Cost-based Solution for Optimizing Multi-join Queries over Distributed Streaming Sensor Data

Joseph Gomes Hyeong-Ah Choi
Department of Computer Science
The George Washington University
Washington, DC, USA
Email: {joegomes,hchoi}@gwu.edu

Abstract

Sensors are envisioned to be at the center of distributed collaborative computing services involving time-critical decision support. Sensors are small devices with limited communication and computational capabilities that collect data on their neighboring physical world and send the data periodically to server machines. Sensors form a collaborative network with these servers, where the sensors gather information and the servers perform various operations (e.g. filter, aggregate, join etc) on the information streams in real-time according to predefined queries or rules. Sensor data streams are continuous, un-ending and have highly volatile characteristics. As a result, traditional database systems are inappropriate for handling queries for sensor streams, and several stream data management systems have been proposed in the literature. In this paper we focus on a special type of query, namely join queries, which is the most expensive query operator. Here, we address the problem of finding an optimal join tree that maximizes throughput for sliding window based multi-join queries over continuous sensor data streams. We present a polynomial time algorithm Fodp and three variants of Fodp. Our experiments in ARES¹ show that for almost all instances, trees from Fodp and its variants perform close to the optimal trees from our exponential time algorithm OptDP [15], and significantly better than existing XJoin based heuristic algorithms.

1 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. We are on our way to creating an environment where computers would create a collaborative computing environment with thousands of sensors and make automated decisions 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. However due to the dynamic and continuous nature of these sensor streams, traditional database management systems (DBMS) are not well suited for them. Especially performing join queries 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,

¹ARES (Adaptively Re-optimizing Engine for Stream queries) is a continuous query processor that adaptively re-optimizes the query plan according to the current stream characteristics [15].

as in Figure 1, that 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.

Figure 1. Sensor stream processing hierarchy

1.1 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 $R_1 \bowtie R_2$.

Figure 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 big 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 read-

ings 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.

2 Rete Based Rule Engines

In this section, we try to draw a connection between continuous query processors and rule engines to justify our choice of Jess for implementing ARES.

A production system or a rule engine is defined by a set of *productions* or *rules* and a set of *facts* or *assertions*. The set of assertions is called the *working memory* (WM), and each assertion is called a *working memory element* (WME). A rule is specified by a left-hand side (LHS) and a right-hand side (RHS). The LHS is just a conjunction of conditions and the RHS specifies the set of actions that has to be performed when LHS of the rule is matched. Conditions on the LHS may contain variables and are mostly used for *inter-condition tests* or *joins*. A rule is said to match if there are WMEs that satisfy all its conditions, with all variables in the conditions bound consistently. It is the job of the match algorithm to determine which productions have all their conditions satisfied.

2.0.1 Rete match algorithm

Rete [13] is a fast matching algorithm based on forward chaining that transforms the left-hand sides of rules to a data-flow network. There are six types of nodes in Rete networks:

1. *root* node: This is the root of the whole rete network.
2. *const* nodes: These one-input nodes test single relation conditions (selections) that are also called

Figure 3. Example alpha network for Rete with conditions C_1 , through C_6

intra-condition tests.

3. α -*memory* nodes: These one-input nodes keep the temporary results of single-relation selection conditions.
4. *and* nodes: An *and* node is a two-input node that joins the tokens from left and right input nodes with a join condition, testing for consistent variable bindings between the two inputs.
5. β -*memory* nodes: These nodes hold the temporary result of joins from *and* nodes.
6. *p* nodes: These are the terminal β -*memory* nodes that hold the final result sets for a single rule.

The one-input nodes form the α -network and two-input nodes form the *pattern*-network. When an assertion adds or removes a WME, a token is created and passed on to the network through the root. The token first goes through the α -network, and if it passes all the intra-condition tests it is stored in the appropriate alpha-memories. Copies of the tokens are passed down to successive join (two-input) nodes. At the join nodes the inter-condition tests are performed using join variable bindings. The matched results are stored in β -*memories* as complex tokens, and copies of the complex tokens are passed down to further successors. Rules are activated when tokens reach their terminal nodes and the corresponding actions are taken. Some of these actions could assert new facts that are fed back into the network through the root. The resemblance between a Rete based rule engine and Stream query processors, specially ones using XJoin [21], is quite evident. Each rule is like a query and its alpha and beta network make up the query plan. Our choice of using Jess, a popular Rete based rule engine, for implementing our system, was mainly motivated by the similarities between Data Stream Management Systems and Production systems.

Figure 4. Example pattern network for Rete with production rules P_1 , P_2 and P_3

3 Related 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, maintaining statistics[10] etc.

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. [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. [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.

4 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 in-

cur 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:

$$\begin{aligned} ic_s &= (out_l \cdot w_r + out_r \cdot w_l) \cdot t_s \\ w_s &= w_l \cdot w_r \cdot j_s \\ dc_s &= (out_l \cdot w_s + out_r \cdot w_s) \cdot t_s \\ out_s &= (out_l \cdot w_r + out_r \cdot w_l) \cdot j_s \end{aligned}$$

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 . The estimated total cost incurred by a given join tree T during a unit-time period is then defined as

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

Our goal is to find a join tree T that minimizes the estimated unit-time cost $cost(T)$ (i.e., maximizes throughput). We have proven that this problem is NP-hard. The proof can be found in [15].

5 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 prominent ones are 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*. 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 then sorting by window size will not produce the desired effect. Similar reasonings also hold for the other two heuristics. Next we will briefly go over the OptDP and XGreedyJoin [15] algorithms, that we developed in earlier work. Then we will present the Fodp algorithm.

5.1 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 $A = \{p \mid p \subset S \text{ \& } p \neq \Phi\}$, 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. Let $p' = S - p$. Then the following recurrence relation holds:

$$C(T_S) = \min_{p \in A} \{C(T_p) + C(T_{p'}) + ic_{p,p'} + dc_{p,p'}\}$$

for $|S| > 1$, and $C(T_S) = 0$, for $|S| = 1$.

Theorem 5.1. *The execution time of OptDP is in the order of $(3^n - 2^{n+1} + 1)/2$.*

Proof. The proof can be found in [15]. $\square$

5.2 The XGreedyJoin Algorithm

XGreedyJoin is a greedy algorithm. 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. 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. During each iteration of the loop, all possible pairs of nodes from $cdts$ are considered to be joined. For each pair of these nodes their conditional join probability is computed based on their common attributes using runtime distinct value counts. 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 to be joined. After being chosen the two nodes are taken out of $cdts$ and the join node

is added to the list in their place. The process is continued until there is only the root of the tree is left in *cdts*.

5.3 The Fodp Algorithm

Given an ordered set of streams, the *Fodp* only looks at the trees that do not violate the given order, which means out of the $n!$ permutations it considers only a single permutation, i.e. the given order. *Fodp* finds an optimal join tree for a single permutation in $O(n^3)$ time.

Notice that given a set of n streams, the same joined tuples are produced regardless of the join tree being used. Consequently, the result size and the output rate of the n -way join should remain same also, regardless of the join tree being used. Therefore the principal of optimality holds. Once the optimal tree for a particular sequence in the given permutation of streams is known, it need not be recomputed. Let us use $T_{i,j}$ to denote an optimal join tree for a given fixed permutation $R_i \bowtie R_{i+1} \bowtie \dots \bowtie R_j$ 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 $R_i \dots R_k$ in its left subtree and $R_{k+1} \dots R_j$ in the right subtree. Then the following recurrence relation holds:

$$C(T_{i,i}) = 0$$

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

Notice that the n leaf nodes can only form $n - 1$ possible pairs since each node can only join with its predecessor or its successor in the given permutation; otherwise the given permutation will be violated. Following the above relation *Fodp* first finds the costs for the $n - 1$ pairs of leaf nodes, which gives their optimal costs since two nodes can be joined in only one way. After the optimal costs for each pair are known, the cost for each triplets are computed. There are $n - 2$ possible consecutive triplets for n nodes. For any triplet n_1, n_2, n_3 there are two possible choices $(n_1 \bowtie n_2) \bowtie n_3$ or $n_1 \bowtie (n_2 \bowtie n_3)$ since any other tree would violate the given order. The better of the two costs and structures are remembered. The algorithm proceeds in this manner until it finds the optimal tree for the whole sequence. Suppose a two dimensional triangular matrix, called *costMatrix*, is used to hold the cost for the optimal tree of a particular sequence of streams. The i^{th} row of the matrix contains cost for sequences of length i and the j^{th} column corresponds to sequences starting with stream R_j . Figure 5 gives pseudo-code for computing the cost matrix in *Fodp*.

5.3.1 Fodp variants

Although *Fodp* finds the optimal tree for a given order, it is handicapped by the order maintaining restriction. The goodness of the tree it produces is also dependent upon what order it starts with. Therefore we have developed three variants of *Fodp* that use the three heuristics discussed earlier, namely sort by arrival rate, sort by window size and sort by join ratio. We call these three variants *FodpAR*, *FodpW* and *FodpJR* respectively. Each *Fodp* variant first sorts the streams using its corresponding heuristic to make sure that it starts with a presumably good order. Then it applies the *Fodp* algorithm on this order to get the final join tree. The heuristics that orders by join ratio first computes the pairwise join ratio for each pair of streams and chooses the pair (R_1, R_2) with the smallest join ratio. At the next step it chooses the stream that has the smallest join ratio with either R_1 or R_2 . Once the third stream is picked every other stream is chosen on the basis that it has the smallest join ratio with the stream that was chosen last.

```

Initialize costMatrix[1][i] to 0 for all i;
for (row = 2; row ≤ n; row++)
  for (col = 1; col ≤ n; col++)
    initialize mincost, minleft, minleftc,
              minright, minrightc;
    for (i=1; i < row; i++)
      left = i;
      leftc = col;
      right = row - i;
      rightc = col + i;
      cost = getCost (left, leftc, right, rightc);
      if (cost < mincost)
        update all the min variables;
    update costMatrix[row][col];

```

Figure 5. pseudo code for Fodp

6 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.

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 [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. E.g. for our synthetic experiments x and y were both set to 5 seconds.

6.2 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. We take a different approach which is similar to the *moving state* strategy in [24]. 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. We say two join nodes are *exactly same* if they have the same left and right children. For example in the two trees from figures 6 (a) and (b), the nodes $R_4 \bowtie R_5$ 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 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 6 (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.

Figure 6. migrating partial results between trees

7 Experimental Results

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

Synthetic tests: We extensively tested the join trees generated by the algorithms that we discussed in this paper under different stream characteristics. Figure 7 shows the results for eight representative samples from these experiments. We do not report the results for *Fodp* since the *Fodp* variants always perform better than *Fodp* itself. 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 $dv_i \in \{800 - 1400\}$. 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 $\{1 \dots dv_i\}$ for that stream to ensure our containment assumption. For each testcase, 1 million tuples are generated and the processing time 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. 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. Note that the results account for all types of overheads. Overhead is measured as a percentage of the time spent in optimization and statistics gathering code out of the total execution time. When *OptDP* was used for $n=10$ the overhead was measured to be 3.5%. For all other cases it stayed below 0.3%. We noticed that *Fodp* variants performed significantly better than the *XJoin* variants and almost as good as *OptDP*. They performed either comparably or better (T2, T5, T6) than *XGreedyJoin*. On average, trees from *Fodp* variants can handle three times the load of the *XJoin* trees.

Tests using sensor data: The scenario for these experiments 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 of the following semantic form:

Figure 7. Performance comparison of stream join trees

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 8 reports the average throughput in tuples/sec for the different algorithms. In the figure noOpt stands for the un-optimized query plan, which joins the streams in the exact order that is given. Since all window sizes are same we did not test the *XJoinW* and *FodpW* algorithms. 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 , *FodpAR* and *FodpJR* significantly outperforms *XJoinAR* and *XJoinJR*. For $n = 10$ they are almost as good as the optimal solution *OptDP*. At least one of the *Fodp* variants is either comparable or slightly better than *XgreedyJoin* for each n .

8 Conclusion

We have considered the problem of finding optimal join trees for processing continuous queries over streaming sensor data and presented two algorithms, *Fodp* and *OptDP*. Our experiments show that our methods used in our system for adaptive re-optimization is inexpensive and the join trees produced by *Fodp* algorithms 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 study the tradeoff between adaptivity and runtime overhead. A potential avenue in this regard would be to find a way to measure and quan-

Figure 8. Comparing throughput

tify the variability of stream characteristics and adjust re-optimizing interval accordingly.

Acknowledgment

The authors would like to thank Sandia Corporation for providing source code for the copyrighted software Jess.

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. ACM Symp. on Principles of Database Systems*, 2002.
- [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. 21st Intl. Conf. on Data Engineering (ICDE'05)*, 2005.
- [5] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *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.
- [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. 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. ACM Intl. Conf. on Management of Data*, 2000.
- [10] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. 13th SIAM-ACM Symp. on Discrete Algorithms*, 2002.
- [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/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.
- [14] L. Golab and T. Oszu. Processing sliding window multi-joins in continuous queries over data streams. In *Proc. of VLDB*, 2004.
- [15] J. Gomes and H. Choi. Ares: an adaptively reoptimizing engine for stream query processing. Technical Report TR-GWU-CS-06-003. Department of Computer Science, George Washington University, 2006.
- [16] C. Intanagonwivat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):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. AAAI-87*, 1987.
- [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*, volume 23, 2000.
- [22] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *Proc. ACM Intl. Conf. on Management of Data*, 2002.
- [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.