PMJoin: Optimizing Distributed Multi-way Stream Joins by Stream Partitioning

Yongluan Zhou[1], Ying Yan[2], Feng Yu[1], and Aoying Zhou[2]

[1] National University of Singapore
[2] Fudan University

Abstract. In emerging data stream applications, data sources are typically distributed. Evaluating multi-join queries over streams from different sources may incur large communication cost. As queries run continuously, the precious bandwidths would be aggressively consumed without careful optimization of operator ordering and placement. In this paper, we focus on the optimization of continuous multi-join queries over distributed streams. We observe that by partitioning streams into substreams we can significantly reduce the communication cost and hence propose a novel partition-based join scheme - PMJoin. A few partitioning techniques are studied. To generate the query plan for each substream, a heuristic algorithm is proposed based on a rate-based model. Results from an extensive experimental study show that our techniques can sufficiently reduce the communication cost.

1 Introduction

Many recently emerging applications, such as network management, financial monitoring, sensor networks, stock tickers etc, fueled the development of continuous query processing techniques over data streams. In these applications, the data sources are typically distributed, e.g. the network hosts or routers in network management. Collecting all the data to a centralized server may not be cost-effective due to the high communication cost. Clearly, a distributed stream processing system is inevitable. Unlike traditional DBMS, where the processing in each node involves expensive I/O operations, stream processing systems often perform main memory operations. These operations are relatively inexpensive in comparison to the communication cost. As both the queries and data streams are continuous, a lot of existing work, such as [2], focus on minimizing the communication cost, especially when the source nodes are connected by a wide-area network. Furthermore, as the streams are continuous and unbounded, a rate-based cost model has to be used.

In this paper, we focus on multi-way window join query which is an important and expensive type of continuous queries. These queries may involve multiple streams from different source nodes. Let us look at an example drawn from the network management application.

Example 1. We want to monitor the traffic that passes through three routers and has the same destination host within the last 0.5 seconds. Data collected from the

Table 1. Distribution (tuples/ second)

S_i	S_1	S_2	S_3	$S_{1,2}$	$S_{1,3}$	$S_{2,3}$
λ_i	100.2	50.07	50.03	9.003	7.001	2.0008
λ_i^a	0.1	0.03	50	0.003	5	1.5
λ_i^b	0.1	50	0.01	5	0.001	0.5
λ_i^c	100	0.04	0.02	4	2	0.0008

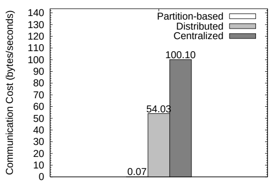


Fig. 1. Communication cost of plans

three routers feed three streams s_1, s_2 and s_3 to three processing nodes n_1, n_2 and n_3. The content of each stream tuple includes the destination host ip *dest* of a data packet and possibly other information. This task can be represented in a three-way window join query $S_1 \bowtie_{S_1.dest=S_2.dest} S_2 \bowtie_{S_2.dest=S_3.dest} S_3$ where the window size of each stream is 0.5 seconds.$\square$

In Table 1, λ_i denotes the rate of stream S_i and λ_i^a denotes the rate of tuples from S_i whose value in the *dest* attribute is a. Furthermore, $S_{i,j}$ is the result stream of $S_i \bowtie S_j$ and its rate is denoted as $\lambda_{i,j}$. The minimum communication cost that can be achieved under different schemes are as follows:

1. Centralized scheme: The best plan in this category is to send both S_2 and S_3 to n_1. If we assume the tuple size of every stream is 1 byte, then this scheme results in communication cost of $\lambda_2 + \lambda_3 = 100.1$(bytes/sec).

2. Distributed scheme: In this category, the best plan is to send S_3 to n_2 first and then ship the result $S_{2,3}$ to n_1. If we assume the tuple size of a join result tuple is the sum of the two input tuples, we can derive the communication cost of this plan as $\lambda_3 + \lambda_{2,3} \times 2 \approx 54.03$(bytes/sec).

3. Partitioned-based scheme: taking a closer look at the problem, we can find that the arrival rates of tuples vary with different values in the joining attributes. Furthermore, the popularity of the values in different streams also vary. Hence the optimal plans for these tuples are also different. For example in Table 1, $dest = a$ is popular in S_3 while it is unpopular in the other two streams. Hence the best plan for these tuples is to ship them from S_2 to n_1 to join with S_1 and then the resulting tuples are sent to n_3 to join with S_3. This results in the cost of 0.036 (bytes/sec). However, for those tuples with $dest = b$, the best plan is totally different: those tuples from S_3 should be sent to n_1 and then to n_2. By exploiting this characteristic, the minimum communication cost that we can get for Example 1 is approximately 0.07 (bytes/sec).

In this paper, we focus on the static optimization of multi-join queries. Static optimization can be applied to applications where the stream's characteristics are relatively stable and their changes are predictable. Moreover, given that our problem has not been previously studied, it is important to examine how static optimization can be performed before extending the work to a dynamic context. To summarize, our main contributions are as follows:

– We formulate the problem and propose a heuristic-based optimization algorithm to decide the join operation locations and the tuple routing orders based on a rate-based cost model.

- To further reduce the communication cost, we propose a novel join scheme: PMJoin. We also study different partitioning strategies (e.g., rate-based, hash, etc).
- We fully implemented the system and run a simulation study. The study shows the efficiency of our techniques.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 presents the proposed techniques. In Section 4, we perform extensive performance studies on our implementation. Section 6 concludes the paper.

2 Related Work

Distributed processing of multi-way join have already been extensively studied in the context of traditional relational database systems. [13] provides a thorough survey on this area. The optimizers in Distributed INGRES [9] and System R* [14] consider both CPU and I/O cost as well as the communication cost of processing a whole dataset. In these systems the I/O cost are so high that they cannot be omitted. SDD-1 [7] uses heuristics to optimize the utilization of semi-join. Semi-join is useful when a tuple is much larger than a single attribute and the selectivity is low. However, semi-join is not readily applicable to window join processing. This is because streams are normally continuous and queries should be evaluated in a nearly real time manner. For example, a tuple t_i may be pruned away because there is no matching tuples in the opposite window. However, new tuples may arrive at the opposite window which may match t_i. Extra complicated mechanisms have to be introduced to ensure the correctness. As shown in our study, we believe our PMJoin, together with the optimization heuristics, is a promising alternative to reduce the communication cost. Our techniques can also be adapted for traditional passive data processing whose performance needs further study. The above-mentioned systems and a considerable amount of work (e.g. [8, 16, 21]) have also exploited horizontal fragmentation of relations to increase the parallelism and consequently to reduce the response time. Static and dynamic data allocation [3, 17, 20] try to allocate replications to reduce communication cost or to balance the load on servers. However, none of the above techniques exploit generating different plans for different partitions. Furthermore, a rate-based cost model has to be used in our problem.

[12] studies techniques for the evaluation of window join queries over data streams. [19, 10] examine the processing of multiple joins over data streams. [4, 15, 5, 6] investigate the static and adaptive ordering of operators for continuous queries over data streams. However, all these studies focus on centralized processing. There are also several recent efforts devoted to extending centralized schemes to distributed context. [1] proposes the design of a distributed stream system. [2] studies the operator placement problem for stream processing. However, these approaches assume there is an already optimized query plan and then allocate the operators, while our approach does not impose such an assumption. Furthermore, they do not explore partitioning of the streams to further optimize the plans. In [18], the operators are assumed to have been allocated, and the proposed scheme adaptively decides the routing order of the tuples.

3 Distributed Multi-join

In this section, we first formulate the problem and then present the scheme to generate a query plan for each substream. It also applies to the case without stream partitioning. Then we study how stream partitioning can be applied to minimize communication cost.

3.1 Problem Formulation

In our system, there is a set of geographically distributed data stream sources $\mathit{\Sigma} = \{S_1, S_2, \cdots, S_{|\mathit{\Sigma}|}\}$ and a set of distributed processing nodes $\mathcal{N} = \{n_1, n_2, \cdots, n_{|\mathcal{N}|}\}$ interconnected by a widely distributed overlay network. Since the data stream sources in practice may not have the ability to communicate with multiple nodes, we separate the data sources from the processing system by assigning nodes as delegations of data sources. Streams are routed to the various processing nodes through their delegated nodes. A multi-way window join query may involve streams from multiple nodes. For simplicity, we assume the queries do not involve stored tables.

As mentioned before, our main concern is to minimize the communication cost. We adopt the unit-time cost paradigm and hence communication cost of a processing scheme $\mathit{\Omega}$ can be computed as $C(\mathit{\Omega}) = \frac{\textit{Amount of communications (in bytes)}}{\textit{Observation period}}$.

The formal problem statement is: *Given a m-way window join ($\forall m < |\mathit{\Sigma}|$) query Q, which involves a set of streams $\mathit{\Sigma}$ and they are located at a set of nodes $\mathcal{N}$, find a join scheme $\mathit{\Omega}$ so that the total communication cost $C(\mathit{\Omega})$ is minimized.*

3.2 Join Operation Locations and Tuple Routing Orders

Before processing the queries, we have to first decide the placement of the join operators. Then we have to route the streams and the intermediate result streams (if necessary) around the nodes. In this subsection, we focus on how to decide the location of the join operations as well as the routing order of the tuples for each substream. Since it also applies to streams without partitioning and we treat each substream independently, we use the term "stream" instead of "substream" in the following discussions. The evaluation of the join operations allocated to each node can use any of the existing centralized join optimization and processing techniques, e.g. [19, 10]. In this paper, we assume the join operations in each node are evaluated using MJoin [19]. In this technique, one in-memory index structure, e.g. hash tables, is built for each joining stream. The joining stream could be a source stream or an intermediate result stream generated by another node. When a tuple from a joining stream arrives, it would be inserted into its corresponding index structure, and be used to probe other index structures one by one to evaluate the query. The optimization of the probing order has already been studied in centralized processing literatures [4, 6, 19] and would not be considered in this paper.

Notations and Cost Model. Let the set of streams and the set of nodes involved in the query Q be $\mathit{\Sigma}$ and $\mathcal{N}$, respectively. The set of streams that are located in $n_i \in \mathcal{N}$ is denoted as $\mathit{\Sigma}_i$. The result stream of $S_i \bowtie S_j$ is denoted as $S_{i,j}$ and the result stream of $S_{i,j} \bowtie S_k$ is denoted as $S_{i,j,k}$ and so on. If two streams are located at one node, we say that they are co-located. A function $col_{i,j}$ is defined as follows:

$$col_{i,j} = \begin{cases} 0 & : \quad S_i \text{ and } S_j \text{ are co-located} \\ 1 & : \quad \text{otherwise} \end{cases} \tag{1}$$

We adopt a rate-based cost model similar to the one developed in [5]. The arrival rates of streams S_i and $S_{i,j}$ are denoted as λ_i and $\lambda_{i,j}$, respectively. Let W_i and W_j be the expected number of tuples in the window of S_i and S_j, respectively. For a tuple-based window, W_i is equal to the window size K_i, while for a time-based window, W_i is equal to $\lambda_i \cdot T_i$, where T_i is the window size. To estimate $\lambda_{i,j}$, we note that for every unit time, λ_i tuples from S_i and λ_j tuples from S_j would be used to probe the windows of S_j and S_i, respectively. Out of the $\lambda_i \cdot W_j + \lambda_j \cdot W_i$ pairs of tuples, $f \times (\lambda_i \cdot W_j + \lambda_j \cdot W_i)$ matches are expected to be found, where f is the join selectivity. Therefore the expected number of tuples generated by $S_i \bowtie S_j$ per unit time can be estimated as

$$\lambda_{i,j} = f \times (\lambda_i \cdot W_j + \lambda_j \cdot W_i) \tag{2}$$

The tuples in the active window of the result stream $S_{i,j}$ are composed of those result tuples that are the join results of the tuples in the active window of S_i and S_j. Hence the expected number of tuples in the active window of $S_{i,j}$ can be computed as

$$W_{i,j} = f \cdot W_i \cdot W_j \tag{3}$$

Eqs. (2) and (3) can be recursively applied to obtain the values for multiple joins. Furthermore, it should be noted that the output rate and the window of the join result of a set of streams are independent of how the join is performed. Hence for a given distributed query plan, we can compute its unit-time communication cost by computing the rates of the streams that are sent over the network.

A Heuristic Algorithm. Given the above cost model, we can use a specific searching algorithm to search a specific solution space. For example, we can use dynamic programming to select an optimal plan from all the left deep tree plans. The computation complexity of the algorithm is $O(n!)$. However, as we will see soon, the search algorithm has to be applied several times in our partition-based join approach. Hence we will propose a much cheaper algorithm which runs in $O(n^2)$ time. Algorithm 1 shows the proposed stream join optimization algorithm. The input of the algorithm is the set of streams Σ involved by the query as well as the join graph representation G of the query. A join graph consists of a set of vertices each representing a stream and a set of edges each representing a join operation between the two connected streams. Furthermore, each vertex in the graph is annotated with the source node of the corresponding stream. We use the following example to illustrate.

Example 2. A query joins 5 streams: S_0, S_1, S_2, S_3 and S_4 which are spread over 3 nodes. Figure 2(a) shows the join graph of this query. The location of each stream is drawn around each vertex. The selectivities of the join operations are also drawn around the corresponding edges. Columns $2-6$ in Table 2 list the arrival rates λ_i and the expected number of tuples in the window W_i of these source streams.

For brevity, we assume that tuples from every stream (either a source stream or an intermediate result stream) have the same sizes in the following discussions. The adoption of this assumption does not lose any generality as we can always incorporate the tuple sizes in the calculation of cost without changing the algorithm.

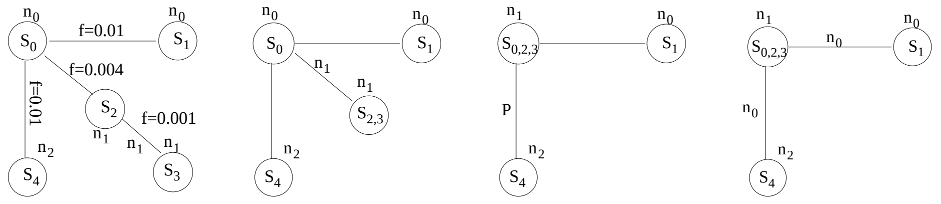
Algorithm 1: StreamJoinOpt(Σ, G)

```
Input: $\Sigma$: A set of streams;
G: A join graph over $\Sigma$;
1  begin
2      for each $n_i \in \mathcal{N}$ do
3          Sort $\Sigma_i$ in increasing arrival rates;
4          for $j = 0; j < |\Sigma_i|; j++$ do
5              for $k = j + 1; k < |\Sigma_i|; k++$ do
6                  if $\lambda_{\Sigma_i[j] \bowtie \Sigma_i[k]} < \lambda_{\Sigma_i[j]}$ then
7                      Label the join between $\Sigma_i[j]$ and $\Sigma_i[k]$ as local;
8                      $\Sigma_i[j] \leftarrow \Sigma_i[j] \bowtie \Sigma_i[k]$;
9                      $\Sigma_i \leftarrow \Sigma_i - \Sigma_i[k]$;

10     Sort $\Sigma$ in increasing arrival rates;
11     while $|\Sigma| > 1$ do
12         $\Sigma_p \leftarrow$ the slowest stream $S_i$;
13         $\Sigma -= S_i$;
14         repeat
15             $cost \leftarrow MaxNumber$;
16             for each stream $S_j$ joinable with any stream in $\Sigma_p$ do
17                 if $C(\Sigma_p + S_j) < cost$ then
18                     $k \leftarrow j$;
19                     $cost \leftarrow C(\Sigma_p + S_j)$;

20             label the edges that connect any stream in $\Sigma_p$ and $S_j$ as pending;
21             if case (1) is chosen then
22                 assign all the pending join operations to the node of $S_j$;
23                 $S_p \leftarrow$ Collapse $\Sigma_p$ and $S_k$ to one node ;
24                 $\Sigma_p \leftarrow S_p$;
25             else
26                 $\Sigma_p += S_k$; $\Sigma -= S_k$;
27         until $|\Sigma_p| = 1$;
28         Insert $\Sigma_p$ into $\Sigma$;
29 end
```

At the first step (lines 2 - 9) of the algorithm, we find whether there is any locally evaluable join operation which can result in a stream whose rate is smaller than both joining streams. Evaluating these joins locally tends to reduce the potential communication cost if some of the streams need to be shipped out to other sites. For Example 2, there are two locally evaluable joins: $S_0 \bowtie S_1$ and $S_2 \bowtie S_3$. By using Equations (2) and (3), $\lambda_{S_0 \bowtie S_1}$ and $\lambda_{S_2 \bowtie S_3}$ can be estimated as 70 and 15, respectively. Hence we choose to allocate $S_2 \bowtie S_3$ to n_1 and we label the corresponding edge with n_1. For ease of processing, once a join operation is allocated, we would collapse the two connected vertices in the join graph and the resulting vertex represents their join result stream. By applying this to Figure 2(a), we can derive Figure 2(b). The rate and window size of $S_{2,3}$ are also listed in column 6 of Table 2.



(a) Join Graph 1 (b) Join Graph 2 (c) Join Graph 3 (d) Join Graph 4

Fig. 2. Processing steps for an example query

Table 2. Parameters of streams

S_i	S_0	S_1	S_2	S_3	S_4	$S_{2,3}$	$S_{0,2,3}$
λ_i	10	35	25	30	15	15	9
W_i	100	350	250	300	150	75	30

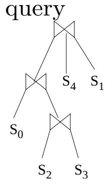


Fig. 3. The plan tree

In the second part (lines 10 - 28) of the algorithm, we employ a heuristic approach to allocate the remaining join operations. There are two nested loops in this part. For each iteration of the outer loop, we will determine the location of a subset of join operations. First, we pick a stream with the smallest rate, say S_i. This is because it may result in less communication cost if S_i has to be transmitted over the network. Next, to evaluate the join between S_i and each of the other streams S_j that are joinable with S_i, we have two cases:

1. Send S_i to the node of S_j. The potential communication cost of this case is equal to the sum of the cost of sending S_i to the node of S_j and the potential cost of sending out the result stream of $S_i \bowtie S_j$, i.e. $\lambda_i \cdot col_{i,j} + \lambda_{i,j}$. The second term is to count the potential cost of sending out the result stream to perform other join operations.
2. Send both S_i and S_j to a third site. The potential cost of this case is $\lambda_i + \lambda_j$.

For each stream, the case with smaller cost is used. We greedily choose a stream S_k with the smallest estimated cost and move it from Σ to Σ_p. If case (1) is chosen for S_k, that means the join operation is already allocated. We will remove streams S_i and S_k from Σ and add the result stream $S_{i,k}$ to Σ and start a new iteration. Correspondingly, in the join graph, we will collapse nodes S_i and S_k into one node $S_{i,k}$. However, if case (2) is chosen for S_k, that means the join operation is still pending for allocation. We will search for another stream S_l that is joinable to any stream in Σ_p with the smallest cost. The cost estimation is similar to the above analysis. To ease the presentation of the algorithm, we define the following function:

$$C(\Sigma_p + S_j) = \min\{\sum_{S_i \in \Sigma_p} \lambda_i + \lambda_j, \sum_{S_i \in \Sigma_p} \lambda_i \cdot col_{i,j} + \lambda_{\Sigma_p,i}\} \quad (4)$$

For example, in Figure 2(b), we first add the slowest stream S_0 to Σ_p. Then for the three joinable streams $S_1, S_{2,3}$ and S_4, using Eqs. (2), (3) and (4), we can find that $C(\Sigma_p + S_{2,3})$ is the smallest. Furthermore, case (1) should happen, i.e. S_0 will be sent to node n_1 to perform the join with $S_{2,3}$. Hence we label the edge between S_0 and $S_{2,3}$ with n_1. Then we collapse nodes S_0 and $S_{2,3}$ to one node $S_{0,2,3}$. This

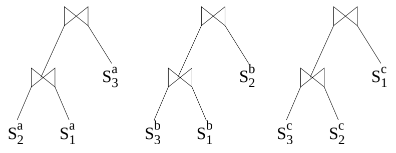


Fig. 4. Plans for the substreams in Example 1

results in Figure 2(c). The rate and window of $S_{0,2,3}$ is computed using Eqs. (2) and (3) and listed in column 7 of Table 2. Now a new iteration of the outer loop in the second part of the algorithm has to be started. The currently slowest stream is $S_{0,2,3}$, hence it is added to Σ_p. Among the two joinable streams S_1 and S_4, the potential cost of adding S_4 is smaller. This time, case (2) is chosen, i.e. $S_{0,2,3}$ and S_4 have to be sent to a third site. We label the edge between node $S_{0,2,3}$ and S_4 with a P to indicate that the join operation is pending for allocation. Then the last stream S_1 has to be chosen and S_1 and $S_{0,2,3}$ have to be sent to n_0 to perform the joins. Now the two join operations can be labeled with n_0. Then all the join operations have already been allocated.

The output plan of Algorithm 1 can be represented using a tree. In the tree, each leaf node is a source stream and each intermediate node is an MJoin operator. Each MJoin operator is located in one node and has two or more input streams. We order these streams in the order such that the right most stream (or abbreviated as the right stream) have the same location with the MJoin operator. That means all the other input streams of this MJoin operator would be sent over to the location of the right stream to perform the join operations. Figure 3 shows the tree representation of the output plan of Example 2.

3.3 Stream Partitioning

In a partition-based scheme, each stream S_i may be partitioned into D substreams $S_i^1, S_i^2, \ldots, S_i^D$ based on the values on the joining attribute. This is based on the observation that the arrival rates of tuples with different values may vary much inside each single stream. Hence the optimal scheme for these tuples are different. We denote the rate of a substream S_i^k as λ_i^k.

PMJoin. In this subsection, we will look at how the partition-based join can be applied to a multi-way equijoin query whose join predicates are specified on a single attribute, say *attr*. This kind of queries is common in a lot of applications, such as Example 1 in Section 1. Furthermore, these could also be a subset of predicates in a multi-way join query that are specified on the same attribute. We propose a scheme that is called Partition-based Multi-way Join (PMJoin) to evaluate this set of join predicates. Every stream involved in these join predicates is partitioned into multiple substreams on *attr*. The substreams of all the streams can be grouped into D groups. The kth group of substreams is $\{S_1^k, S_2^k, \ldots, S_{|\mathcal{N}|}^k\}$. For each group of substreams, we can use Algorithm 1 to decide the allocation of the join operations.

We illustrate the plan of PMJoin by using Example 1. First, based on the value of the *dest* attribute, we partition each stream into three substreams S_i^a, S_i^b and S_i^c. These streams are grouped into three groups. Then for each group of substreams, we use Algorithm 1 to optimize the plan. The resulting plans for the three groups of substreams are shown in Figure 4.

To get the lowest cost, we can partition each stream into as many substreams as possible. For example, we can put tuples with each distinct value in the joining attribute into one substream. Let the number of these values be R then we could partition the stream into R substreams. However, it is clear that with more partitions, more plans have to be generated and it complicates the processing. So we adopt a more flexible approach where the number of partitions can be specified as any D. This can be viewed as clustering the above finest substreams (i.e., one substream per value) into D partitions. In the following discussions, we refer to these finest substreams as FStreams. FS_i^k stands for the kth FStream from stream S_i. And the unique *attr* value of the tuples of a FStream is called the value of the FStream. We consider three approaches:

1. Hash partition. A hash function can be applied to hash the values of the FStreams into one of the D buckets. The FStreams in each bucket compose a substream. This is actually a random partitioning method.

2. Range partition. Divide the data range into D sub-ranges. FStreams whose values fall into the ith sub-range compose the ith substream.

3. Rate-based partition. The above two approaches ignore the arrival rates of the various FStreams. A good partitioning method should put those groups of FStreams whose optimal plans are similar to each other in one partition. In this way, the generated plan for that partition would be good for all its FStreams. Here we use an approximate approach to estimate the similarity of the optimal plans of two groups of FStreams. For each group of FStreams, $\{FS_1^k, FS_2^k, \ldots, FS_{|\mathcal{N}|}^k\}$, we sort them in increasing order of their arrival rates. Then we create a vector V_k where the ith element indicates the position of FS_i^k in the above sorted list. For example, if we have a sorted list as $\langle FS_3^k, FS_1^k, FS_2^k\rangle$, then $V_k = \langle 2, 3, 1\rangle$. So the distance between the kth and the lth groups of FStreams is measured by the distance between V_k and V_l, which is measured as $|V_k - V_l|$. The intuition is that the more similar the sorted lists of the two groups of FStreams are, the more similar their optimal plans would be. Now we can employ any clustering techniques to cluster the groups of FStreams into D clusters. In this paper, we adopt the k-Means approach [11].

To apply all the above mechanisms, we need to know the rate of each FStream. To reduce the cost of maintaining such statistics, we can use traditional histogram approaches. Only statistics of histogram buckets are maintained, and the rates of an FStream is estimated based on the statistics of the bucket it belongs to.

Multi-join on different attributes. For a generic multi-join query whose joins involve multiple attributes, our approach works as follows. We first run Algorithm 1 to determine the plan for the scheme without partitioning. Given the output plan of Algorithm 1, we will try to find out several sets of join predicates where we can apply PMJoin.

We call a MJoin operator to be *partitionable* on *attr* if the join predicates in the Mjoin operator are all (equalities) on the same attribute *attr*. The procedure to find the subset of join predicates to apply partitioning works in two steps. In the first step, from the output plan of Algorithm 1, we try to aggressively determine the subsets of join predicates that can be partitioned by using Algorithm 2. The algorithm starts from the root. If the current operator is found to be partitionable

Algorithm 2: FindPartition(O_i)

```
Input: $O_i$: an MJoin operator ;
$R$: an boolean array, $R[i]$ is true if $O_i$ is the right child of its parent;
1 begin
2     if !$R[i]$ AND $O_i$ is partitionable on an attribute attr then
3         Mark $O_i$ as PMJoin ;
4         for each child operator $O_j$ of $O_i$ do
5             if $O_j$ is partitionable on attr then
6                 Merge $O_j$ to $O_i$;
7     for each child operator $O_j$ of $O_i$ do
8         FindPartition($O_j$);
9 end
```

on an attribute, say *attr*, it would be marked as a PMJoin operator. Then if any child of the current operator is also partitionable on *attr*, it would merge that child with the current operator. Note that after the merge, the prior grandchildren would become children of the current operator. These new children would also be searched to see if they can be merged. After the merging attempt, we recursively call the algorithm on each child of the current operator.

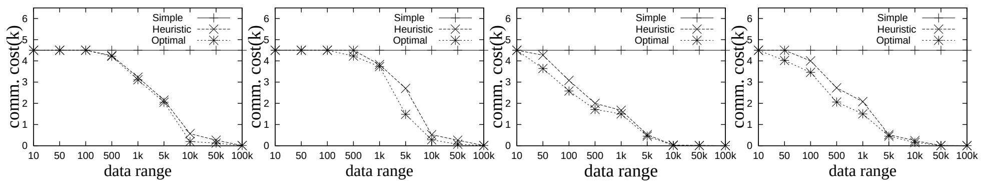
In the second step, we try to select some of the PMJoins from those found by the above algorithm. Note that the output stream of a PMJoin consists of a number of substreams that would be located at several sites. For example, the result stream $S_{1,2,3}$ of Example 1 consists of three substreams that are located at n_1, n_2 and n_3. Now suppose the result stream has to join with another steam, say S_i, on another attribute. If PMJoin is used to join $S_{1,2,3}$ and S_i, we have to repartition the substreams of $S_{1,2,3}$ that are located at the three nodes. Furthermore, the substreams of S_i may have to be sent to all these three nodes. This results in high communication cost. Therefore, we opt to impose two constraints on the application of PMJoin. (1) The input streams of a PMJoin should be located at a single node. That means a PMJoin cannot be the child of another PMJoin. (2) The right child of a MJoin operator cannot be a PMJoin operator. Otherwise, the other input streams of the MJoin operator have to be sent over to the output nodes of that PMJoin.

Our heuristic, which is given below, favors those PMJoins that have high input stream rates. This is because they may provide more opportunities to reduce the communication cost by using PMJoin.

1. Sort all the PMJoins on the total input stream rates.
2. Remove the one with the largest input stream rate.
3. Remove the parent PMJoin (if any) from the sorted list, and restore it back to one or more MJoin operators.
4. If the list is not empty go to step 2.

4 Performance Study

In this section, we present a performance study of our techniques. We fully implemented the system using Java. To ease the control of experiments, we use a discrete



(a) LASW, SD (b) HASW, SD (c) LASW, DD (d) HASW, DD

Fig. 5. Performance of the Heuristic Algorithm

event simulation package JavaSim to simulate the distributed processing effect. For each experiment, we would collect the total communication cost for every second. Without loss of generality, we assume that tuples from all the streams and the join result tuples have the same sizes. Hence we only count the number of tuples that are transmitted over the network. Without loss of generality, we assume the joining attributes are of integer values and the windows are all time-based windows specified in seconds. All the arrival rates are specified in the unit of tuples/second. To model different data frequencies, we use various types of distributions. The value distributions are chosen from the following distributions: (1) Uniform distribution, (2) Normal distribution with the mean being the mid value and varied standard deviation, (3) Zipf distribution with the skew parameter θ varied from 0.1 to 1.5, (4) Self-Similar with the skew parameter h varied from 0.1 to 0.9. (For integers $1 \ldots N$, the first $h \cdot N$ integers gets $1-h$ of the weight.) To examine the performance of our heuristic algorithm, we compare it with two algorithms: (1) Simple: send the other streams to the location of the stream with the highest rate; (2) Optimal: exhaustively enumerate the possible plans and choose the best one.

4.1 The Heuristic Optimization Algorithm

In the first experiment, we consider the following different situations: (1) streams with lower arrival rates have smaller window sizes (LASW); (2) streams with higher arrival rates have smaller window sizes (HASW). Both situations would be studied under two senarios: similar data distribution (SD) and different data distributions (DD). For the similar distribution scenarios, we randomly choose 10 zipfian distributions with the skew parameters varied from 0.1 to 0.3. For the case with different distributions, we randomly choose 10 distributions from all those listed above. We vary the data ranges from 1-10 to 1-100000. Note that query selectivities would be smaller with larger data ranges. Each stream is from a different node. Figure 5 presents the results of this experiment. From the figures, we can see that the communication cost of the Simple approach is constant to various data ranges. That is because this approach simply chooses to send all the other 9 streams to the location of the fastest stream. Thus, the communication cost is equal to the sum of these 9 streams. For the heuristic and the optimal approach, when data range is small, the communication cost are the same as that of Simple. The reason is the selectivities of the join operations are high and any intermediate result streams would have relatively large rates. Hence the best plan here is the same as Simple. However the communication cost of the heuristic and optimal approach drops with the increase in data ranges. That is because the join operations become more selective, hence it brings more benefits to perform distributed processing to minimize the commu-

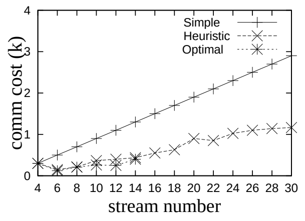


Fig. 6. #streams

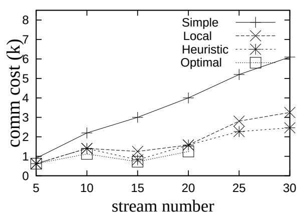


Fig. 7. #streams/node

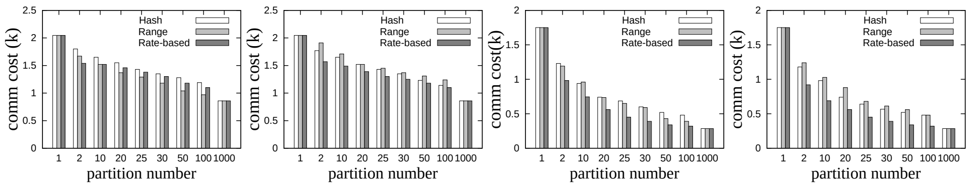
nication cost. Furthermore, we can see that our heuristic algorithm performs very close to the optimal approaches under the 4 different situations.

In the second experiment, we study the effect of the number of streams. We fix the data range at 1-1000 and all the arrival rates at 100 tuples/second. We vary the number of streams and randomly choose the window sizes from 10 to 100 seconds. Data distributions are also randomly chosen. We compare our heuristic algorithm with the Simple method and the Optimal algorithm. Due to the long running time of the Optimal algorithm, we can only get the results up to 14 streams. The results are presented in Figure 6. We can see that the cost of the Simple method increases proportionally as the number of streams increases. The improvements of the heuristic approach and Optimal approach over the Simple method is larger with larger number of streams. That is because more steams provide more opportunities to optimize allocation of join operations to reduce the communication cost. Furthermore, we can see that the heuristic approach is very close to the Optimal algorithm.

In the third experiment, we examine the effect of the number of streams on each node, which is handled by the first step of our heuristic algorithm. We fix the number of nodes in this experiment to 5. The streams are randomly allocated to the nodes and their arrival rates are varied from 100 to 500. The other configurations are similar to the earlier experiment. We compare our heuristic to (a) Optimal: the optimal scheme and (b) Local: the one with the first step replaced by simply joining all the local streams. When the total number of streams increases, the average number of streams in each node also increases. The results are shown in Figure 7. We can see that our heuristic works better than Local. That is because it would only perform those joins that would reduce the rates, while Local would perform also those that may increase the rates.

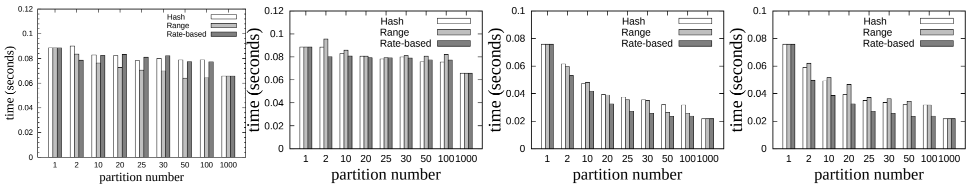
4.2 PMJoin

In the first experiment, all the join predicates are equalities on a single attribute. Hence PMJoin can be used here. We use 10 streams with arrival rates varying from 10 to 1000 and window sizes randomly chosen from 10 to 100. We fix the data range of all the streams to 1-1000. Each stream is located at one node. We vary the partition number of our partition functions to examine the sensitivity of the PMJoin. Note that when the partition number is equal to 1, it is the same as the scheme without partitioning. When the partition number is the same as the data range, there is only one value in each partition. To examine the effect of the different partition methods, we study two cases: (1) the values in a "hot spot" is randomly spread over the data range (Random Hot Spot: RHS); (2) the values are



(a) CHS, SD (b) RHS, SD (c) CHS, DD (d) RHS, DD

Fig. 8. Performance of PMJoin



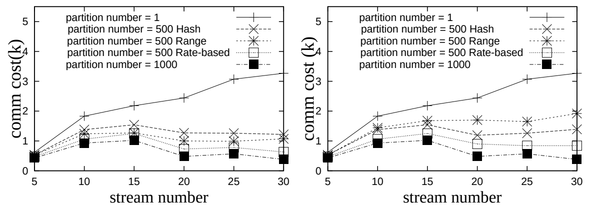
(a) CHS, SD (b) RHS, SD (c) CHS, DD (d) RHS, DD

Fig. 9. Routing overhead comparison

contiguously located in the data range (Contiguous Hot Spot: CHS). Under each case, we also study two senarios: similar distribution (SD) and different distribution (DD) mentioned in Section 4.1. The results are shown in Figure 8. When streams have similar distributions, PMJoin has only moderate improvement over the non-partitioning approach. That is because the frequent values among the streams are similar. That means most of the substreams would have similar plans as that of the non-partitioning scheme. However, in the case of different distributions, we found that large communication cost can be saved even when the streams are partitioned into only two substreams. The reason is the optimal plans of the substreams are much different from each other. PMJoin optimizes their plans independently, hence results in less communication cost. Furthermore, as we can see, finer granularities of the partition function can result in larger improvements.

Furthermore, in Figure 8(a), range partition works the best most of the time. That is because in this situation, the groups of FStreams in a contiguous range would have similar optimal plans. Rate-based partition works better only when there are two partitions. The bad performance of Rate-based partition is due to the fact that the data distributions are all zipfian distributions. When there are more than two partitions, most of the groups of FStreams would have the same sorted list and hence the same vector V_k. So the distances between them are all 0. As a result, the k-Means clustering algorithm randomly places them into different partitions. However, for the other three conditions, rate-based partition works persistently the best. This is attributed to its ability to identify those groups of FStreams that have similar optimal plans. Range partition loses its advantage because those groups of FStreams having similar optimal plans are not contiguous. It works worse than hash partition when the hot spot is randomly spread over the data range.

One may worry that PMJoin would bring too much routing overheads due to its more complicated routing mechanisms (each substream has a different routing order). Here we conduct another experiment to measure its overhead. The configurations are the same as the experiment above. We use our implementation to compare the routing cost of PMJoin with different number of partitions. Figure 9 shows the cpu time used for routing in each second. Surprisingly, most of the time,



(a) Contiguous Hot Spot (b) Random Hot Spot

Fig. 10. Sensitivity to #Streams

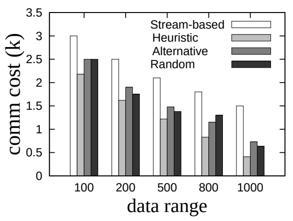


Fig. 11. Different Attributes

PMJoin has even lower routing cost than the scheme without partitioning (i.e. partition number = 1). This can be attributed to the ability of PMJoin to minimize the communication cost. Because fewer tuples are routed in PMJoin, its routing cost is smaller. PMJoin is more powerful in the case of different data distribution, hence its routing cost is much lower than the scheme without partitioning in this case. In addition, a better partitioning scheme further reduces more routing cost.

In the third experiment, we study the sensitivity of PMJoin to the number of streams. The distributions of the streams are randomly selected which is similar to the "Different Distributions" above. The results are presented in Figure 10. We only present the results when the partition number is 1, 500 and 1000, respectively. The others would lie between them. We can see that with increasing number of streams, PMJoin has larger improvement over the scheme without partitioning. Rate-based partition works the best under various number of streams. Range partition works better than hash partition for contiguous hot spot, while the reverse is true for randomly spread hot spot. Interestingly, with the increase in the number of streams, the cost of the scheme without partitioning increases while those of most of the PMJoin schemes decrease. Note that the distributions of the streams in this experiment are different, hence with more number of streams, the selectivity of the query is decreased. That means more tuples can be dropped before reaching the output, which can save the communication cost. PMJoin provides more opportunities to exploit this effect by using different plans for different group of substreams.

4.3 Multi-Joins on Different Attributes

In this section, we examine our techniques for multi-join queries whose equality predicates involve different attributes. We compare our heuristics to select the PMJoin operator with two other approaches: (1) Random: replace the step 1 and 2 in the heuristic with a random selection; (2) Alternative: choose from those not selected by the heuristic algorithms. We randomly select 20 streams with different distributions. Their arrival rates vary from 100 to 1000 tuples/second and their window sizes vary from 10 seconds to 100 seconds. These streams are randomly allocated to 10 nodes. 200 random queries are generated with the number of joining attributes varied from 3 to 7. We get the average resulting cost of these queries under the three approaches. Figure 11 shows the results under different data ranges. In all the cases, the scheme without partitioning performs the worst. With larger data ranges (i.e. lower selectivity), the partition-based scheme is more beneficial. Furthermore, our heuristic outperforms the other two approaches.

5 Conclusion

In this paper, we studied the optimization of multi-join queries over distributed data streams. We proposed a heuristic optimization algorithm to minimize the communication cost. Furthermore, a partition-based join scheme: PMJoin was presented. Different partition techniques were discussed and heuristics to utilize PMJoins were also proposed. Our performance study showed that our techniques can sufficiently reduce the communication cost of the system. Although we propose the techniques under the context of distributed stream processing, the techniques can also be adapted to traditional distributed database systems. Further performance study in this context is required.

References

1. D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.
2. Y. Ahmad and U. Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, 2004.
3. P. M. G. Apers. Data allocation in distributed database systems. *ACM Trans. Database Syst.*, 1988.
4. R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.
5. A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, 2004.
6. S. Babu et al. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.
7. P. A. Bernstein et al. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 1981.
8. D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *VLDB*, 1985.
9. R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *SIGMOD*, 1978.
10. L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.
11. A. Jain and R. Dubes. *Algorithms for Clustering Data.* Prentice Hall, 1998.
12. J. Kang et al. Evaluating window joins over unbounded streams. In *ICDE*, 2003.
13. D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.
14. G. M. Lohman et al. Query processing in r*. In *Query Processing in Database Systems.* Springer, 1985.
15. S. Madden et al. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.
16. D. Shasha and J. T.-L. Wang. Optimizing equijoin queries in distributed databases where relations are hash partitioned. *ACM Trans. Database Syst.*, 1991.
17. J. Sidell et al. Data replication in mariposa. In *ICDE*, 1996.
18. F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.
19. S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.
20. O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 1997.
21. C. T. Yu et al. Partition strategy for distributed query processing in fast local networks. *IEEE Trans. Software Eng.*, 1989.