Adaptive Optimization of Multi-Join Stream Queries in RETE-based Production System

Joseph Gomes
The George Washington University
Washington, DC, USA
jogomes@gwu.edu

Heyong-Ah Choi
The George Washington University
Washington, DC, USA
hchoi@gwu.edu

ABSTRACT
Data Stream Management Systems (DSMS) have recently received a lot of attention from the database research community. A DSMS handles a particular type of applications that involve multiple continuous data streams with inputs arriving at highly variable and unpredictable rates. Since the data rate fluctuates over time, an appropriate join tree for processing queries must be maintained adaptively in response to dynamic environments 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 streams and present the following results. We first show the problem is NP-hard when no restrictions are imposed on the join tree. We developed an exponential time dynamic programming algorithm OPTDP, that produces the optimal tree. We then present a heuristic algorithm XGreedy that can produce efficient join trees that are not restricted to left-deep trees. We implemented our algorithms into the adaptively re-optimizing stream query processor that we developed by extending JESS, which is a popular RETE-based production system written in java. We then compare XGreedy to OPTDP and common heuristics such as sorting by arrival rate, window size etc. We also compared XGreedy to FODP, which we developed in earlier work. Our extensive experimental results show that for most instances, both XGreedy and FODP perform close to OPTDP and significantly better than the common heuristics.

1. INTRODUCTION
We consider the problem of constructing an optimized join tree for processing continuous stream queries in a volatile environment. A stream join is just like relational joins in traditional database systems with the static relations replaced by continuous streams of structured data. Example applications include network traffic engineering, network monitoring, intrusion detection, financial monitoring, data mining, signal processors from various types of sensor networks etc. Continuous streams are unbounded by nature. Performing join queries on them is deemed to be impractical as one would need to compare every tuple in one infinite stream with every tuple 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 \( \alpha \) tuples are of interest. In either case the windows are continuously sliding, i.e., as new tuples enter the windows, old ones exit. In this paper we focus on count based sliding windows.

Let us consider an example to see what is involved in processing a sliding window based query in a streaming environment.

Example 1. Consider a three-way join \( R_1 \bowtie R_2 \bowtie R_3 \) where the join predicates are \( R_1.x = R_2.x \) and \( R_2.y = R_3.y \) and the window predicates specify a window size of 5 for all three streams. Consider a join tree in Figure 1 (a) and the contents of streams shown in (b) where the oldest data is on top of the list. The join results of the intermediate join node for \( R_1 \bowtie R_2 \) are shown in (c). If a new tuple \( (1) \) arrives at \( R_1 \) then the following actions will be performed.

1. \( (1) \) will be appended to \( R_1 \)’s window.
2. \( (1) \) will be compared against all the tuples in \( R_3 \)’s window. All joined tuples of the form \( (R_1.x, R_2.x, R_3.y) \), in this case \( (1, 1, 4) \), 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 \( (R_1.x, R_2.x, R_2.y, R_3.y) \), in this case \( (1, 1, 4, 4) \), will be propagated as part of the result.
3. \( (3) \) will be removed from \( R_1 \)’s window to maintain the window size. As a result of this, the matching tuple \( (3, 3, 6) \) will be removed from the intermediate join node \( R_1 \bowtie R_2 \).

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, suppose
stream $R_1$ has the highest arrival rate. Having a join tree with the order $(R_2 \bowtie R_3) \bowtie R_1$ might be preferred over the one shown in Figure 1 (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 big compared to that of $R_1 \bowtie R_2$ then the tree in Figure 1 (a) may still be the better choice.

Finding the optimal join tree is NP-Hard even for traditional databases[@2, @2, @2]. The addition of dynamic streams with varying arrival rates makes the problem even tougher to solve. The vast majority of the work dealing with this problem only considers linear join orders (i.e., left-deep binary tree structures) leaving out bushy trees (i.e., arbitrary binary tree structures) from the search space. In this paper we prove that finding optimal join trees for stream queries is NP-Hard. We show that by considering bushy trees we can get significant performance gains over left join trees. We develop a greedy algorithm, called XGreedy, and the optimal dynamic programming algorithm, OptDP. We revisit FODP[@2], a dynamic programming algorithm that only considers bushy trees from a single permutation of the streams. We also introduce a low cost join ratio approximation technique that combines traditional database techniques with dynamic adjustments from runtime statistics. We implemented XGreedy, OptDP and FODP, along with its variants into our Adaptively Re-optimizing Stream Management System (ARSMS) based on Jess.

1.1 Outline of paper
The rest of the paper is organized as follows: section 2 discusses the cost function and formalizes problem statement. Section 3 discusses related work. Section 4 presents a proof of NP-Hardness of the problem. Section 5 presents XGreedy, OptDP and FODP. Section 6 discusses different aspects of adaptively optimizing join trees, such as join ratio approximation, plan migration etc. and the runtime overheads associated to ARSMS. In sections 7 and 8 we respectively discuss how we set up our experiments and show their results. Finally we draw conclusion in section 9.

2. COST FUNCTION AND PROBLEM STATE-MENT
We are given $n$ streams $R_1, ..., R_n$ such that each $R_i$ has a set $A(i)$ of attributes. Let $a(i)$ and $w(i)$ denote the arrival rate and window size of stream $R_i$, respectively. Each of these streams can be represented by a leaf node in any join tree. As we assume count-based windows, each leaf node has the same arrival and output rates.

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.) When a new tuple $(q)$ arrives at $s$ from $l$ or $r$, it has to be matched with all the tuples in $r$ or $l$, respectively, using the join predicates that apply between $l$ and $r$. When a match is found the joined tuple is inserted to the list of partial results stored at $s$. The output rate $out(s)$ represents the number of tuples that are added to the partial results of $s$ and passed along to the parent node of $s$ in unit time. 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 $(q)$ expires at $l$ or $r$, all the tuples stored in $s$ that were created from $(q)$ have to be deleted constituting the deletion cost $dc(s)$.

We use $t(s)$ to denote the cost for comparing 1 tuple in $l$ with 1 tuple in $r$ at node $s$ and $j(s)$ to denote the join ratio(probability) at node $s$. Note that $t(s)$ should depend on the number of common join attributes between two nodes $l$ and $r$ and the complexity of matching those join attributes. Given the above notations, we calculate $ic(s), w(s), dc(s),$ and $out(s)$ as follows:

$$ic(s) = (out(l) \cdot w(r) + out(r) \cdot w(l)) \cdot t(s)$$

$$w(s) = w(l) \cdot w(r) \cdot j(s)$$

$$dc(s) = (out(l) \cdot w(s) + out(r) \cdot w(s)) \cdot t(s)$$

$$out(s) = (out(l) \cdot w(r) + out(r) \cdot w(l)) \cdot j(s)$$

We ignore the actual costs of accessing, inserting and deleting a tuple, since they should be negligible compared to matching costs. Although our matching costs looks more suitable to nested loop joins, it is also applicable to hash joins if each window and intermediate result has the same number of buckets. In such cases, the matching costs should only differ by a factor of $B$ where $B$ is the avg. number of hash buckets per table. Since we only need relative costs for comparing plans, our cost function can be used in such cases also.

Note that the final results of any query query go out of the system and are not stored as far as the deletion cost is concerned. Consequently, $dc(s) = 0$ if $s$ is the final join node. There will be exactly $n-1$ join nodes in any join tree with $n$ streams. (This due to the fact that any tree with $n$ leaf nodes has $n-1$ internal nodes.) Let $S^T = \{ s_i^T \mid 1 \leq i \leq n-1 \}$ be the set of these join nodes resulted from a join tree $T$. The estimated total cost incurred by a given join tree $T$ during a unit-time period is then defined as

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

We are now ready to state our problem.

Optimal Join Tree (OptJT) Problem: Given a set of
streams $R_1, \cdots, R_n$ 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 $cost(T)$ (i.e., maximize the throughput).

We consider only main-memory resident stream systems, i.e. we assume there is always enough memory to store the partial results and the stream’s window buffers. In addition to that, we consider eager-reevaluation and eager-expiration strategies as described in Example 1. In other words, query results are generated after each new arrival or expiration of a tuple rather than performing the query periodically on blocks of inputs. Moreover, once a result tuple is produced, it is streamed to the user immediately without storing.

3. RELATED WORK

DSMS has garnered a lot of attention from the research community in the past few years due to their vast applicability in a variety of stream processing applications. [?] provides a good survey of issues with data management under streaming environment. A number of different systems have been developed that use novel techniques in dealing with different aspects and issues of the continuous query processing: Cougar[7], TelegraphCQ[?], NiagaraCQ[?] and STREAM[?], [?, ?, ?, ?] are a few of them. [?] provides the user with a visual interface to design query plans, TelegraphCQ considers the adaptive aspect of continuous query processing. NiagaraCQ addresses scalability issues in processing multiple queries and STREAM focuses on issues such as memory management[?], operator scheduling[?], caching[?], filter ordering[?], adaptive plan generation, maintaining statistics[?] etc.

A rate-based query optimization model was proposed for continuous queries in[?]. Kang et al. propose asymmetric joins to enhance performance and window size manipulation to cope with memory constraints in [?]. Relevant work on joining algorithms include XJoin[?], and MJoin[?]. XJoin is similar to the RETE match algorithm[?] used in most production systems or rule engines in that it uses left deep trees and stores intermediate join results to avoid re-computation. MJoin differs from XJoin in 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 m-way join. MJoin is a close counterpart of TREAT, another matching algorithm for production systems proposed by Miranker[?]. The benefits of both TREAT and MJoin stem from the low cost of deleting a tuple due to not having to maintain partial results.

Most adaptive systems in the literature consider MJoin and XJoin plans with left-deep trees. The experiments in eddies[?] and stream[?] suggest that the benefits of storing partial results outweigh the costs of maintaining them in most cases. In this paper we focus on join trees that store partial results and do not have any restriction on its topology(bushy).

4. 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 2 has window size $2^0$ and will be called as such. Each input stream has a single common attribute$s$, 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 4.1.** Given 3 streams $R_1, R_2, R_3$ with window sizes $w_1, w_2, w_3$ respectively, such that (i) $w_3 \geq w_1, w_2$, (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 $(R_1 \Join R_2) \Join R_3$ (or equivalently $R_3 \Join (R_1 \Join R_2)$).

**Proof.** Let $w_1 + w_2 + w_3 = b$. Using the join tree shown in Figure 2, the insertion and deletion costs for the first join node are given by $ic(2^{w_1 + w_2}) = 2^{w_1} + 2^{w_2}$ and $dc(2^{w_1 + w_2}) = 2^{w_1 + w_2 + 1}$. The costs for the second join node are $ic(2^b) = (2^{w_1} + 2^{w_2}) 2^{w_3} + 2^{w_1 + w_2} = 2^{w_1 + w_3} + 2^{w_2 + w_3} + 2^{w_1 + w_2}$ and $dc(2^b) = (2^{w_1} + 2^{w_2} + 1)2^b$. 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, $C_0 = 2^{w_1} + 2^{w_2} + 2^{w_1 + w_2} + (2^{w_1} + 2^{w_2} + 1)2^b$. It is clear to see that $C_0$ is minimized if and only if $w_3 \geq w_1, w_2$. 

![Figure 2: Join tree for 3 streams](image)

4.1 Transformation from 3Partition to OptJT

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

**Definition 4.1.** **3PARTITION** Problem:

We are given a finite set $A$ of $3m$ elements, a bound $b \in Z^+$, and a positive integer $s(a) \in A$, such that each $s(a)$ satisfies $b/4 < s(a) < b/2$ where $\sum_{s(a) \in A} s(a) = mb$. The following decision problem is called 3PARTITION: Can $A$ be partitioned into $m$ disjoint sets $S_1, S_2, \ldots, S_m$ 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 $A = \{a_1, a_2, \ldots, a_{3m}\}$ be a set of $3m$ elements. We assume $m = 2^k$ for some $k \in Z^+$. (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/k < a_i < b/2. Hence, b ≥ 5. For the rest of section, we will assume that m is a power of 2 and b ≥ 5.

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

\[L(T) = A_0 \cup L_0 \cup L_1 \cup \cdots \cup L_{k-1},\]

where (i) A_0 = \{2^{m_i} \mid 1 \leq i \leq 3m\} where the window size and the arrival rate of each node 2^{m_i} are 2^{m_i} and one, respectively, and (ii) for 0 ≤ i ≤ k - 1, there are 2^{k-i} nodes in L_i with window size 2^{m_{i+1}} and arrival rate 2^{d_{i+1}}. 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^{i-1} = m + (2m - 1) leaf nodes. We will call this transformed OptJT instance as RInstance.

**Theorem 4.1.** There exists a solution for the 3PARTITION problem with A if and only if the optimal solution of the OptJT 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 of OptJT has the above property. Suppose 3PARTITION has a solution \( A = S_1 \cup \cdots \cup S_m \) such that \( S_i = \{a_{3i-2}, a_{3i-1}, a_{3i}\} \) for \( 1 \leq i \leq m = 2^k \). Then we 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 · 2^i nodes from \( A_0 \) are included in \( T_i \), where \( T_i \) (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 3 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}m_{i+1}^i} and arrival rate (i.e., their output rate) 2^{2^{i-1}m_{i+1}^i}. As a basis case, \( T_1 \) is given in Figure 4. The following two lemmas will be used to complete the proof of the theorem.

![Figure 3: T_i for 2 ≤ i ≤ k, where h = 2(m + 1) and l, l' ∈ L_{i-1.}](image)

**Lemma 4.2.** For each i, 1 ≤ i ≤ k, we have the following bounds:

\[ QCost(T_i) < 2^{(3m+1)k^i-1} + 2^{2(m+1)k^i-1} b + 2 \]

\[ ic(2^{k+1}) < 2^{(3m+2)k^i-1} b + 2 \]

\[ dc(2^{k+1}) < 2^{(4m+3)k^i-1} b + 2^{(4m+4)k^i-1} b \]

**Proof.** We will prove the lemma using induction on i. For \( i = 1 \), the tree \( T_1 \) shown in Figure 4 is clearly an optimal join tree due to Lemma 4.1. The insertion cost at node \( 2^{i+1} \) is maximum when \( (a_1, a_2, a_3) = (1, 2, 3) \). Of course \( a_1, a_2 \) can be interchanged without changing any cost. Additionally, the maximum possible size of node \( 2^{i+1} \) is \( 2^b \) and occurs when \( (a_1, a_2, a_3) = (\frac{b}{3}, \frac{b}{3}, \frac{b}{3}) \). Consequently, we get the following inequalities.

\[ ic(2^{i+1}) < 2^{i+1} + 2 = 2^{i+1} \]

\[ dc(2^{i+1}) < 2^{i+1} + 2 = 2^{i+1} \]

\[ i(2^{i+1}) < 2^{i+1} + 2^{i+1} = 2^{i+1} \]

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.

\[ SCost(2^{(m+1)b}) < 2^{m+b} + 2^{2m+b+1} \]

As \( QC_{Cost} \) for node \( 2^{(m+1)b} \) is the sum of the \( SCost \) of its two children subtrees, we have

\[ QC_{Cost}(2^{(m+1)b}) < 2^{3m+b+1} + 2^{3m+b+1} \]

\[ ic(2^{(m+1)b}) < 2^{3m+b+1} + 2^{3m+b+1} \]

\[ dc(2^{(m+1)b}) < 2^{3m+b+1} + 2^{3m+b+1} \]

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 $de$.

$$SCost(2^{h-k-1}b) < 2(3m+3)h^{k-1}b + 2(3m+4)h^{k-2}b$$

$$+ 2(3m+2)h^{k-2}b + 2(3m+1)h^{k-3}b + 2(3m+3)h^{k-4}b$$

$$< 2(3m+3)h^{k-2}b + 1 + 2(3m+4)h^{k-2}b$$

From Figure 3, we derive the following for level $k$.

$$ic(2^{(m+1)h^{k-1}b}) < 2(2^{m+2}h^{k-2}b + 2^{(m+1)h^{k-1}b})$$

$$+ 2^{(m+1)h^{k-2}b} + 2^{(m+2)h^{k-3}b}$$

$$< 2^{(m+1)h^{k-2}b} + 2^{(m+2)h^{k-3}b}$$

$$dc(2^{(m+1)h^{k-1}b}) < 2(3m+3)h^{k-2}b + 2^{(m+1)h^{k-1}b}$$

$$+ 2^{(m+2)h^{k-3}b} + 2^{(m+3)h^{k-4}b}$$

$$< 2^{(m+1)h^{k-2}b} + 2^{(m+2)h^{k-3}b}$$

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

$$SCost(2^{(m+1)h^{k-1}b}) < 2(3m+1)h^{k-1}b + 2^{(m+3)h^{k-1}b}$$

$$+ 2^{(m+1)h^{k-1}b} + 2^{(m+2)h^{k-3}b} + 2^{(m+3)h^{k-2}b}$$

$$< 2^{(m+1)h^{k-1}b} + 1 + 2^{(m+2)h^{k-3}b}$$

$$QCost(2^{h}b) = 2(SCost(2^{(m+1)h^{k-1}b}))$$

$$< 2(3m+1)h^{k-1}b + 1 + 2^{(m+3)h^{k-1}b}$$

$$ic(2^{h}b) < 2^{(2m+1)h^{k-1}b} + 2^{(2m+3)h^{k-1}b}$$

$$< 2^{(2m+1)h^{k-1}b} + 2^{(m+2)h^{k-1}b}$$

$$dc(2^{h}b) < 2^{(2m+1)h^{k-1}b} + 2^{(m+3)h^{k-1}b}$$

$$< 2^{(2m+1)h^{k-1}b} + 2^{(m+2)h^{k-1}b}$$

Therefore, the induction holds, and this completes the proof of Lemma 4.2.

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

**Lemma 4.3.** Let $R_s$ be a new stream with window size $2^{2^{m+1}h^{k-1}b}$ and arrival rate $2^{2^{m+1}h^{k-1}b}$. Then, $R_s$ is the last node joined in any optimal join tree (i.e., a join tree with the minimum $SCost$) for $RInstance \cup \{R_s\}$.

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

$$out_T'(2^{2^{mh^{k-1}b+x}}) > 2^{2^{mh^{k-1}b+x}}$$

$$SCost_T'(2^{2^{mh^{k-1}b}}) > dc(2^{2^{mh^{k-1}b}})$$

$$out_T'(2^{2^{mh^{k-1}b+x}}) > 2^{2^{mh^{k-1}b+x}}$$

This completes the proof of Lemma 4.3.

Using Lemmas 4.2 and 4.3, 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 $T'_r$, and two nodes $l$, $l'$ from $L_{k-1}$ must be included in $T'$. Two cases are considered. In Case 1, both $l$ and $l'$ are in one of the subtrees, say $T'$. (See Figure 5.) In Case 2, $l$ belongs to $T'_l$ and $l'$ belongs to $T'_r$. The window size of the root of subtrees in each case is defined as $2^{(m+1)h^{k-1}b+x}$ and $2^{(m+1)h^{k-1}b+e}$ (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 3).

77. The subtree that has the two biggest nodes may or may not have joined to the big nodes. The presence or absence of these additional nodes are $-x$ in the sizes of the roots of the subtrees, respectively. In the second case, it has the biggest nodes on each side of the tree as shown in figure

![Figure 5: T' in Case 2 where h = 2(m+1) and l, l' ∈ L_{k-1.}](image)

**Case 1:** Note that both sub-trees are non-empty implying $0 \leq x < 2^{2^{mh^{k-1}b-1}}$, and we observe the following:

$$QCost_T'(2^{2^{mh^{k-1}b}}) > dc(2^{2^{mh^{k-1}b+x}})$$

$$= 2^{2^{mh^{k-1}b+x}}$$

$$QCost_T'(2^{2^{mh^{k-1}b}}) > QCost_T'(2^{2^{mh^{k-1}b}})$$

$$out_T'(2^{2^{mh^{k-1}b+x}}) > 2^{2^{mh^{k-1}b+x}}$$

$$SCost_T'(2^{2^{mh^{k-1}b}}) > dc_T'(2^{2^{mh^{k-1}b}})$$

$$= 2^{2^{mh^{k-1}b+x}}$$

$$SCost_T'(2^{2^{mh^{k-1}b}}) > SCost_T'(2^{2^{mh^{k-1}b}})$$

**Case 2:** In this case, by virtue of Lemma 4.3, we don’t need to consider any other form for the left or right subtree,
where the big nodes are not the last nodes being joined; hence, we have $0 \leq x \leq 2^{b-1}$. Depending on the value of $x$, we consider three sub-cases.

**Case 2.1:** $x = 2^{b-1}$

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

$$QCost_{T'}(2^b) > dC_{T'}(2^{(m+1)b^*-1}b+x)$$

$$= dC_{T'}(2^{(m+1)b^*-1}b)$$

$$> 2^{(3m+3)b^*-1}b$$

$$= QCost_{T}(2^b)$$

$$out_{T'}(2^{(m+1)b^*-1}b+x) > 2^{2mb^*-b+2m+1}b^* - b$$

$$= 2^{4mb^*-1}b$$

$$> SCost_{T'}(2^b)$$

**Case 2.2:** $1 \leq x < 2^{b-1}$

This gives us the following:

$$QCost_{T'}(2^b) > dC_{T'}(2^{(m+1)b^*-1}b+x)$$

$$+ dC_{T'}(2^{(m+1)b^*-1}b-x)$$

$$> 2^{2mb^*-b+2m+1}b^* - b$$

$$+ 2^{4mb^*-1}b$$

$$> 2^{(3m+3)b^*-1}b+x + 2^{(3m+3)b^*-1}b-x$$

$$> QCost_{T'}(2^b)$$

**Case 2.3:** $x = 0$

In this case, $T'$ looks $T^*$. However subtrees rooted at the two nodes of size $2^{b-1}b$ in $T'$ may not conform to $T^*$. So, assume that $k$ is different. We will then use induction on $k$ to prove $QCost_{T'}(2^b) > QCost_{T}(2^b)$. When $k = 1$, it is trivial. So, assume that for any $m = 2^k$, for $1 \leq i \leq k-1$, $QCost_{T'}(2^b) > QCost_{T'}(2^b)$ and $SCost_{T'}(2^b) > SCost_{T'}(2^b)$. Consider $i = k$. Note that $T'$ and $T^*$ both have the same insertion costs at the root node $2^b$, 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 $(2^{(m+1)b^*-1}b)$ is equal to that in $T^*$. As a result, the sum of $dc$ is 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 $2^{b-1}b$ both contain exactly $2^{b-1}b$ leaf nodes with window size equal to $2^{m+1}b$, for $0 \leq i \leq k-1$, and original nodes constituting a total window size of $2^{b-1}$, as otherwise the two subtrees would not have the same root size. This means each subtree represents an $Rm$ instance with $m = 2^b$, 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 $m = 2^b$ which completes the induction.

This completes the proof of the theorem.

Theorem 4.1 proves the validity of our transformation from 3Partition to OptJT, which establish the following NP-hardness result.

**Theorem 4.2.** The OptJT problem is NP-hard even if the join ratio at each join node is one and all input streams have only one common attribute.


The difficulty of finding optimal join trees has driven researchers towards limiting their searches to left deep join trees. Usually some low cost heuristic is used to come up with the linear join tree. However variable arrival rates in streaming environments often render the linear trees to be inefficient. In previous work [7] we extended the search space to include all possible join trees given a non-commutative but associative order of streams and developed a dynamic programming algorithm FODP that finds an optimal join tree for the given order. In this paper we further extend the search space to include all possible join trees. Here we present a greedy algorithm called XGreedy that finds efficient join trees under stable conditions, which are not restricted to left deep trees. We also briefly describe our FODP algorithm and its variants. Then we present the exponential time optimal dynamic programming algorithm OptDP.

### 5.1 The XGreedy Algorithm

Let each stream $R_i$ be represented by a leaf node $m_i$, each with its output rate, window size and attribute list. Let $candidate$nodes be the set of currently available nodes that can be joined. Initially it contains all the leaf nodes. During each iteration of the loop, all possible pairs of nodes from candidate nodes are considered. For each pair of nodes their conditional join probability is computed based on their common attributes. Insertion and deletion costs are computed and added to get the incremental cost. The pair of nodes that has the minimum incremental cost is chosen and replaced by their join node.

Figure 6 shows the pseudocode for XGreedy. In the pseudocode the getJoinRatio function measures the join ratio using the method described in section 6.1. Function getCost measures the insertion and deletion costs and returns their sum. createAndConnectJoin creates a new join node, updates its result size(window), output rate and attribute list, and connects it to its left and right children.

### 5.2 Fixed Order Optimal Algorithm

Given an ordered set of streams, the FODP only looks at the trees that do not violate the given order, i.e. out of the $n!$ permutations it considers only a single permutation of the given order. FODP finds an optimal join tree for a single permutation in $O(n^3)$ time. In [7] we showed that FODP-trees perform significantly better than the given left deep trees.
candidateNodes = [n1, n2, n3, ..., nn];
numnodes = n;
for ( p = 1 ; p <= n - 1 ; p=p+1 )
// calculate the cost for all join pairs
// and find the minimum
Node leftCdt, rightCdt;
Mincost = 8;
for ( i = 1; i < numnodes ; i++)
for ( j = i + 1 ; j < numnodes; j++)
Node left = candidateNodes[i];
Node right = candidateNodes[j];
joinratio = getJoinRatio(left, right);
cost = getCost(left, right, joinratio);
if (cost < mincost)
    mincost = cost;
leftCdt = left;
rightCdt = right;
Node join = createAndConnectJoin(leftCdt, rightCdt);
candidateNodes = candidateNodes + join - leftCdt - rightCdt;
numnodes = numnodes - 1;

Figure 6: pseudo code for XGreedy

Notice that given a set of n streams, the size of the output and the output rate of the n-way join should remain same regardless of the join tree being used. Let us use $T_{i,j}$ to denote an optimal join tree for a given fixed permutation $R_1 \bowtie R_{k+1} \bowtie \cdots \bowtie R_n$ and let $C(T_{i,j})$ be its cost. Also, let $ic_{i,j;k}$ and $dc_{i,j;k}$ denote the insertion and deletion costs associated to a join node that has streams $R_{i} \cdots R_{k}$ in its left subtree and $R_{k+1} \cdots R_{n}$ in the right subtree. Then the following recurrence relation holds:

$$C(T_{i,j}) = \min_{i \leq k \leq j} \{ C(T_{i,k}) + C(T_{k+1,j}) + ic_{i,j;k} + dc_{i,j;k} \}$$

### 6.1 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. 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 [7, 8]. 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 with the ratio of the smaller range to the bigger range. Using the containment assumption we calculate the join ratio and expected result size of the two streams $R_i$ and $R_j$ as follows:

$$joinratio = \frac{1}{\prod_{X \in \text{attr}(R_i) \cap \text{attr}(R_j)} \max(dv_{R_i,X}, dv_{R_j,X})}$$

$$\text{resultsize} = \frac{w(i) \cdot w(j)}{joinratio}$$

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.

\[
dv_{\text{join}, X} = \min(\min(dv_{\text{left}, X}, dv_{\text{right}, X}), \text{resultSize})
\]

\[
\max_{\text{join}, X} = \max(\min(dv_{\text{left}, X}, \max_{\text{right}, X})
\]

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

\[
dv_{\text{join}, Y} = \min(dv_{\text{child}, Y}, \text{resultSize})
\]

\[
\max_{\text{join}, Y} = \max(dv_{\text{child}, Y})
\]

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 straightforward manner. For the given experiments \(x\) and \(y\) were set to 2 and 5 seconds respectively.

### 6.2 Adjusting join ratio and result size according to input characteristics

The standard assumptions such as uniform distribution, independence, containment of values etc. are not always realistic for real world data. However they help us get a crude estimation of the real join ratio and result size. We adjust these crude estimations further using a factor called the adjustment factor that is calculated using the runtime intermediate result sizes of the current join tree at work. Let the current join tree at work be denoted by \(T\). Let \(\{j_1, \ldots, j_n\}\) be the set of join nodes in \(T\) and \(\{r_1, \ldots, r_m\}\) be the actual result sizes of these join nodes in \(T\). Let \(\{\text{dv}_{j_1}, \ldots, \text{dv}_{j_n}\}\) be the estimated result sizes at these join nodes. Then the adjustment factor is measured by \(\frac{\text{dv}_{\text{join}}}{\sum_{j \in \text{join}} \text{dv}_{j}}\). The estimated result size and join ratio for the candidate intermediate nodes in our algorithms are multiplied by this adjustment factor to get better estimations.

### 6.3 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[2]. 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 streams according to the new join tree to populate its intermediate nodes. We take a different approach which is similar to the moving state strategy in [2]. Our objective is to try to reuse as much intermediate result from the old tree to avoid re-computation.

**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. For our purposes two join nodes are exactly same if they have the same left and right children. For example in the two trees from figures 7 (a) and (b), the nodes \(R_4 \gg R_5\) 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 re-generate its results.

**Reuse existing results:** If the previous case is not satisfied, we check if the current node is semantically the 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 figures 7 (a) and (b), \(R_1 \gg R_2 \gg R_3\) and \(R_2 \gg R_3 \gg R_1\) are semantically same. If such a node is found we can reuse its results, however 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.

### 6.4 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 (2 seconds). 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 2 seconds.

**Construct 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 6.3. 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 5% 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.

### 7. EXPERIMENTAL SETTING

We implemented our algorithms in JESS, which is a popular rule engine written in Java. We tested our system using synthetic datasets and a real sensor dataset from Intel Research, Berkeley Lab.

**Synthetic tests:** For the synthetic datasets our experiments used n-way joins of the form \(R_1(A) \bowtie A R_2(A) \bowtie A \cdots \bowtie A R_n(A)\) using integer attributes. We used a synthetic data generator to produce multiple streams of data according to specified data characteristics and arrival rates. In each iteration of a for loop a stream \(R_i\) is chosen with probability
8. EXPERIMENTAL RESULTS

8.1 Varying join ratio

Five streams with relative arrival rates \( \{5, 1, 5, 1, 1\}\) each with window size \(=1000\) and equal \(dv\) were used for each test case. Join ratios were varied among the test cases by changing the \(dv\). Figure 8 shows that as the join ratios increase (\(dv\) decreases) the performance of the optimized trees from all the algorithms get better compared to the un-optimized tree. The performance increase varies between 10-55%. \textit{FODP} performs worst out of the four algorithms. For all the test-cases except for when \(dv = 400\) and 600, \textit{OptDP, SortAR + FODP}, and \textit{XGreedy} chose the same trees. For \(dv = 400\) and 600 \textit{XGreedy} chose a different tree and performs marginally worse than \textit{OptDP} and \textit{SortAR + FODP}.

8.2 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 9 shows the results. Performance enhancements were observed across all window sizes, going from 5% to about 50% as windows get larger. Again \textit{FODP} lags behind the other 3 due to its order violating restriction. The curves for the other 3 algorithms mostly coincide as a result of choosing the same tree barring when \textit{XGreedy} chose different trees for \(dv = 400\) and 600.

8.3 Varying Number of Streams

For this scenario all window sizes and \(dvs\) were set to \(1000\). For each case two very fast streams were used with relative rate of 10, with one at the beginning of the order and one at the middle. All other relative rates were set to 1. E.g. when six streams are used the relative arrival rates are \(\{10, 1, 1, 10, 1, 1\}\), for seven streams they are \(\{10, 1, 1, 10, 1, 1, 1\}\). Figure 10 shows that performance increase goes higher with larger stream counts for all algorithms. \textit{FODP} performs much better for these test-cases than the two previous categories since the two large arrival rates fall in two halves of the stream order, which helps it to generate a more balanced tree. \textit{SortAR + FODP} and \textit{XGreedy} perform comparably throughout all the test cases. \textit{OptDP}'s performance starts to degrade slightly for higher stream counts (\(\geq 7\)) due to its exponential order.
8.4 Varying arrival rates
We examined ten different test cases $T_1 - T_{10}$ with varied arrival rates shown in table 1. The window sizes 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 11 shows the results for these tests. As expected when faster streams are positioned in the beginning the un-optimized tree performs poorly and performance increase is higher for all algorithms. For $T_i$, no optimization is necessary as all the algorithms use the same tree, which is the un-optimized join tree. We noticed a 0.9% performance overhead for using the adaptive optimization algorithms. $FODP$ uses different trees in $T_5 - T_7$ from the others, that perform worse than the optimal trees. Apart from that all the algorithms produced the optimal trees.

<table>
<thead>
<tr>
<th>Relative Arrival Rates</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$T_6$</th>
<th>$T_7$</th>
<th>$T_8$</th>
<th>$T_9$</th>
<th>$T_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>$R_2$</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>$R_3$</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$R_4$</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$R_5$</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Sample Tests with Varied Arrival Rates

8.5 Performance comparison under varied stream characteristics
We extensively tested the join trees generated by the algorithms that we discussed in this paper under different stream characteristics. Figure 12 shows the results from eight of these experiments, 4 of which involves 5-way joins of the form $R_1(A) \bowtie_4 R_2(A) \bowtie_4 R_3(A) \bowtie_4 R_4(A) \bowtie_4 R_5(A)$ and 4 involves 8-way joins $R_1(A) \bowtie_4 R_2(A) \bowtie_4 R_3(A) \bowtie_4 \cdots R_6(A)$. Arrival rates, window sizes and DV counts were varied in our experiments. The relative arrival rates for the 8 sample test cases are shown in table 3. Tables 2 and 4 respectively show count based window sizes and distinct value counts for the streams. As usual the results in figure 12 show the maximum number of tuples the system can handle for each sample test. Our adaptively re-optimizing system was executed using all the algorithms that we discussed earlier including the optimal dynamic programming algorithm for all the test scenarios. Thus the result shown in figure 12 accounts for all sorts of overhead as well. We noticed that all the algorithms generated join trees that performed significantly better than the un-optimized join tree. $OptDP$'s trees perform the best as expected. Its join trees can handle 68% more load than the un-optimized trees on average. However trees from the rest of the algorithms are quite close to those of $OptDP$ in terms of performance. XGreedy-trees usually performed well and only around 3% worse than OptDP-trees on average. Trees from SortAR+FODP, SortWIN+FODP, SortJR+FODP and FODP performed around 4%, 4.5%, 6% and 8% worse than OptDP, respectively.

<table>
<thead>
<tr>
<th>Sample Test</th>
<th>Window Size (in hundreds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>10 12 8 10 16 - - -</td>
</tr>
<tr>
<td>$T_2$</td>
<td>14 12 8 16 10 - - -</td>
</tr>
<tr>
<td>$T_3$</td>
<td>12 8 14 16 16 - - -</td>
</tr>
<tr>
<td>$T_4$</td>
<td>10 6 16 8 14 - - -</td>
</tr>
<tr>
<td>$T_5$</td>
<td>10 12 15 4 6 16 8 14</td>
</tr>
<tr>
<td>$T_6$</td>
<td>14 12 16 8 14 8 6 14</td>
</tr>
<tr>
<td>$T_7$</td>
<td>18 12 6 8 14 18 6 4</td>
</tr>
<tr>
<td>$T_8$</td>
<td>8 12 6 8 14 18 6 14</td>
</tr>
</tbody>
</table>

Table 2: Window Sizes for Sample Tests

<table>
<thead>
<tr>
<th>Sample Test</th>
<th>Relative Arrival Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>1 2 10 2 1 - - -</td>
</tr>
<tr>
<td>$T_2$</td>
<td>10 10 2 2 1 - - -</td>
</tr>
<tr>
<td>$T_3$</td>
<td>5 10 5 1 2 - - -</td>
</tr>
<tr>
<td>$T_4$</td>
<td>1 1 1 1 1 - - -</td>
</tr>
<tr>
<td>$T_5$</td>
<td>1 5 1 7 1 8 2 10</td>
</tr>
<tr>
<td>$T_6$</td>
<td>1 5 5 2 10 1 1 5</td>
</tr>
<tr>
<td>$T_7$</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>$T_8$</td>
<td>2 1 8 7 10 1 1 5</td>
</tr>
</tbody>
</table>

Table 3: Arrival Rates for Sample Tests

8.6 Adaptivity testing with sensor data
### Table 4: DV counts for Sample Tests

<table>
<thead>
<tr>
<th>Sample Test</th>
<th>DV count (in hundreds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>6 10 8 8 12</td>
</tr>
<tr>
<td>$T_2$</td>
<td>8 10 6 7 5</td>
</tr>
<tr>
<td>$T_3$</td>
<td>12 8 10 6 6</td>
</tr>
<tr>
<td>$T_4$</td>
<td>8 2 20 3 12</td>
</tr>
<tr>
<td>$T_5$</td>
<td>10 12 6 8 4</td>
</tr>
<tr>
<td>$T_6$</td>
<td>10 12 6 8 14</td>
</tr>
<tr>
<td>$T_7$</td>
<td>10 12 6 8 16</td>
</tr>
<tr>
<td>$T_8$</td>
<td>10 12 6 8 16</td>
</tr>
</tbody>
</table>

### Figure 12: Performance comparison of stream join trees

In this experiment we consider adaptivity to dynamic variability in stream characteristics. For adaptivity testing we use the real sensor data set as discussed in section 7. To evaluate the various algorithms we ran queries of the following semantic form:

Select * from Humid H, Volt V, Temp T, Light L  
Where (H.humid > lowH AND H.humid < highH)[1000]  
AND (V.volt > lowV AND V.volt < highV)[1000]  
AND (T.temp > lowT AND T.temp < highT)[1000]  
AND (L.light > lowL AND L.light < highL) [1000]  
AND (H.id = V.id = T.id = L.id)  
AND (H.ts = V.ts = T.ts = L.ts)

We ran 25 such random queries and noticed an improvement of 11.6% for OptDP, FODP and XGreedy, and an 11.3% improvement for SortJR+FODP. We did not run the other two versions of FODP since both the arrival rates and window sizes are equal among the streams.

As we have seen from previous experiments drastic improvements can be expected when the number of streams is high and the input characteristics significantly differ among the streams. In this experiment for the first 500,000 tuples of first three streams the departure rates out of the selection operators were similar and varied between 1 and 2 relative to each other. For the last stream, the selectivity varied between 1/20 and 1 relative to the least selective of the other operators. With only 4 streams and stable arrival rates of 10, 10, 10, 1, with all window sizes and dv counts set to 1000 even the optimal tree gets only 12% improvement. In light of that we think the performance improvements in this experiment are significant.

### 9. CONCLUSION

We have shown that finding optimal join trees for continuous stream queries is NP-Hard. We have presented and analyzed XGreedy that finds efficient join trees for such queries. We introduced a low cost join ratio computing strategy that combines traditional techniques and dynamic re-adjustment. Our experiments show that the join trees produced by XGreedy, FODP and its variants perform almost as well as optimal join trees produced by OptDP under varying stream characteristics. Our experiments using both synthetic data and real data have also shown that our algorithms and methods used in ARMS for adaptive re-optimization is inexpensive, and achieves near optimal performance improvements. In future we would like to consider the tradeoff between adaptivity and runtime overhead. A potential avenue in this regard would be to find a way to measure and quantify the variability of stream characteristics and adjust re-optimizing interval accordingly.

**Acknowledgement** Source code for the copyrighted software Jess was provided by Sandia Corporation.

Note that in the above query a window predicate Fof 1000 tuples takes effect on the result of the corresponding selection. The joins are performed on the last 1000 tuples in each stream that pass the selection predicates using the the attributes $ts$ and $id$. For each query the relative arrival rates of all four streams were set to 1. During each iteration a tuple was added to each stream with probability 1. For each experiment we ran 500,000 such iterations. Note that the arrival rates used by the join optimization algorithms are the departure rates from the selection operators. For each attribute $A$, the parameters $lowA$ and $highA$ were respectively chosen randomly from the bottom 35% and top 35% of the domain values.