m+1)b}) &< 2(2^{2mb+b} + 2^{mb+b})(2^{(m+1)b}) \\ &< 2^{3mb+2b+2} \end{aligned}$$

$$\begin{aligned} dc(2^{2(m+1)b}) &< 2(2^{2mb+b} + 2^{mb+b})(2^{2(m+1)b}) \\ &< 2^{(4m+3)b+1} + 2^{(3m+4)b} \end{aligned}$$

Now suppose the lemma holds for $i = 2^{k-1}$. Then we can get the following inequality for the $SCost$ of a join node at level $k-1$ by adding up the $QCost$, ic and dc .

$$\begin{aligned} SCost(2^{h^{k-1}b}) &< 2^{(4m+3)h^{k-2}b+1} + 2^{(3m+4)h^{k-2}b} \\ &\quad + 2^{(3m+2)h^{k-2}b+2} + 2^{(3m+1)h^{k-2}b+2} \\ &< 2^{(4m+3)h^{k-2}b+1} + 2^{(3m+4)h^{k-2}b+1} \end{aligned}$$

From Figure 4, we derive the following for level k .

$$\begin{aligned} ic(2^{(m+1)h^{k-1}b}) &< (2^{(3m+2)h^{k-2}b+2})(2^{mh^{k-1}b}) + (2^{2mh^{k-1}b})(2^{h^{k-1}b}) \\ &= 2^{(2m+1)h^{k-1}b} + 2^{(2m^2+5m+2)h^{k-2}b+2} \\ &< 2^{(2m+1)h^{k-1}b} + 2^{(m+2)h^{k-1}b} \end{aligned}$$

$$\begin{aligned} dc(2^{(m+1)h^{k-1}b}) &< (2^{(3m+2)h^{k-2}b+2} + 2^{2mh^{k-1}b}) \cdot (2^{(m+1)h^{k-1}b}) \\ &= 2^{(3m+1)h^{k-1}b} + 2^{(2m^2+7m+4)h^{k-2}b+2} \\ &< 2^{(3m+1)h^{k-1}b} + 2^{(m+3)h^{k-1}b} \end{aligned}$$

By adding the $SCost$ of the subtree rooted at node $2^{h^{k-1}b}$, and the ic and dc at node $2^{(m+1)h^{k-1}b}$, we get the following equation for $SCost$ of the subtree rooted at node $2^{(m+1)h^{k-1}b}$.

$$\begin{aligned} SCost(2^{(m+1)h^{k-1}b}) & < 2^{(3m+1)h^{k-1}b} + 2^{(m+3)h^{k-1}b} + 2^{(2m+1)h^{k-1}b} \\ & \quad + 2^{(m+2)h^{k-1}b} + 2^{(4m+3)h^{k-2}b+1} + 2^{(3m+4)h^{k-2}b+1} \\ & < 2^{(3m+1)h^{k-1}b} + 2^{(2m+1)h^{k-1}b+1} \end{aligned}$$

Finally,

$$\begin{aligned} QCost(2^{h^k b}) & = 2(SCost(2^{(m+1)h^{k-1}b})) \\ & < 2^{(3m+1)h^{k-1}b+1} + 2^{(2m+1)h^{k-1}b+2} \end{aligned}$$

$$\begin{aligned} ic(2^{h^k b}) & < 2(2^{(2m+1)h^{k-1}b} + 2^{(m+2)h^{k-1}b}) \cdot (2^{(m+1)h^{k-1}b}) \\ & < 2^{(3m+2)h^{k-1}b+2} \end{aligned}$$

$$\begin{aligned} dc(2^{h^k b}) & < 2(2^{(2m+1)h^{k-1}b} + 2^{(m+3)h^{k-1}b}) \cdot (2^{h^k b}) \\ & < 2^{(4m+3)h^{k-1}b+1} + 2^{(3m+4)h^{k-1}b+1} \end{aligned}$$

Therefore, the induction holds, and this completes the proof of Lemma 3.2. ■

Note that the window size of the root node of any join tree for the $RInstance$ is $2^{2^k(m+1)^k b}$.

Lemma 3.3: Let R_z be a new stream with window size $2^{2^k m(m+1)^k b}$ and arrival rate $2^{2^{k+1} m(m+1)^k b}$. Then, R_z is the last node joined in any optimal join tree (i.e., a join tree with the minimum $SCost$) for $RInstance \cup$

$\{R_z\}$.

Proof: Assume $h = 2(m+1)$. Let T' be a join tree such that R_z joins with some other node before joining to the root. Then in T' , there must be an intermediate join node of size $2^{mh^k b+x}$ which eventually joins to the root, where $1 \leq x < h^k b$. Therefore, we have the following:

$$\begin{aligned} out_{T'}(2^{mh^k b+x}) & > 2^{2mh^k b+x} \\ SCost_{T'}(2^{(m+1)h^k b}) & > dc(2^{(m+1)h^k b}) \\ & > out(2^{mh^k b+x}) \cdot (2^{(m+1)h^k b}) \\ & > 2^{(3m+1)h^k b+x} \\ & > SCost_{T^*}(2^{(m+1)h^k b}) \end{aligned}$$

This completes the proof of Lemma 3.3. ■

Fig. 6. T' in Case 1 where $h = 2(m+1)$ and $l, l' \in L_{k-1}$.

Using Lemmas 3.2 and 3.3, we next proceed to show that for any join tree $T' \neq T^*$ constructed for the $RInstance$, $QCost(T^*) < QCost(T')$ and $SCost(T^*) < SCost(T')$. Note that T' also has two subtrees T'_L and T'_R . l and l' from L_{k-1} must also be included in T' . Thus two cases are considered. In Case 1, both l and l' are in one of the subtree, say T'_R . (See Figure 6.) In Case 2, l belongs to T'_L and l' belongs to T'_R . **Case 1:** Note that both sub-trees are non-empty implying $0 \leq x < 2h^{k-1}b$, and we observe the

following:

$$\begin{aligned}
QCost_{T'}(2^{h^k b}) &> dc(2^{2mh^{k-1}b+x}) \\
&> 2^{4mh^{k-1}b+x} \\
&> QCost_{T^*}(2^{h^k b}) \\
out_{T'}(2^{2mh^{k-1}b+x}) &> 2(2^{2mh^{k-1}b})(2^{mh^{k-1}b}) \\
&= 2^{3mh^{k-1}b+1} \\
SCost_{T'}(2^{h^k b}) &> dc_{T^*}(2^{h^k b}) \\
&> 2^{3mh^{k-1}b+1}(2^{h^k b}) \\
&= 2^{(5m+2)h^{k-1}b+1} \\
&> SCost_{T^*}(2^{h^k b})
\end{aligned}$$

Case 2: In this case, by virtue of Lemma 3.3, we don't need to consider any other form for the left or right subtree, where the big nodes, l and l' , are not the last nodes being joined. The window sizes of the roots of left and right subtrees in this case is defined as $2^{(m+1)h^{k-1}b+x}$ and $2^{(m+1)h^{k-1}b-x}$. (Note that $2^{(m+1)h^{k-1}b}$ is the size of the two children nodes of the final root of T^* , as shown in Figure 4. Hence, we have $0 \leq x \leq h^{k-1}b$. Depending on the value of x , we consider three sub-cases.

Case 2.1: $x = h^{k-1}b$

Node $2^{ma^{k-1}b}$ of the right subtree becomes the only node on the right and directly connects to the root node. So,

$$\begin{aligned}
QCost_{T'}(2^{h^k b}) &> dc_{T^*}(2^{(m+1)h^{k-1}b+x}) \\
&= dc_{T^*}(2^{(m+2)h^{k-1}b}) \\
&> 2^{(m+2)h^{k-1}b+(2m)h^{k-1}b} \\
&= 2^{(3m+2)h^{k-1}b} \\
&> QCost_{T^*}(2^{h^k b})
\end{aligned}$$

$$\begin{aligned}
out_{T'}(2^{(m+1)h^{k-1}b+x}) &> 2^{(2mh^{k-1}b+2h^{k-1}b)} \\
&= 2^{h^k b}
\end{aligned}$$

$$\begin{aligned}
SCost_{T'}(2^{h^k b}) &> dc_{T^*}(2^{h^k b}) \\
&> 2^{h^k b+h^k b} \\
&= 2^{(4m+4)h^{k-1}b} \\
&> SCost_{T^*}(2^{h^k b})
\end{aligned}$$

Case 2.2: $1 \leq x < h^{k-1}b$

This gives us the following:

$$\begin{aligned}
QCost_{T'}(2^{h^k b}) &> dc_{T^*}(2^{(m+1)h^{k-1}b+x}) + dc_{T^*}(2^{(m+1)h^{k-1}b-x}) \\
&> 2^{2mh^{k-1}b+x+(m+1)h^{k-1}b} + 2^{2mh^{k-1}b-x+(m+1)h^{k-1}b} \\
&> 2^{(3m+1)h^{k-1}b+x} + 2^{(3m+1)h^{k-1}b-x} \\
&> QCost_{T^*}(2^{h^k b})
\end{aligned}$$

Case 2.3: $x = 0$

In this case, T' looks like T^* . However subtrees rooted at the two nodes of size $2^{h^{k-1}b}$ in T' may not conform to T^* . So, assume they are different. We will then use induction on k to prove $QCost_{T'}(2^{h^k b}) > QCost_{T^*}(2^{h^k b})$. When $k = 1$, it is trivial. So, assume that for any $m = 2^i$, for $1 \leq i \leq k-1$, $QCost(T') > QCostT^*$ and $SCost(T') > SCostT^*$. Consider $i = k$. Note that T' and T^* both have the same insertion costs at the root node $2^{a^k b}$, since the output rate of the root node must be same at both trees. Consequently, the sum of the insertion costs, in T' , of nodes $(2^{(m+1)h^{k-1}b})$ is equal to that in T^* . As a result, the sum of dc 's are also equal for each of nodes in the last two levels for the two trees. This

makes the sum of all the costs of nodes in the last two levels to be equal in both trees. For T' , the two subtrees rooted at nodes $2^{h^{k-1}b}$ both contain exactly 2^{k-i-1} leaf nodes with window size equal to 2^{mbh^i} for $0 \leq i \leq k-1$, and original nodes from A_0 constituting a total window size of 2^{k-1} ; otherwise the two subtrees would not have the same root size. This means each subtree represents an *RInstance* with $m = 2^{k-1}$ as does the corresponding subtrees in T^* . Consequently, by combining our induction hypothesis with the costs of nodes in the last two levels, we get $QCost_{T'} > QCost_{T^*}$, and $SCost_{T'} > SCost_{T^*}$ for $m = 2^k$, which completes the induction.

This completes the proof of the theorem. ■

Theorem 3.1 proves the validity of our transformation from 3Partition to OptSJT, which establish the following NP-hardness result.

Theorem 3.2: The OptSJT problem is NP-hard even if the join ratio at each join node is one (i.e., cross product) and all input streams have only one common attribute.

IV. ALGORITHMS FOR GENERATING JOIN TREES UNDER STABLE STREAM CONDITIONS

The difficulty of finding optimal join trees has driven researchers towards limiting their searches to left deep linear join trees. Usually some low cost heuristic is used to come up with a linear XJoin tree. The most common ones are the three variants of *XJoin*, namely sorting the streams in ascending order of window sizes, in ascending order of arrival rates and in ascending order of expected

join ratio. We will respectively call these three heuristics *XJoinW*, *XJoinAR* and *XJoinJR*. *XJoinW* first sorts the streams according to their window sizes in ascending order, and then creates a left-deep linear tree materializing all partial results using this order. For instance, if $R_1, R_2, \dots, R_4$ are four streams in ascending order of window size, *XJoinW* will produce the join tree $((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4$. Similarly, *XJoinAR* sorts the streams by arrival rates and then creates a left-deep linear tree. Finally *XJoinJR* sorts by pairwise join ratio. However, since the cost of a join tree depends on all three of these factors, sorting by any one of them can be short-sighted. For example, if the streams with small window sizes have really high arrival rates or high join ratios then sorting by window size will not produce the desired effect. Similar reasonings also hold for the other two heuristics.

A. The OptDP Algorithm

Our optimal dynamic programming algorithm *OptDP* considers all possible join trees for a given set of streams under stable conditions. Notice for any set of n streams, the output rate and result size of the root node are same regardless of the join tree. Consequently once an optimal join tree for a subset p of n streams is known, it does not have to be recomputed. However, *OptDP* has to consider all possible subsets of a given set to find the optimal tree.

Let T_S denote the optimal join tree for a set of streams S and let $C(T_S)$ be its cost. Let $A = \{p | p \subset S \text{ \& } p \neq \Phi\}$, i.e., A is the set of all non-empty proper subsets of S . Let $ic_{p,q}$ and $dc_{p,q}$ denote the insertion and deletion costs

associated to a join node that has the set p of streams in its left subtree and q in its right subtree. Then the following recurrence relation holds:

$$\begin{aligned} C(T_S) &= 0, \text{ when } |S|=1 \\ C(T_S) &= \min_{p \in A} \{C(T_p) + C(T_{S-p}) \\ &\quad + ic_{p,S-p} + dc_{p,S-p}\} \end{aligned}$$

Theorem 4.1: The execution time of *OptDP* is in the order of $(3^n - 2^{n+1} + 1)/2$.

Proof: The number of binary partitions that has to be tried from a set of n elements is

$$\begin{aligned} N &= \sum_{i=2}^n \binom{n}{i} (2^{i-1} - 1) \\ &= \sum_{i=2}^n \binom{n}{i} 2^{i-1} - \sum_{i=2}^n \binom{n}{i} \end{aligned} \quad (8)$$

From binomial theorem we know that,

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} \cdot x^k \cdot y^{n-k} \quad (9)$$

Letting $x = 2$, and $y = 1$ we get the following

$$\begin{aligned} 3^n &= \sum_{k=0}^n \binom{n}{k} \cdot x^k \\ \Rightarrow 3^n &= \sum_{k=2}^n \binom{n}{k} \cdot x^{k+1} + nx \\ \Rightarrow \sum_{k=2}^n \binom{n}{k} \cdot 2^k &= 3^n - 1 - 2n \\ \Rightarrow \sum_{k=2}^n \binom{n}{k} \cdot 2^{k-1} &= (3^n - 1 - 2n)/2 \end{aligned} \quad (10)$$

From Equation (9), by setting $x = 1$ and $y = 1$ we get

$$\begin{aligned} 2^n &= \sum_{k=0}^n \binom{n}{k} \\ \Rightarrow 2^n &= \sum_{k=2}^n \binom{n}{k} + \binom{n}{0} + \binom{n}{1} \\ \Rightarrow \sum_{k=2}^n \binom{n}{k} &= 2^n - 1 - n \end{aligned} \quad (11)$$

By combining Equations (8), (10) and (11) we get the following,

$$\begin{aligned} N &= (3^n - 1 - 2n)/2 - (2^n - 1 - n) \\ &= (3^n - 2^{n+1} + 1)/2 \end{aligned}$$

■

Table I shows the execution time (in milliseconds) required by *OptDP* to find the optimal join tree as a function of stream count.

<i>StreamCount</i>	8	10	12	14
<i>Executiontime</i>	25	154	1607	16870

TABLE I
STREAM COUNT VS EXECUTION TIME FOR OPTDP

B. The XGreedyJoin algorithm

Here we present our greedy algorithm, *XGreedyJoin*, which runs in polynomial time and is still able to balance all three factors, namely arrival rate, window size and join ratio, that determine the efficiency of a join tree in a streaming environment. Let each stream R_i be represented by a leaf node n_i , each with its output rate, window size and attribute list, where the output rates are the measured output rates during runtime. Let $cdts$ be the set of currently available nodes that can be joined. Initially it contains all the leaf nodes. At each step, it chooses the pair of nodes that would minimize the sum of insertion and deletion costs incurred at the resulting join node. Remember that insertion and deletion costs take all the three factors into account. Figure 7 shows the pseudo code for XGreedyJoin. In the pseudo code, function *join* creates a new join node, updates its result size

(window), output rate and attribute list, and connects it to its left and right children.

XGreedyJoin is an $O(n^3)$ algorithm in the form that is given in Figure 7. However, it is not necessary to recompute all the pairwise costs (the two inner loops) after each iteration of the outermost loop. All pairwise costs are only computed during the first iteration. At each subsequent iteration p the pairwise costs from the previous iteration are already there. Therefore only the pairwise costs for the newly created join node with the other nodes in the candidate list have to be computed. This brings down the complexity to $O(n^2)$.

```

cdts = {n1, n2, n3, ..., nn};
numnodes = n;
for ( p = 1 ; p <= n - 1 ; p=p+1 )
    for ( i = 1 ; i < numnodes ; i++)
        for ( j = i + 1 ; j < numnodes ; j++)
            find the pair of nodes, leftCdt and rightCdt,
            with minimum incremental cost;
    Node jnode = join(leftCdt, rightCdt);
    cdts = cdts + jnode - leftCdt - rightCdt;
    numnodes = numnodes - 1;

```

Fig. 7. pseudo code for XGreedyJoin

V. ADAPTIVE OPTIMIZATION OF JOIN TREES UNDER VARYING STREAM CHARACTERISTICS

Our algorithms in the previous section assume stable conditions. However, in real stream environment, conditions vary and can be quite unpredictable. An optimal join tree may become drastically inefficient within a short period. Therefore, systems have to adapt continuously to the current conditions to maintain high throughput. Our system continuously monitors the streams and reevaluates stream arrival rates. It periodically calls the

optimizing algorithm to adapt to the current stream conditions. Next we discuss how the join ratios are approximated, which is followed by a discussion on dynamic plan migration.

A. Estimating expected intermediate result size, join ratio and arrival rates

When estimating intermediate result size and join ratios, we make standard assumptions regarding containment and preservation of value sets and uniform distribution of attribute values. Although advanced histogram based techniques (e.g. equi-depth, v-optimal, max-diff etc.) can be used for improved accuracy, we take the simpler approach to reduce runtime overhead. In a dynamic environment, the maintenance of histograms can be quite costly. However, we wish to investigate the tradeoff between speed and accuracy in regard to join ratio estimation in future work.

Using the containment assumption, it is concluded that each group of distinct valued tuples belonging to the window with the smaller number of distinct values, joins with some group of tuples in the other window [6], [14]. We measure the number of distinct values $dv_{R_i, X}$ in the current window of each stream R_i for each attribute $X \in R_i$. We also find the maximum value $max_{R_i, X}$ that represents the range for attribute X in the current window of stream R_i . This is done only before calling our optimizing algorithm to reduce runtime overhead. Assuming uniform distribution we adjust the *window size* and *dv* values of the window with the bigger range by multiplying it with the ratio of the smaller range to

the bigger range. Using the containment assumption we calculate the join ratio jr and expected result size rs of the two streams R_i and R_j as follows:

$$jr = \frac{1}{\prod_{X \in J} \max(dv_{R_i, X}, dv_{R_j, X})}$$

$$rs = w(i) \cdot w(j) \cdot joinratio$$

where $J = \{A(i) \cap A(j)\}$ and $A(i)$ is the set of attributes in R_i . Note that each $dv_{R_i, X}$ and w_i are the adjusted values. For the intermediate join nodes the dv and max values for each joining attribute X is measured using the following equations, where $left$ and $right$ represent the two children of the join node.

$$dv_{join, X} = \min(\min(dv_{left, X}, dv_{right, X}), rs)$$

$$max_{join, X} = \min(max_{left, X}, max_{right, X})$$

For each non-joining attribute Y , the equations are as follows, where $child \in \{left, right\}$,

$$dv_{join, Y} = \min(dv_{child, Y}, rs)$$

$$max_{join, Y} = max_{child, Y}$$

Arrival count is incremented after each arrival for each stream. From these arrival counts, arrival rates are estimated after each x seconds over a sliding window of y seconds in a straight forward manner. For example, for our synthetic experiments, x and y were both set to 5 seconds.

B. Migration of intermediate results during transition of join tree

When a new join tree is created, the new join nodes have to be populated with the appropriate intermediate results in order to maintain correctness. This process

is known as the migration process [24]. A migration strategy must guarantee that exactly the same results would be generated by the system during and after the migration is done. A common strategy used for migration, first stops executing the joins in the old tree, and performs the joins on the current tuples in the windows according to the new join tree to populate its intermediate nodes. Although correct, this is a waste of both time and storage. A much better and advanced approach, the *moving state* strategy, was given in [24], which is implemented in our system. Our objective is to try to reuse as much intermediate result from the old tree to avoid re-computation. We briefly go over this strategy below.

Reuse existing node: First we try to see if a node in the current tree is exactly the same as an existing node in the old tree. We say two join nodes are *exactly same* if they have the same left and right children. For example, in the two trees from Figure 8 (a) and (b), the nodes $R4 \bowtie R5$ are same. In these cases we can reuse the node from the old tree, and thus avoid the computation necessary to create a new node and to re-generate its results.

Reuse existing results: If the previous case is not satisfied, we check if the current node is *semantically same* as any node in the old tree. Two join nodes are semantically same if they produce the same results despite having different structures. In Figure 8 (a) and (b), $R_1 \bowtie R_2 \bowtie R_3$ and $R_2 \bowtie R_3 \bowtie R_1$ are semantically same. If such a node is found, we can reuse its results, but we still have to create a new join node.

Re-generate results: If none of the previous cases are

satisfied, we create a new join node and re-generate its results by performing the join.

Fig. 8. migrating partial results between trees

C. RunTime Overhead due to adaptive re-optimization

The runtime overhead of our system comes from three factors.

Maintain stream characteristics: For each stream, we have to measure its arrival rate and the *distinct value (dv) counts*, and *max values* for each attribute in its current window. Overhead due to arrival rate estimation is negligible since it is done periodically (5 seconds for our experiments). Getting *max* and *dv* counts is only done before re-optimizing and thus does not incur too much overhead.

Performing re-optimizing algorithm: Re-optimizing algorithm is also called periodically. For our experiments it is set to 5 seconds.

Constructing and populating new join tree: After the re-optimizing algorithm suggests the best structure for the current conditions, a new join tree has to be created according to the suggested tree. Populating the new tree during tree migration will cause additional overhead, which we try to minimize as discussed in Section V-B. However if the suggested tree is only marginally better than the current one, we should not create a new tree. In

our current implementation, we create a new tree only if the estimated cost of the suggested tree is more than 10% less than the estimated cost of the current tree at work under current conditions.

Our experiments show that the added overhead for these tasks is negligible compared to the performance gains.

VI. EXPERIMENTAL RESULTS

We implemented our algorithms along with *XJoinAR*, *XJoinW* and *XJoinJR* in the sliding window based stream query processor ARES [15]. *XJoinAR*, *XJoinW* and *XJoinJR* were discussed in section IV.

A. Synthetic tests

We divided our synthetic tests into the following five categories of varying conditions. For the first four category we compare the performance results of the join trees chosen by *XGreedyJoin* to the optimal trees produced by *OptDP*, the left deep join trees constructed by *XJoinAR* and *noOpt*, where *noOpt* is the basic *XJoin* or *Rete* algorithm that does not perform any optimization. It creates a left-deep tree using the stream order provided in the query and uses the same tree throughout the simulation. All the other algorithms in our experiments go through adaptive re-optimizations. For the first categories of experiments we do not evaluate *XJoinW* and *XJoinJR* since for each individual test-case we set the window sizes and *dv* counts to be equal across all streams. In order to show the performance gain over *noOpt*, the results are shown in terms of the ratio of tuple processing time in the tree produced

by an optimization algorithm to those of *noOpt*. For the fifth category, we use scenarios with dynamically changing mixed stream characteristics and compare all the algorithms. In each iteration of a for loop, a stream R_i is chosen with probability $\frac{a(i)}{\sum_{j=1}^n a(j)}$. Once a stream R_i is chosen, we generate a tuple with an integer attribute value chosen uniformly from $\{1 \cdots dv_i\}$ for that stream to produce data with uniform distribution and to ensure our containment assumption. For each testcase, 1 million tuples are generated and the processing time is measured. Tests are handled in a saturated manner, i.e., the new tuple is sent as soon as the old tuple has been processed. In other words, the throughput rates represent the maximum load the system can handle.

Note that due to the time spent in gathering statistics, running optimization algorithms and populating new join trees, our system incurs some runtime overhead, which increases tuple processing time (decreases throughput). However our results show that the added overhead is negligible compared to the performance gains. All the results shown hereinafter account for all types of overheads, i.e., we include the time spent on the overhead tasks in our tuple processing times.

Runtime overhead is a function of how often (reoptimization interval) the reoptimization algorithm is run to adapt to the current conditions. We ran several experiments with eight streams and reoptimization interval ranging from 1 second to 10 seconds. The results are shown in figure 9. Overhead is measured as a percentage of the time spent in optimization and statistics gathering out of the total execution time. Overhead for *OptDP* grows exponentially as the reoptimization interval grows

smaller. *OptDP*'s overhead also grows exponentially as the number of streams increase due to the exponential nature of the algorithm. It goes up-to around 3% for ten streams. For all the other algorithms overhead stays below 0.3%.

Fig. 9. Runtime overhead for queries with 8 streams

1) *Varying join ratio*: Five streams with relative arrival rates $\{1, 10, 1, 1, 1, 1, 10, 1\}$ each with window size = 2000 and equal dv were used for each test case. Join ratios were varied among the test cases by changing the dv . Figure 10 shows that as the join ratios increase (dv decreases) the performance of the optimized trees from both *XGreedyJoin* and *OptDP* algorithms get better compared to the *XJoinAR* and *noOpt*. When dv equals 1300 and 1700 we see a sudden increase in improvement as *noOpt* and *XJoinAR* did not perform very well for those scenarios. As the join ratios decrease the partial result sizes decrease to the point where it does not require much optimization. The performance increase varies between 20-80% over *XJoinAR* for both algorithms, with *XGreedyJoin* performing almost as well as the optimal algorithm in all the testcases.

Fig. 10. Varying Join Ratio

Fig. 11. Varying Window Size

2) *Varying window size*: Streams with similar arrival characteristics as above were used here. Join ratios were kept the same with $dv_i = 1000$ for all streams and window sizes were varied to see the effect. Figure 11 shows the results. Performance enhancements over *XJoinAR* were observed across all window sizes, going from 33% to about 80%. Performance increases as windows get larger as expected. Again *XGreedyJoin* closely follows *OptDP* in performance. Notice that performance of *XJoinAR* compared to *noOpt* improves at first with increasing window size due to benefits of placing the fastest streams at the end. However it starts to decrease after a while. This happens because as the window size is increased, the intermediate result sizes down the tree grow exponentially. When the faster streams near the root join with these nodes with large intermediate results, the overall processing time drastically increases.

3) *Varying Number of Streams*: For this category all window sizes were 2000 and dvs were set to 1000. For each case, two very fast streams were used with relative rate of 10, with one at the second position of the order and the other one at the second last position. All other

relative rates were set to 1. Figure 12 shows that for both algorithms a greater improvement in performance is noticed with larger number of streams.

Fig. 12. Varying the Number of Streams

4) *Varying arrival rates*: We examined eight different test cases $T_1 - T_8$ with varied arrival rates, as shown in Table II. The window sizes were 2000 and dv counts were set to 1000 as before. We tried to cover the whole spectrum of scenarios from fast streams at the end to fast streams at the beginning of the order. Figure 13 shows the results for these tests. The performance improvements were really high for all of the tests except for the last one which uses six streams. Even though

XJoinAR sorts by the arrival rate, the other algorithms can get much better performance by producing more balanced trees.

Sample Tests	Relative Arrival Rates							
	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
T_1	1	5	4	2	7	10	7	7
T_2	1	5	15	2	1	5	5	15
T_3	10	7	2	4	4	1	2	10
T_4	15	1	4	5	2	4	1	15
T_5	1	7	4	15	10	5	4	15
T_6	7	5	4	4	5	4	2	1
T_7	15	4	2	2	10	5	7	5
T_8	7	7	7	5	15	15	-	-

TABLE II
SAMPLE TESTS WITH VARIED ARRIVAL RATES

Fig. 13. Varying Arrival Rate

5) *Mixed stream characteristics*: We extensively tested the join trees generated by the algorithms that we discussed in this paper under different stream characteristics. Figure 14 shows the results for eight representative samples from these experiments with eight streams. All of the tests are 8-way joins of the form $R_1(A) \bowtie_A R_2(A) \bowtie_A \dots \bowtie_A R_8(A)$, using integer attributes. Relative arrival rates a_i , window sizes w_i and DV counts dv_i were varied in these experiments, where $a_i \in \{1 \dots 15\}$, $w_i \in \{1400 - 2000\}$ and $dv_i \in \{800 - 1400\}$. For

these testcases a number z is randomly chosen from the range 1-20. The arrival rates were permuted among the streams after z seconds to simulate volatile conditions. After each shuffle a new z is randomly chosen and the process continues. Figure 14 shows the results from these experiments. We noticed that *XGreedyJoin* performed significantly better than the *XJoin* variants and almost as well as *OptDP*. On average, trees from *XGreedyJoin* and *OptDP* can handle three times the load of the *XJoin* trees. Figure 15 shows the results for sample experiments with five streams. These testcases were constructed from the tests in Figure 14 by taking only the first five streams. In these results, the performance improvements decline due to the reduction in the number of streams.

Tests using sensor data: The scenario for our exper-

Fig. 14. Performance comparison with mixed stream characteristics for 8 streams

iments is exactly the scenario that was described in example 2. We used a real sensor dataset from Intel Research, Berkeley Lab. The Intel Lab dataset contains readings from 54 sensors in the Intel Research, Berkeley Lab [11]. Each sensor sample has the attributes *temperature*, *humidity*, *light*, *voltage*, *sensorID* and *timestamp*. To evaluate the various algorithms we ran join queries

Fig. 15. Performance comparison with mixed stream characteristics for 5 streams

of the following semantic form:

Select * **from** $S_1[100], S_2[100], \dots, S_n[100]$

Where $(S_1.ts = S_2.ts = \dots = S_n.ts)$

where ts is the timestamp of a tuple, n is the number of sensor streams and sliding window size is 100 for each stream. For each n , where $n \in \{10, 15, 20\}$, we ran 10 different tests. For each test, n sensor streams were randomly chosen from the set of 54 sensors. Figure 16 reports the average throughput in tuples/sec for the different algorithms. Since all window sizes are same, we did not test the *XJoinW* algorithm. Also, we do not report the result for *OptDP* for $n = 15$ and 20 , as the exponential nature of this algorithm makes it unsuitable for such large n . For all n , *XGreedyJoin* significantly outperforms *XJoinAR* and *XJoinJR*. For $n = 10$, it is almost as good as the optimal solution *OptDP*.

VII. FUTURE WORK AND CONCLUSION

We have considered the problem of finding optimal join trees for processing continuous queries over streaming sensor data and presented two algorithms, *XGreedyJoin* and *OptDP*. Our experiments show that our methods used in our system for adaptive re-optimization is inexpensive and the join trees produced

Fig. 16. Comparing throughput

by *XGreedyJoin* perform almost as well as optimal join trees produced by *OptDP*, and significantly better than *XJoin* based heuristic algorithms under varying stream conditions. In future, we would like to predict the need for adaptivity based on the variability of stream conditions. Also, our assumptions regarding containment of values, preservation of values, and uniform distribution are not going to be always valid in reality. For this, we would like to explore various histogram techniques in estimating result sizes, and join ratios, and see their effects on performance and runtime overhead. It also remains to be seen how our system would react when connected to a realistic sensor network, where energy consumption issues control data transmission rates and data quality.

REFERENCES

- [1] B. Babcock, S. Babu, M. Datar, and R. Motwani, "Chain: Operator scheduling for memory minimization in data stream systems," in *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, 2003.
- [2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data streams," in *Proc. of ACM Symp. on Principles of Database Systems*, 2002, pp. 1–16.

- [3] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive ordering of pipelined stream filters," in *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, 2004.
- [4] S. Babu, K. Munagala, J. Widom, and R. Motwani, "Adaptive caching for continuous queries," in *Proc. of 21st Intl. Conf. on Data Engineering (ICDE'05)*, 2005.
- [5] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *Proc. of 2nd International conference on Mobile data management*, 2001.
- [6] N. Bruno and S. Chaudhuri, "Exploiting statistics on query expressions for optimization," in *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, 2002.
- [7] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring streams - a new class of data management applications," in *Proc. of VLDB*, 2002, pp. 215–226.
- [8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "Telegraphcq:continuous dataflow processing for an uncertain world," in *Proc. of 1st Biennial Conf. on Innovative Data Syst. Res. (CIDR)*, 2003.
- [9] J. Chen, D. DeWitt, F. Tian, and Y. Wang, "Niagaraqc:a scalable continuous query system for internet databases," in *Proc. of ACM Intl. Conf. on Management of Data*, 2000, pp. 379–390.
- [10] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," in *Proc. of 13th SIAM-ACM Symp. on Discrete Algorithms*, 2002, pp. 635–644.
- [11] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong, "Model-driven data acquisition in sensor networks," in *Proc. of VLDB*, 2004.
- [12] A. Deshpande and J. Hellerstein, "Lifting the burden of history from adaptive query processing," in *Proc. of VLDB*, 2004.
- [13] C. L. Forgy, "Rete: A fast algorithm for the many pattern/match object pattern match problem," *Artificial Intelligence*, vol. 19, pp. 17–37, 1982.
- [14] L. Golab and T. Ozsu, "Processing sliding window multi-joins in continuous queries over data streams," in *Proc. of VLDB*, 2004.
- [15] J. S. Gomes and H.-A. Choi, "Maximizing throughput for queries over streaming sensor data," 2006, to Appear in *Proc. of MASS2006 workshops*.
- [16] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed diffusion for wireless sensor networking," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 2–16, Feb. 2003.
- [17] J. Kang, J. Naughton, and S. Viglas, "Evaluating window joins over unbounded streams," in *Proc. of VLDB*, 2002.
- [18] S. Madden and M. J. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data," in *Proc. of ICDE*, 2002.
- [19] D. P. Miranker, "Treat: a better match algorithm for ai production systems," in *Proc. of AAAI-87*, 1987, pp. 42–47.
- [20] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network query processing," in *Proc. of PODS*, June 2005.
- [21] T. Urhan and M. Franklin, "Xjoin: A reactively scheduled pipelined join operator," in *IEEE Data Engineering Bulletin*, vol. 23, no. 2, 2000.
- [22] S. Viglas and J. Naughton, "Rate-based query optimization for streaming information sources," in *Proc. of ACM Intl. Conf. on Management of Data*, 2002, pp. 37–48.
- [23] S. Viglas, J. Naughton, and J. Burger, "Maximizing the output rate of multi-way join queries over streaming information sources," in *Proc. of VLDB*, 2003.
- [24] Y. Zhu, E. Rundensteiner, and G. Heineman, "Dynamic plan migration for continuous queries over data streams," in *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, 2004.