Evaluating Window Joins over Unbounded Streams

Jaewoo Kang Jeffrey F. Naughton Stratis D. Viglas

University of Wisconsin-Madison
Computer Sciences Department
1210 West Dayton Street
Madison, WI 53706, USA
{jaewoo, naughton, stratis}@cs.wisc.edu

Abstract

We investigate algorithms for evaluating moving window joins over pairs of unbounded streams. We introduce a unit-time-basis cost model to analyze the expected performance of these algorithms. Using this cost model, we propose strategies for maximizing the efficiency of processing joins in three scenarios. First, we consider the case where one stream is much faster than the other. We show that an asymmetric combination of hash join and nested loops join (hash join on one input, nested loops join on the other) can outperform both the symmetric hash join and the symmetric nested loops join. Second, we investigate the case where system resources are insufficient to keep up with the input streams. We show that we can maximize the number of join result tuples produced in this case by properly allocating computing resources across the two inputs streams. Finally, we investigate strategies for maximizing the number of result tuples produced when memory is limited. We revisit the first two cases in this context, and show that proper memory allocation across the two input streams can result in significantly lower resource usage and/or more result tuples produced.

1. Introduction

Recently the database research community has begun focusing its attention on query processing over unbounded continuous input streams rather than fixed-size stored data sets. In such environments, many assumptions made in traditional query processing are no longer valid, and new problems arise. One of the fundamental questions that naturally arises is how to process joins over unbounded streams.

In the limit, processing a join over unbounded input streams requires unbounded memory, since every tuple in one infinite stream must be compared with every tuple in the other. Clearly this is not practical. In view of this, we expect that in practice most join queries over unbounded input streams will contain "window predicates" that restrict the number of tuples that must be stored for each stream. For example, in a join of two streams *R* and *S*, we might specify that we are only interested in *R* tuples that have arrived in the last *t1* seconds, and *S* tuples that have arrived in the last *t2* seconds. We call a join with such timing constraints a *moving window join*. Moving window joins are continuously running queries that produce new results as new input tuples arrive.

Figure 1 illustrates a moving window join. There are two input streams, *A* and *B*, each with its respective moving window size. Assuming an arrival rate of λ_a and a window size of T_a for input stream *A* (λ_b and T_b respectively for input stream *B*), the moving window size is $\lambda_a T_a$ ($\lambda_b T_b$, respectively). For the join to correctly compute the answer, it must at all times maintain one window's worth of input tuples for each input stream. Tuples enter and leave this window of stored tuples as time progresses.

To illustrate a simple algorithm for evaluating a moving window join, assume we perform a nested loops join (*NLJ*) over the two windows, *A* and *B*. By this we mean that upon each arrival of a new tuple from stream *A*, three tasks must be performed:

1. Scan stream *B*'s window to find any matching tuples and propagate them to the result.
2. Insert the new tuple into stream *A*'s window. Since we are using *NLJ*, this insertion can be a simple append from the stream's input buffer to the end of the allotted window buffer.

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

T_b	stream B time window size
λ_b	stream B arrival rate
B	number of tuples in window B - $T_b\lambda_b$
|B|	number of hash buckets in window B
B/|B|	number of tuples in a B hash bucket
NKey(B)	number of unique keys in window B
M	number of tuples that fit in memory
C_n	cost of accessing one tuple in NLJ
C_h	cost of accessing one tuple in HJ
σ_b	window B selectivity factor - 1/NKey(B)
σ	join selectivity factor - min(σ_a, σ_b)

Table 1. Definition of terms used in cost model

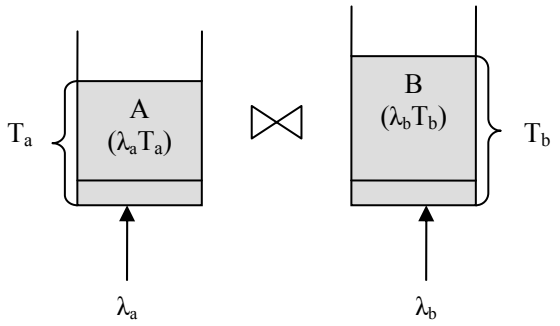

Figure 1. Moving window join

3. Invalidate all expired tuples in stream *A*'s window (this is just those tuples whose timestamp is now outside the current time window.) We can naïvely perform this invalidation by a simple scan of the window buffer, marking as expired all tuples violating the window's constraints.

Even with this simple example, some interesting questions arise:

- How can we measure the efficiency of a moving window join evaluation strategy, since the traditional metric of execution time to completion does not apply?
- Can an algorithm for a moving window join take advantage of asymmetries in the rates of the input streams?
- How can we deal with cases in which an input stream is so fast that the system cannot keep up?
- If memory is the bottleneck, how should we allocate memory between the two windows for the two inputs?

We seek to address these questions in the rest of this paper. Turning to our first question above, we introduce a unit-time-basis cost model to analyze the expected performance of moving window joins evaluation algorithms. Using this cost model, we propose strategies for evaluating window joins in three scenarios.

In the first scenario one stream is much faster than the other. We investigate whether this difference between speeds plays any role in join algorithm performance and, if so, how we can take advantage of this role. We show that this issue can be addressed by further refining our cost model so that it models the contribution to the join evaluation cost by each join input stream separately. Using this refined cost model, we show that, perhaps surprisingly, asymmetric streaming join algorithms can perform better than their symmetric counterparts the symmetric hash join and symmetric nested loops join. (By "asymmetric" we mean that, for example, the join operator might use nested loops for one input stream and hash join for the other.)

In the second scenario we assume that system resources are insufficient to keep up with the speed of the input streams. Such a scenario can arise either when the query to be evaluated contains expensive predicates or the input stream's arrival rate is faster than the join operator's service rate. We address the questions of optimal resource allocation in such a scenario by introducing an analytical model that allows us to maximize the number of result tuples generated by properly allocating computing resources across the two input streams.

In our third and final scenario memory is the constraining resource. In more detail, the problem is the following: given a fixed amount of memory and flexible time windows for the join, how can we adjust the window sizes in a way that the total number of tuples produced is maximized? We show that the proper allocation of memory to the streams in such a case can have a strong effect on the performance of the join algorithm. As we will see, there is a strong correlation between memory allocation techniques, resource utilization and the number of answer tuples generated.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 describes our proposed cost model framework for moving window joins. Section 4 presents techniques for maximizing join efficiency in terms of the resource consumption and/or the number of result tuples produced. Finally, Section 5 gives our conclusions and identifies future work.

2. Related Work

As the Internet computing infrastructure matures, the data access paradigm considered by DBMS researchers is expanding from the traditional disk-oriented paradigm to include network stream-oriented applications. A large and growing body of research exists addressing the new problems that arise in such situations.

One thrust in this body of research addresses problems arising when processing continuous queries [1][2]. The NiagaraCQ system addresses scalability in terms of the number of queries by introducing predicate grouping and group optimization techniques. This system was built in the context of the Niagara Internet query system [3], which proposes a combining XML Internet searching and query processing. Such continuous query systems can

utilize the analytical framework proposed in this paper to extend their domain to include window join queries.

Another relevant research area deals with adaptive query processing [4][5] and query scrambling [6]. In adaptive query processing, the goal is to identify at run time when sub-optimal performance arises because of differences between the estimated and measured selectivity factors in the query. When such a case is detected, the plan is dynamically altered in a way that is believed to enhance the overall performance. In query scrambling, the focus is on identifying and exploiting periods during which some input streams are blocked. Whenever an operator blocks, the execution frameworks pre-empts the operator, allowing other, non-blocked operators to execute. Both research directions are compatible with ours, as they can take advantage of our cost models and asymmetric window join processing algorithms.

Streaming algorithms for join evaluation is another relevant research area. The first such algorithm was the Symmetric Hash Join [7], which was optimised for in-memory performance, leading into thrashing on larger inputs. To rectify the situation XJoin was introduced [8]. Similar techniques are presented in [4] as well. A symmetric nested loops join was proposed in the context of online aggregation [9]. None of this work addressed the issues of performing window joins over unbounded input streams.

Some research exists on single input stream moving window or temporal aggregates [11][12]. Datar et al. presented a technique to maintain moving window statistics [13]. None of these considers moving window joins. The closest work to our problem domain is the Diag-Join proposed by Helmer et al. [10]. Diag-Join was developed for data warehouse environment in which large relations are being joined. It takes advantage of append-heavy nature of warehouse relations by exploiting the fact that most of the warehouse join performed on foreign keys, and matching tuples are likely to be found in the physically close time frame of their creation. This work did not consider streaming output, nor did it consider asymmetric join algorithms or the resource allocation issues we raise here.

A good deal of research has been conducted on the general architecture of stream processing systems. Seshadri et al. developed a sequence data base system, SEQ [17][18]. In [14], Babu and Widom proposed an architecture for a general purpose stream data management system and identified research problems in continuous query processing over streams. Tribeca is an example of special purpose stream database implementations [19]. Other general stream-oriented database architecture work appears in the sensor network domain [20][21]. Examples of this work include Berkeley's Telegraph project [15] and Cornell's Cougar database project [16] is also extended to support sensor data. None of these papers considers window join evaluation.

Finally, Viglas and Naughton proposed a rate based streaming query optimization framework [22]. Integrating the rate based optimization model with our unit time cost model is an interesting area for future research.

3. Estimating the Cost of Moving Window Joins

Conceptually, a join operator must ensure that every tuple in one of its inputs is compared with every tuple in the other. When these input sets are unbounded, as is the case for infinite streams and continuous queries, this of course is problematic – to compare two infinite inputs would require infinite storage. We think that in view of this, most practical joins over streaming inputs will limit the storage required for each input by imposing a *temporal constraint* on each streaming input. That is, instead of saying we want to join all tuples of *R* and *S*, we say we want to join all tuples that have arrived on *R* in the last *t1* seconds with all the tuples that have arrived on *S* in the last *t2* seconds. With the addition of such moving window predicates, even a join over infinite inputs is a bounded memory operation.

A window join query consumes unbounded input streams and produces outputs as long as the input continues to stream in. A traditional, cardinality-based, cost model for an evaluation algorithm is incapable of producing cost estimates in such a scenario since it estimates the time needed for a query to be run to completion, and the algorithm may never complete. Estimating the cost of a continuous window-join query, therefore, requires a new metric; we propose a *unit-time-basis* cost model as such a metric.

Consider the scenario in Figure 1. Each arrival in stream *A*'s window (hereafter we will use simply *A* to refer to this window when this is clear in context) triggers three tasks: checking window *B* for joining tuples, inserting the tuple in window *A* and invalidating any expired tuples from that window. A cost formula for that operation is shown below.

$$C_{A \bowtie B} = \lambda_a(probe(b) + insert(a) + invalidate(a)) + \lambda_b(probe(a) + insert(b) + invalidate(b))$$

The first half of the formula captures the cost of operation for stream *A* arrivals, while the second half does the same for stream *B* arrivals. In the formula, each component (*probe*, *insert*, *invalidate*) is multiplied by the expected number of arrivals per unit time.

Notice that this model captures the invalidation cost. Expired tuples must be invalidated to ensure that the window predicates are correctly evaluated and to avoid wasting memory. In the formula, we assume that *active invalidation* is used, i.e., every time a new tuple is

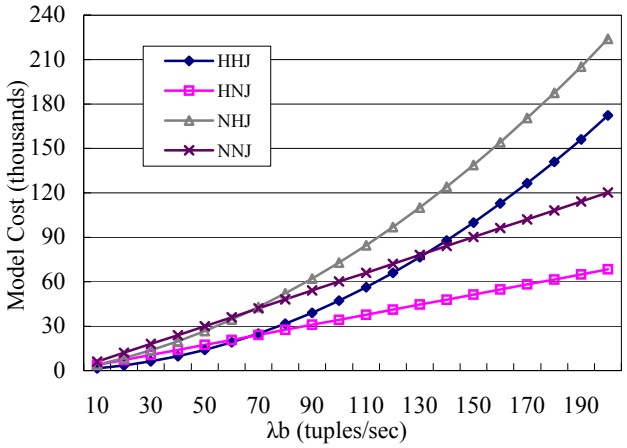

Figure 2. Cost of moving window joins (T_a=60, λ_a=10, T_b=60, σ_a=0.1, σ_b=0.1, C_n=0.5, C_h=0.65)

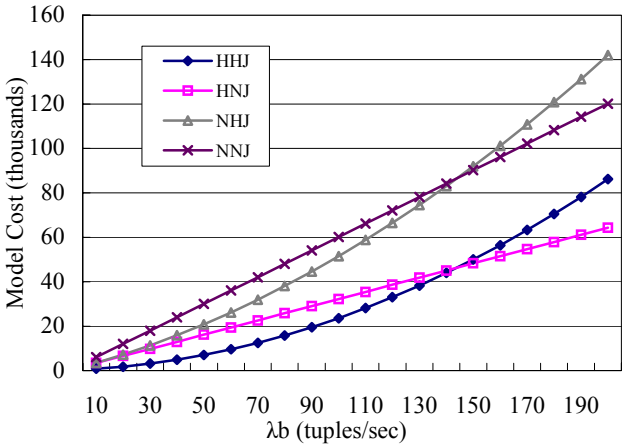

Figure 3. Cost of moving window joins (T_a=60, λ_a=10, T_b=60, σ_a=0.05, σ_b=0.05, C_n=0.5, C_h=0.65)

inserted, expired tuples get checked and removed. Alternatively, a *lazy invalidation* could be used, in which the invalidation would be postponed until the window is probed.

There are space-time tradeoffs involved in the choice of the invalidation scheme. Lazy invalidation may improve the overall join processing cost, but it requires more memory. While lazy vs. active invalidation is an interesting area for future work, it is not central to the contributions of this paper, so for the most part we will use the active invalidation cost model in the remainder of this paper.

Another interesting point is that the cost of a single join operation can be divided into two independent components, one for each input stream. For example, the following is the unit cost of joining A tuples to B tuples, plus the invalidation and insertion cost for tuples into B.

$$C_{A\times B} = \lambda_a(probe(b)) + \lambda_b(insert(b) + invalidate(b)) \qquad (1)$$

This formula captures the aggregate cost of accessing window B in a single time unit. Suppose we perform *NLJ* from A to B, and we estimate the cost based on the number of tuples each algorithm touches. Then the cost of *probe*(b) equals the size of window B (since the whole window must be scanned) and the *insert*(b) and *invalidate*(b) components are both equal to one (since one tuple will be inserted and one tuple invalidated.) Notice that the all three terms are determined without knowing the join algorithm chosen for the B join A direction. We denote the cost of each join direction as $C_{A\times B}$ and $C_{A\times B}$, for joins A to B and B to A, respectively.

3.1 Cost of Nested Loops Join *A* to *B*

The cost formula for a nested loops join from A to B is shown below (the terms used in cost model are described in Table 1):

$$C_{A\times B}(NLJ) = (\lambda_a B + 2\lambda_b)\times C_n = (\lambda_a T_b \lambda_b + 2\lambda_b)\times C_n \qquad (2)$$

The cost of nested loops join is equal to the number of tuples accessed in a time unit during the join operation, multiplied by the cost of accessing a single tuple. The term $\lambda_a B$ represents the number of tuples accessed to search for matches in window B. It is the *NLJ*-specific equivalent of $\lambda_a(probe(b))$ in equation (1). The number of tuples in window B is $T_b\lambda_b$ because in each time unit, λ_b arriving tuples will enter the window and stay there for T_b time units.

The invalidation and insertion costs are straightforward for the *NLJ* case. The insertion cost is the cost of handling one tuple, multiplied by stream B's arrival rate. This is because a tuple insertion requires no extra tuple accesses except for the inserted tuple itself. For invalidation, the average number of expired tuples for each new tuple arrival is similarly equal to one. Although the actual number of invalidated tuples may differ for each time unit, depending on stream B's distribution, with a fixed time window and a constant arrival rate, the average number of invalidated tuples is one. Hence, the cost of insert and invalidation for *NLJ* becomes $2\lambda_b$, multiplied by the single tuple access cost, C_n.

We briefly mentioned earlier that there are tradeoffs between time and space when using lazy invalidation. The cost formula for lazy invalidation is shown below:

$$C_{A\times BLAZY}(NLJ) = \lambda_a(T_b\lambda_b + \lfloor \log_2(\lceil \lambda_b / \lambda_a \rceil) + 1 \rfloor)C_n + \lambda_b C_n$$

The number of expired tuples, piled up between A tuple arrivals, is equal to the product of B's arrival rate and A's inter-arrival time, i.e., λ_b / λ_a. We assume the simple optimization of performing binary search to reduce the invalidation cost.

3.2 Cost of Hash Join A to B

In the case of a hash join *HJ*, the cost of *probe*(*b*) and *invalidate*(*b*) in equation (1) is a function of the hash bucket size in window *B*. A typical probe action requires one key hashing and key comparisons for each tuple in the retrieved bucket. The invalidation task also performs similar actions. The *HJ* cost formula becomes:

$$C_{A\times B}(HJ) = (\lambda_a \frac{T_b\lambda_b}{|B|} + \lambda_b(\frac{T_b\lambda_b}{|B|}+1)) \times C_h \quad (3)$$

As shown in Table 1, $T_b\lambda_b/|B|$ represents the number of tuples in a hash bucket in window *B*. A typical in-memory hash table implementation can ensure the number of buckets is close to the number of unique keys in the window. Throughout the paper, unless specified otherwise, we will use the terms *NKey*(*B*), |*B*|, and 1/ σ_b interchangeably. The following equation is equivalent to the one given above.

$$C_{A\times B}(HJ) = (\lambda_a T_b\lambda_b\sigma_b + \lambda_b T_b\lambda_b\sigma_b + \lambda_b) \times C_h$$

The constant weight factor, C_h, represents the cost of accessing a single tuple in a specific hash table implementation. Later, we will show how to determine these weight factors for both *HJ* and *NLJ*.

The cost formula above reflects active invalidation. For lazy invalidation, invalidation takes place only when a new tuple probes a bucket and it is performed one bucket at a time. As a result, expired tuples in other buckets survive until the next probe hits the bucket. Unlike the *NLJ* case, *HJ* lazy invalidation requires even more extra memory because the invalidation is performed only on one bucket after each probe.

3.3 Cost of Full Joins

In this section, we present three cost formulas including two symmetric join implementations (*HJ*, *NLJ*) and an asymmetric combination of H*J* and *NLJ*. We denote these by *HHJ*, *NNJ*, and *HNJ*, respectively. We named each combination by concatenating the initials of two join algorithms. For instance, *HNJ* means that the left window (*A*) contains hash data structure for hash join and the right window (*B*) contains a data structure for nested loops join. The cost formulas for the three full joins are given below, while the cost formula from *NHJ* is omitted, as it is obvious from *HNJ*.

$$\begin{aligned} C_{A\bowtie B}(HHJ) &= C_{A\times B}(HJ) + C_{A\times B}(HJ) \\ &= (\lambda_a + \lambda_b)(\frac{T_b\lambda_b}{|B|} + \frac{T_a\lambda_a}{|A|} + 1)C_h \end{aligned} \quad (4)$$

$$\begin{aligned} C_{A\bowtie B}(NNJ) &= C_{A\times B}(NLJ) + C_{A\times B}(NLJ) \\ &= \lambda_a(T_b\lambda_b + 2)C_n + \lambda_b(T_a\lambda_a + 2)C_n \end{aligned} \quad (5)$$

$$\begin{aligned} C_{A\bowtie B}(HNJ) &= C_{A\times B}(NLJ) + C_{A\times B}(HJ) \\ &= \lambda_a(T_b\lambda_b C_n + \frac{T_a\lambda_a}{|A|}C_h + C_h) + \lambda_b(\frac{T_a\lambda_a}{|A|}C_h + 2C_n) \end{aligned} \quad (6)$$

Figure 2 and 3 illustrates example cost curves for each join algorithm combination for various assumed input rates. The difference between the two graphs is that Figure 2 is plotted using the join selectivity of 0.1 and Figure 3 is plotted using 0.05. We only show the half of the graph for the case where λ_b is greater than λ_a. The remaining half can be generated from this graph by changing the x-axis label to λ_a and switching the two series, *HNJ* and *NHJ*. Other than that, everything is identical to the presented case.

As expected, the *HJ* outperforms every other combinations in the beginning where the input streams' speed difference is minimal. However, as the speed gap between the two gets wider, the cost of *HJ* increases faster than that of *HNJ* and finally exceeds that of *HNJ* at around 70 tuples/sec (Figure 2) and 140 tuples/sec (Figure 3) in each graphs. The performance crossover point in Figure 3 is about twice as that in Figure 2, which is equal to the ratio of *Nkey* values used in each graph. In the following section, we will discuss issues regarding how to estimate weight factors, C_n and C_h, and how to determine the crossover points.

3.4 Estimating the Weight Factors

So far, we have been using C_n and C_h to represent implementation effects and/or system dependent costs. In this section, we focus on estimating the ratio between them rather than their absolute values, because that is what is required for the cost model. In the following we illustrate measuring this ratio in our implementation.

We implemented a moving window hash join and a moving window nested loops join algorithm, and a moving window join operator that can accommodate asymmetric combinations of the two. The operators were implemented in Java and run on Sun Microsystems' Java HotSpot Client VM 1.3.1. Experiments were performed on an Intel PIII 600Mhz machine with 128MB of memory, running Windows 2000.

To validate our cost model, we first need to find the correct weight factors. Figure 4 illustrates system time vs. cost model curves for one-way joins from *A* to *B*. CPU time is measured by processing 60 second's worth of tuples without intermittent delays. For instance, to measure the CPU time for λ_b equal to 100 tuples/sec, we process 6000 tuples in one batch and measure the total running time. We chose this way instead of measuring individual tuple handling costs for two reasons. First, in this way, we can measure the CPU time even for an input load that exceeds the system capacity. For instance, at the arrival rate 120 tuples/sec, *HJ* crosses the 60 seconds line. This means that the *HJ* requires full computing power of

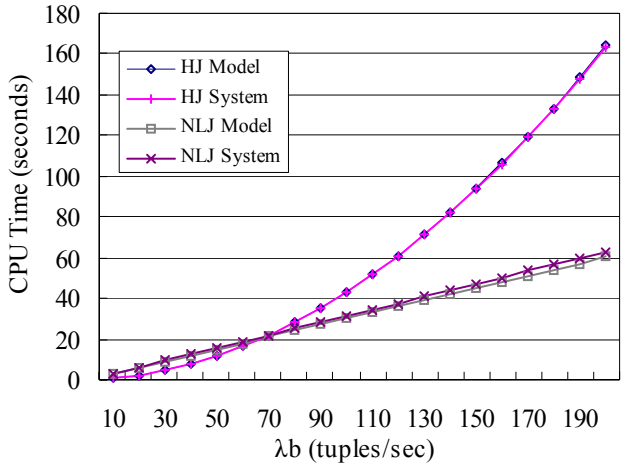

Figure 4. System Time vs. Model Cost for $A \bowtie B$ (T_b=60, λ_a=10, σ_b=0.1, C_n=0.5, C_h=0.65)

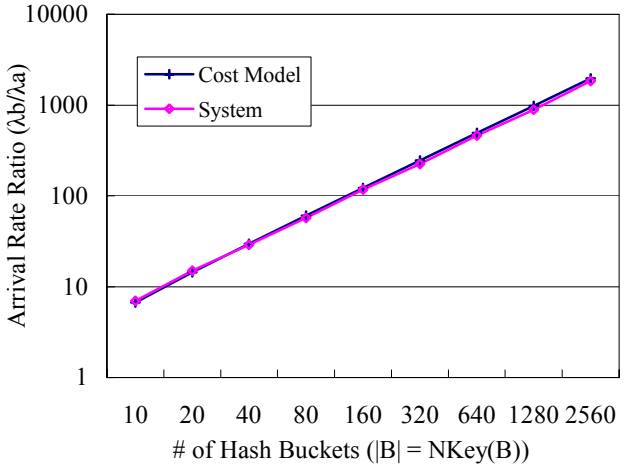

Figure 5. Cross-over points (λ_b/λ_a) where NLJ starts to outperform HJ ($C_{A \bowtie B}$(HJ) > $C_{A \bowtie B}$(NLJ))

the system to process the input rate, λ_b, of 120 tuples/sec and above. Second, processing single tuples takes too little time to be measured accurately given the granularity of our system timers.

To plot the graph in Figure 4, we chose 0.5 and 0.65 for C_n and C_h, but what is important here is not the actual values for each, but the ratio between the two. The weight factor ratio between the two implementations, *HJ* and *NLJ*, is 1.3 in our implementation. The values 0.5 and 0.65 are chosen only for the purpose of aligning the model cost with the actual CPU time.

Notice that both system time and model cost curves have a crossover point where *NLJ* starts to outperform *HJ*. The turnover point was close to 70 tuples/sec for arrival rate *B*. In other words, the performance crossover happens roughly when input stream *B* becomes more than 7 times faster than stream *A*.

It is much more important to ensure that the cost model accurately predicts the crossover points than the actual system time. That is because the point of estimating cost is to compare alternative algorithms, and to choose the right algorithm for a given scenario.

The crossover points can be calculated by equating the two cost formulas $C_{A \times B}(HJ)$ and $C_{A \times B}(NLJ)$.

$$C_{A \times B}(HJ) = C_{A \times B}(NLJ)$$

$$\Leftrightarrow (\lambda_a(\frac{B}{|B|}) + \lambda_b(\frac{B}{|B|}+1)) \times C_h = (\lambda_a B + 2\lambda_b) \times C_n$$

$$\Leftrightarrow \frac{\lambda_b}{\lambda_a} = \frac{BC_n - \frac{B}{|B|}C_h}{\frac{B}{|B|}C_h + C_h - 2C_n} \approx \frac{BC_n - \frac{B}{|B|}C_h}{\frac{B}{|B|}C_h} \qquad (7)$$

$$\Leftrightarrow \frac{\lambda_b}{\lambda_a} \approx \frac{|B| - C_h/C_n}{C_h/C_n}$$

Constant terms are ignored in the denominator of the third equation, as their effects are small when compared to the other terms; this makes the formula simpler and more intuitive. The ratio C_h/C_n represents the tuple access overhead of a hash join when compared to that of a nested loops join.

The overhead ratio can differ from system to system and from implementation to implementation. To measure it, we ran a simple test on our particular implementation. We compared actual CPU time of each join operation with a varying input arrival rate for *B,* and then found the system's crossover point. Next, we replaced the arrival rates in the last equation with the measured values and solved the equation to get the overhead ratio. In our system, the overhead ratio was equal to 1.3.

Figure 5 illustrates two curves representing the measured and predicted crossover points. We plot the graph with varying |*B*|, which happens to be the only variable that actually determines the crossover points, as illustrated in equation (7) above. It shows that the cost model with an overhead ratio of 1.3 closely approximates the crossover points measured using the real system implementation.

To illustrate the implication of this, suppose we are joining two streams, *A* and *B*, with rates of 10 tuples/sec and 100 tuples/sec respectively and suppose there are 10 unique keys in stream *B*. We assume the number of hash buckets is roughly equivalent to the number of unique keys in the target stream. Given this, we can tell using the cost model that *NLJ* will outperform *HJ* for join direction *A* to *B*. since the turnover point is calculated to be approximately equal to 6.7 and the ratio between the input stream arrival rates (λ_b/λ_a =10) is greater than the calculated crossover point.

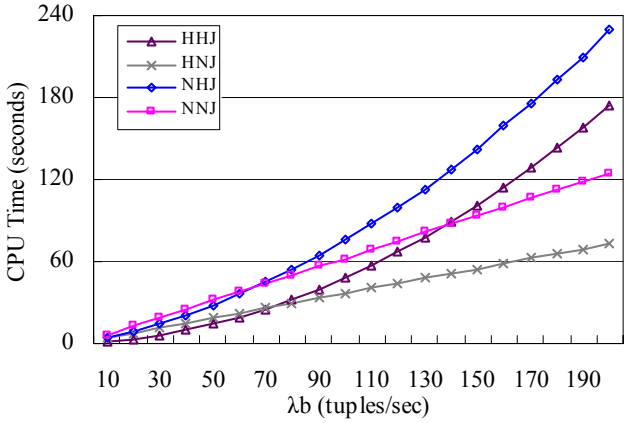

Figure 6. CPU Time of four full joins (T_a=60, T_b=60, λ_a=10, σ_a=0.1, σ_b=0.1)

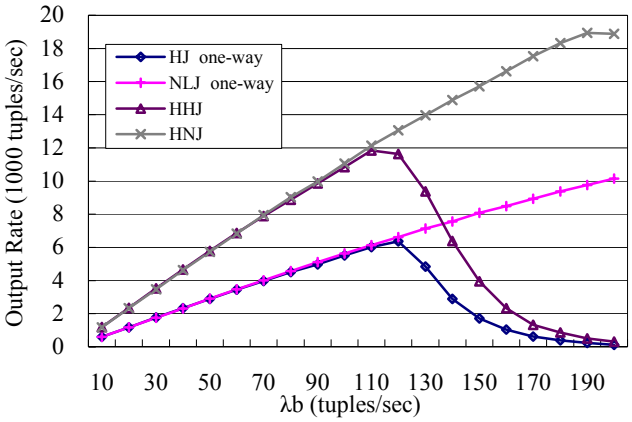

Figure 7. Join output rates with varying λ_b (T_a=60, T_b=60, λ_a=10, σ_a=0.1, σ_b=0.1)

4. On Maximizing the Efficiency of Processing Joins

In this section, we investigate strategies for maximizing the efficiency of processing moving window joins in three scenarios: (i) one stream is much faster than the other, (ii) computing resources are insufficient to keep up with the speed of the input streams, and (iii) memory resources are limited.

We divided the problem of maximizing join efficiency into four categories of sub-problems:

- Constant time window and Constant arrival rate (CTCA)
- Constant time window and Variable arrival rate (CTVA)
- Variable time window and Constant arrival rate (VTCA)
- Variable time window and Variable arrival rate (VTVA)

We consider CTCA in Section 4.1, CTVA in Section 4.2, and both VTCA and VTVA in Section 4.3.

4.1 Exploiting Asymmetry in Input Streams Speed

In this section, we consider the case where the two time windows are fixed and the aggregate speed of two streams is less than the system's service rate μ (i.e., $\lambda_a + \lambda_b < \mu$). We evaluated both *NLJ* and *HJ* in this context. *NLJ* is potentially the worst join algorithm and *HJ* is often the best join algorithm in traditional DBMS settings. We show, however, that with asymmetry in the arrival rates of input streams, the asymmetric combination of *HJ* and *NLJ* can outperform both previously proposed symmetric *HJ* and symmetric *NLJ*.

From the equation (7) in Section 3.4, we get an *algorithm determinant,* shown below.

$$\frac{\lambda_b}{\lambda_a} > \frac{|B| - C_h/C_n}{C_h/C_n} \quad (8)$$

For given parameters, we can predict the likely winner using this formula. If this inequality holds, we can predict *NLJ* will outperform *HJ*; otherwise, *HJ* will outperform *NLJ*. By choosing the cheaper algorithm, we can accommodate faster input rates without thrashing and in turn generate more results.

Figure 6 illustrates the CPU time curves for four full join combinations. For the same reasons discussed in connection with Figure 4, we measured CPU time by processing 60 second's worth of tuples without intermittent delays. The graph shows that *HHJ* outperforms any other algorithm until the input rate reaches about 70 tuples/sec; then, *HNJ* takes over for the rest of the graph. In other words, either *HHJ* or *HNJ* is the winner over other combinations for join direction A to B, in the range where stream *B* is faster than or equal to the speed of stream *A*. For the sake of presentation, we ignored the losing combinations, *NHJ* and *NNJ*, in Figure 7.

Figure 7 illustrates the output rates of two one-way joins, *HJ* and *NLJ*, and two full joins, *HHJ* and *HNJ*. One-way *HJ* reaches its thrashing point at 120 tuples/sec and symmetric *HJ* reaches the point a little earlier, at 110 tuples/sec. This seems reasonable since the cost of a full join should be greater than that of a one-way join. Both hash join output rates decrease drastically after each passes its thrashing point. On the other hand, *HNJ* output rates continue to increase until λ_b hits 190 tuples/sec. It produces 18939 tuples per second at its peak right before entering the thrashing point. Meanwhile, *HHJ* produces 11844 tuples per second at its peak, when λ_b hits 110 tuples/sec. Interestingly, both one-way *HJ*'s and *HHJ*'s thrashing points are closely predicted in the CPU time curve shown in Figure 4 and Figure 6, respectively.

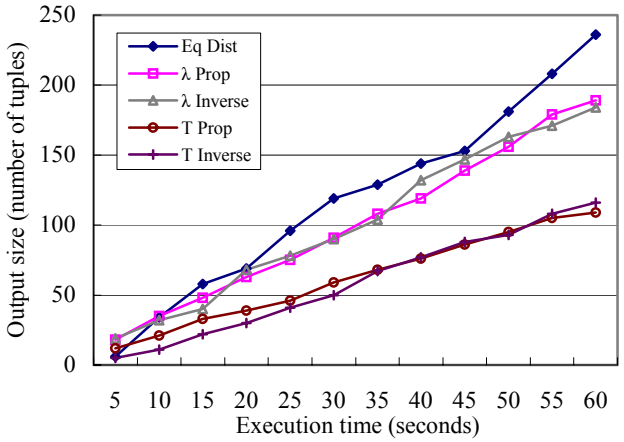

Figure 8. Performance of computing resource allocation strategies (λ_a=80, λ_b=20, T_a=9, T_b=1, σ_a=0.02, σ_b=0.08, μ=10)

For *HNJ*, however, the prediction is less accurate than for *HHJ*. The CPU time curve of *HNJ* crosses the boundary between 160 and 170 tuples/sec, but the output rate graph in Figure 7 shows thrashing occurring at around 190 tuples/sec. This discrepancy is largely due implementation details. The stream feeder we implemented emulates the streams' arrival rate by putting inter-arrival delays (1/arrival rate) between transmissions. The actual inter-arrival time that join operator observes, however, is slightly longer than that, because the stream feeder's stream handling overhead is added to each tuple delivery. As a consequence, the join operator processes the streams at a slightly slower speed than intended. On the other hand, the CPU time measurement did not rely on the stream feeder. Instead, it generated the exact number of tuples for each arrival rate. As a result, it processes more tuples than the actual join operator does for each arrival rate.

Moreover, it is not the case that the prediction for *HHJ* is correct and the one for *HNJ* prediction is not, because the *HHJ's* boundary crossing range, which is between 110 and 120 tuples/sec, is roughly equivalent to the *HNJ* range of 160 to 190 tuples/sec in terms of CPU time, due to the differences in steepness of two curves.

The graph in Figure 7 did not show much of the data after the *HNJ*'s thrashing point, but the experiment result indicates that the thrashing effect is not as severe as in the *HHJ* case; its performance degradation was much more graceful than *HHJ*'s one. We can explain this in the cost formula as well. The increase of *B*'s arrival rate hurts the overall cost of *HJ* more than the cost of *NLJ*. That is because *HJ*'s invalidation task is much heavier than that of *NLJ*, and invalidation occurs after each arrival of a *B* tuple.

Both join algorithms produce almost the same number of result tuples per unit time, until *HHJ* reaches its thrashing point. However, the CPU time or actual cost for running the algorithm is different, even during the time where both algorithms produce almost the same number of results. The effect of the higher cost of HHJ is unobservable in Figure 7, until it reaches a thrashing point. It is hidden while system is under-loaded but as the load increases close to the thrashing points, the cost difference can affects overall system throughput significantly. The cost difference is clearly illustrated in Figure 4 and Figure 6.

The performance difference of the two algorithms can be explained by comparing the two cost formulas given in Section 3. Invalidation is relatively costly in *HJ*, and that makes it less favorable when invalidation happens high frequencies compared to probe operation, which is the case where $\lambda_b > \lambda_a$.

4.2 Maximizing the Number of Result Tuples with Limited Computing Resources

In this section we investigate the case where computing resources are insufficient to keep up with the input streams. As briefly introduced in Section 1, this scenario can arise at least in two cases: 1) when system evaluates very expensive predicates 2) or the input stream's speed is faster than the join operator's service rate. Both cases require a careful allocation of computing resources across the input streams.

Note that since the input rate exceeds the service rate, in such a scenario we cannot generate all answer tuples (the system must drop some or fall hopelessly behind.) Thus the question that naturally arises is how to regulate the input streams in order to maximize the number of result tuples. Here we "regulate the streams" by dropping some of their constituent tuples, so that the system sees an effective input rate that it can support. But should we drop tuples from input streams in proportion to input stream rates? Should we do so proportionally to the size of each time window?

Interestingly, the answer is neither of the above. Figure 8 shows performance curves for five different stream regulation strategies. The first strategy regulates the streams proportionally to the original streams' speeds. In this experiment, we assume 80 and 20 tuples/sec for *A*'s and *B*'s arrival rates, respectively. With this resource allocation the regulated streams' arrival rate becomes 8 and 2 tuples/sec for *A* and *B* since the stream processor's service rate, μ, is set to 10 tuples/sec, and the sum of two regulated streams' speed must match μ. The second strategy is to allocate resources inversely proportional to the speed of the original streams. Hence, we effectively see 2 and 8 tuples/sec for *A* and *B*, respectively. We also considered allocations proportional or inverse proportional to the two time window sizes. Finally, we added the equal distribution strategy.

As shown in Figure 8, the winner is the equal distribution strategy. We distributed computing resources equally across the two streams so that the regulated speed

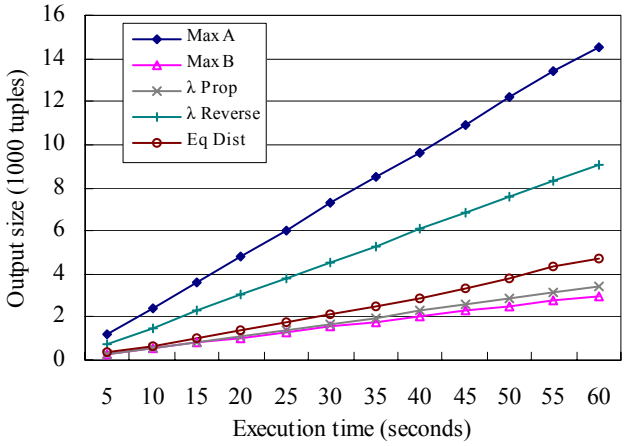

Figure 9. Performance of memory allocation strategies w/ fixed arrival rates (λ_a=10, λ_b=50, M=1000, σ_a=0.005, σ_b=0.01)

becomes 5 and 5 tuples/second. Note that the proportional and inverse proportional strategies did not show much difference in performance.

We begin our explanation of this effect by introducing a moving window join output rate equation. In the following equation, the selectivity factors of window *A* and *B* are denoted as σ_a and σ_b, respectively. To approximate join selectivity, we take the smaller value between the two and denote it as σ.

$$r_o = \min(\sigma_a, \sigma_b)(\lambda_a \lambda_b T_b + \lambda_b \lambda_a T_a) = \sigma \lambda_a \lambda_b (T_a + T_b) \qquad (9)$$

In this scenario, however, unlike the previous case in Section 4.1, the two incoming streams' arrival rates become variables. The join output rate in this scenario becomes the following:

$$r_o = \sigma(\lambda_{a'} T_a)\lambda_{b'} + \sigma(\lambda_{b'} T_b)\lambda_{a'} = \sigma \lambda_{a'} \lambda_{b'} (T_a + T_b)$$
$$\text{where } \lambda_{a'} + \lambda_{b'} = \mu,\ \lambda_{a'} \le \lambda_a,\ \lambda_{b'} \le \lambda_b \qquad (10)$$

Given that the two time window sizes are fixed in the query, we need to maximize the product of two regulated streams' arrival rates, $\lambda_{a'}\lambda_{b'}$, in order to maximize the output rate r_o. We will investigate the case where the two time window sizes become variables in the following section.

By taking the first derivation of the product, we find the ideal allocation ratio for maximizing the output size.

$$f = \lambda_{a'}\lambda_{b'} = \lambda_{a'}(\mu - \lambda_{a'}) = -\lambda_{a'}^2 + \mu\lambda_{a'}$$
$$\frac{\delta f}{\delta \lambda_{a'}} = -2\lambda_{a'} + \mu = 0 \qquad (11)$$
$$\therefore \lambda_{a'} = \mu/2$$

As shown above, the ideal ratio for regulated streams' rates is equal to one. In other words, we get the best results, by allocating half of the resources to each stream, regardless of time window sizes and window selectivity factors. In case one input stream's input rate, say λ_a, is less than the ideal rate, $\mu/2$, the best we can do is to allow λ_a flowing in at its original rate and allocate $\mu - \lambda_a$ to $\lambda_{b'}$.

On determining a join algorithm, the ratio of regulated streams' rates must be taken into account instead of the streams' original rates. In the ideal case where equal distribution is possible, the join algorithm selection becomes straightforward. The ratio $\lambda_{b'}/\lambda_{a'}$ equals one, and it suggests that the hash join needs to be used for both directions, unless the *NKey* value for each window is unusually small.

For instance, if we have two streams with λ_a = 5 and λ_b = 100 tuples/sec, and the system service rate is 50 tuples/sec, the regulated rates become $\lambda_{a'} = 5$, $\lambda_{b'} = 45$. For the case where $|B|$ = 10 or σ_b = 0.1, the algorithm determiner will suggest *NLJ* for join direction *A* to *B*. On the other hand, for $|B|$ = 20 or σ_b = 0.05, *HJ* becomes the algorithm of choice.

4.3 Maximizing the Number of Result Tuples with Limited Memory

In this subsection, we investigate strategies for maximizing the number of results in the case where memory resources are limited. We assume that the two time window size can be adjusted to fully utilize available memory. That is, rather than viewing the time windows as being specified in the query, we regard them as tunable parameters controlled by the system in order to maximize performance. In the previous two subsections, we considered cases where the time window size is fixed and memory was sufficient to hold both moving windows. The problem we address here has one more degree of freedom, choosing the two time window sizes, and this increases the complexity of problem significantly.

Case of variable time window / constant arrival rate

In this scenario, we investigate memory allocation strategies for maximizing the size of query result. To motivate the discussion, we present an experiment result that shows the performance of five sample memory allocation strategies. Figure 9 illustrates the test result.

The first strategy, *Max A*, represents a strategy that allocates all available memory to the slower stream. On the other hand, the second strategy, *Max B*, allocates all memory to the faster stream. The third strategy, *λ Prop*, allocates memory proportional to the speed of the two input streams. The fourth strategy, *λ Inverse*, allocates memory inverse proportionally to the two input streams' speed. Lastly, *Eq Dist*, distributes equal amounts of memory resources to both input streams.

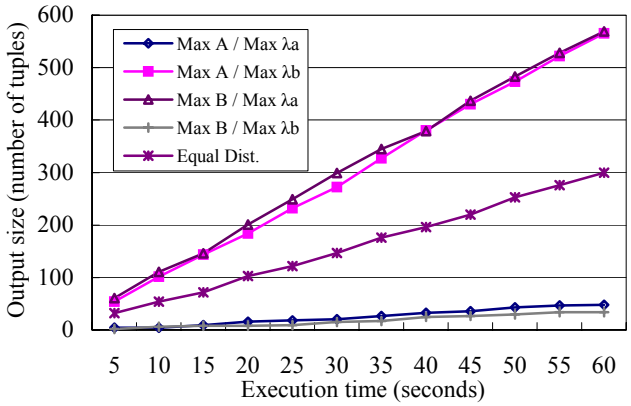

Figure 10. Performance of memory allocation strategies w/ variable arrival rates (μ=10, M=100, σ=0.01)

As shown in Figure 9, the best performance is obtained by *Max A* and the worst by its opposite strategy *Max B*. Now the question to answer is whether *Max A*, which allocates all memory to the slower stream, is the winner in all cases or if this is just an artifact of the specific parameters used in our experiment.

To address the issue, we introduce the following equations. The first equation below represents the relations between the available memory and two adjustable time window sizes.

$$M = A + B = \lambda_a T_{a'} + \lambda_b T_{b'} \quad \text{where } T_{a'} \le T_a,\ T_{b'} \le T_b \tag{12}$$

We can rewrite $T_{b'}$ as $(M - \lambda_a T_{a'})/\lambda_b$ and by replacing T_b in equation (9) with this one, we get:

$$r_o = \sigma\lambda_a\lambda_b(T_{a'} + (M - \lambda_a T_{a'})/\lambda_b) = \sigma(\lambda_b - \lambda_a)\lambda_a T_{a'} + \sigma\lambda_a M \tag{13}$$

Now, $T_{a'}$ becomes the only variable in this equation, and we can see that to maximize the output rate, $T_{a'}$ must be set to either its maximum or its minimum within the range, depending on the sign of coefficient, which is determined by $(\lambda_b - \lambda_a)$.

$T_{a'}$ can take any value in $[0, M/\lambda_a]$. It must be set to its max value, M/λ_a (i.e. $T_b = 0$), when the input stream *B* is faster than the input stream *A*, otherwise it must be set to zero. Simply put, if λ_b is greater than λ_a, the input stream *A* should take up all memory available and the input stream *B* should just probe against *A* and pass by without *build* or *invalidation*.

Interestingly, this means that the cost of the join in this class becomes one of the two one-way joins depending on the relative speed of the input streams. In other words, the cost becomes $C_{A\ltimes B}$ in case the input stream *B* is faster than the *A*, and $C_{A\rtimes B}$ otherwise.

Intuitively, we can see that it would be beneficial to keep the slower stream in memory and let the faster one just probe against it and pass by. At the other end of the spectrum, we can think of an opposite strategy that allocates all memory to the faster one and lets the slower one probe it, i.e., *Max B*, in our experiment. It is straightforward that the first scenario is going to outperform second one, because the number of probe is greater in the first case while the size of the target window being probed is identical, and equal to the memory size, for both cases. Other options in the middle of the spectrum are expected to be worse than the first case and better than the opposite, as is illustrated in Figure 9.

In the case where both input streams' speeds are equal, the value of $T_{a'}$ and $T_{b'}$ becomes irrelevant to the output rate. In this case the output rate becomes a constant, equal to $\sigma\lambda_a M$.

Meanwhile, the join algorithm selection should be performed depending on the decision made here for memory allocation. For instance, suppose the input stream *A* is faster than *B*. In this case, the join process becomes an *A* to *B* one-way join, and the *algorithm determinant* in equation (8) fails given that the |*B*| is in reasonable range, indicating that the *HJ* must be used. In other words, when considering join from a faster to slower stream it would be safe to assume that *HJ* always should be the choice of algorithm. Hence the final cost of join becomes $C_{A\rtimes B}(HJ)$.

Case of variable time window / variable arrival rate

Unlike the previous case in Subsection 4.2, we assume here the two time window sizes can be adjusted to fully utilize given limited chunk of memory. Hence we have four free variables in the equation (10) shown in Section 4.2. We revisit the equation here for reference.

$$r_o = \sigma(\lambda_{a'}T_{a'})\lambda_{b'} + \sigma(\lambda_{b'}T_{b'})\lambda_{a'} = \sigma A\lambda_{b'} + \sigma B\lambda_{a'}$$
$$\text{where } \lambda_{a'} + \lambda_{b'} = \mu,\ \lambda_{a'} \le \lambda_a,\ \lambda_{b'} \le \lambda_b,$$
$$\lambda_{a'}T_{a'} + \lambda_{b'}T_{b'} = A + B = M,\ T_{a'} \le T_a,\ T_{b'} \le T_b$$

In above equation, we have one more constraint that describes the relations between the four free variables and the memory size (i.e. number of total tuples that fits in memory). To maximize the output rate, r_o, we need to find the winning combination of four free variables. To make the task little simpler, we reduce the number of variables into two using the two constraints given in the equation above.

The arrival rates $\lambda_{a'}$ and $\lambda_{b'}$ can be rewritten in terms of μ by applying distribution factor x, which is a fraction between 0 and 1. The window size *B* is also represented in terms of *A* and *M*.

$$\lambda_{a'} = x\mu,\ \ \lambda_{b'} = (1-x)\mu,\ \ B = M - A$$

By replacing the variables in output rate equation, we get:

$$r_o = \sigma(x\mu(M-A)+(1-x)\mu A) \\ = \sigma(x\mu M-(2x-1)\mu A) \quad (14)$$

Now the output rate becomes a function of the two variables x and A. Before we start analyzing the above equation, we present an experiment result on the performance of various resource allocation strategies. Figure 10 shows the test result. It evaluates five different strategies: (i) *Max A / Max* λ_a which maximizes both window A and stream A's arrival rate, similarly, (ii) *Max A / Max* λ_b, (iii) *Max B / Max* λ_a, (iv) *Max B / Max* λ_b, and (v) *Equal Distribution*.

In the experiment, the best performing group is the combinations of maximizing the time window size in one window and maximizing the arrival rate in the other window. These include *Max A / Max* λ_b, and *Max B / Max* λ_a. The next higher performer, *Equal Dist*, is a strategy that distributes equal amount of resources to each stream. The worst performer was a group of combination that maxed out the time window size and arrival rate of the same window.

Figure 11 is a 3-dimensional depiction of the output rate given in the equation (14). The graph shows there are two maxima and two minima as we observed in the experiment in Figure 10. The valid range for distribution factor, x, is (0, 1), because either 0 or 1 forces one of the two streams completely stop. On the other hand, the valid range for A is [0, M]. Either 0 or M means to perform one-way join, which is similar case to what we saw in the VTCA scenario.

In summary, to maximize the output rate of the VTVA, we can either maximize stream A's time window in conjunction with maximizing B's arrival rate, or we can maximize B's time window and A's arrival rate alternatively.

We can understand the situation by examining equation (14). The output rate is a function of two variable, x and A. In case x > ½, the second term in the subtraction, $(2x-1)\mu A$, becomes greater than zero. To maximize r_o, this term must be minimized, thus indicating to minimize A. If A gets minimized to zero, the term $\sigma x\mu M$ remains. To maximize it, we need to take the maximum x value, then r_o converges to $\sigma\mu M$.

As we can see in Figure 11, $\sigma\mu M$ is the maximum possible output rate we can get. It is intuitive to see because μ is the maximum number of tuples that join will process in a time unit. It would be beneficial to use all μ tuples to probe against entire memory rather than sharing memory resources between the two windows, because the memory sharing effectively limits the target table size for each tuple being probed.

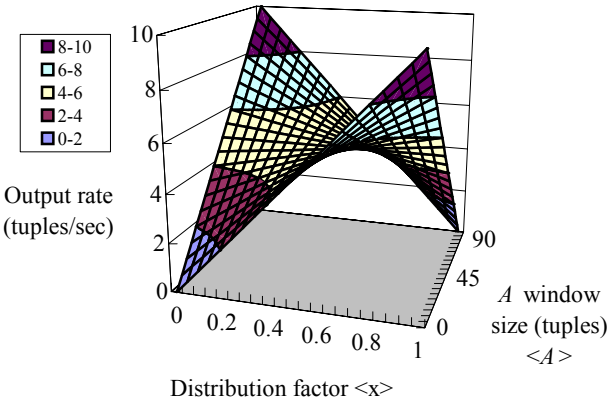

Figure 11. Performance of memory allocation strategies with variable time window / variable arrival rate (μ=10, M=100, σ=0.01)

5. Conclusion

In this paper we investigated strategies for evaluating moving window joins over pairs of unbounded streams. We introduced a unit-time basis cost model to analyze the expected performance of these strategies. One of the notable aspects of the proposed cost model is that it divides the join cost into two independent terms, each corresponding to one of the two join directions. This property allows it to estimate the cost of each join direction separately.

A good deal of room for future work exists. One interesting extension of our work would be to extend the cost model beyond single joins to full query plans. Another potentially interesting direction would be to incorporate the findings in this paper into the previously proposed adaptive query optimization frameworks, so as to extend that work to handle moving window joins. Finally, it would be interesting to model and evaluate other algorithms besides the *NLJ* and *HJ* algorithms.

Bibliography

[1] D. Terry, D. Goldberg, D. Nichols, and B. Oki: Continuous Queries over Append-Only Databases. SIGMOD 1992: 321-330

[2] J.Chen, D. J. DeWitt, F. Tian and Y. Wang: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD 2000:379-390

[3] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, R. Chen: The Niagara Internet Query System. IEEE Data Engineering Bulletin 24(2): 27-33 (2001)

[4] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, D. S. Weld: An Adaptive Query Execution System for Data Integration. SIGMOD Conference 1999: 299-310

[5] R. Avnur, J. M. Hellerstein: Eddies: Continuously Adaptive Query Processing. SIGMOD Conference 2000: 261-272

[6] T. Urhan, M. J. Franklin, L. Amsaleg: Cost Based Query Scrambling for Initial Delays. SIGMOD Conference 1998: 130-141
[7] A. N. Wilschut, P. M. G. Apers: Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991: 68-77
[8] T. Urhan, M. J. Franklin: XJoin: A Reactively-Scheduled Pipelined Join Operator. IEEE Data Engineering Bulletin 23(2): 27-33 (2000)
[9] P. J. Haas, J. M. Hellerstein: Ripple Joins for Online Aggregation. SIGMOD Conference 1999: 287-298
[10] S. Helmer, T. Westmann, G. Moerkotte: Diag-Join: An Opportunistic Join Algorithm for 1:N Relationships. VLDB 1998: 98-109
[11] J. Yang, J. Widom: Incremental Computation and Maintenance of Temporal Aggregates. ICDE 2001: 51-60
[12] I. Motakis, C. Zaniolo: Temporal Aggregation in Active Database Rules. SIGMOD Conference 1997: 440-451
[13] M. Datar, A. Gionis, P. Indyk, R. Motwani: Maintaining Stream Statistics over Sliding Windows, 2002 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)
[14] S. Babu, J. Widom: Continuous Queries over Data Streams. SIGMOD Record 30(3): 109-120 (2001)
[15] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, M. A. Shah: Adaptive Query Processing: Technology in Evolution. IEEE Data Engineering Bulletin 23(2): 7-18 (2000)
[16] P. Bonnet, J. Gehrke, P. Seshadri: Towards Sensor Database Systems. Mobile Data Management 2001: 3-14
[17] P. Seshadri, M. Livny, R. Ramakrishnan: Sequence Query Processing. SIGMOD Conference 1994: 430-441
[18] P. Seshadri, M. Livny, R. Ramakrishnan: The Design and Implementation of a Sequence Database System. VLDB 1996: 99-110
[19] M. Sullivan, A. Heybey: Tribeca: A system for managing large databases of network traffic. In Proceedings of the USENIX Annual Technical Conference, New Orleans, LA, June 1998
[20] D. Estrin, R. Govindan, J. S. Heidemann, S. Kumar: Next Century Challenges: Scalable Coordination in Sensor Networks. MOBICOM 1999: 263-270
[21] J. M. Kahn, R. H. Katz, K. S. J. Pister: Next Century Challenges: Mobile Networking for "Smart Dust". MOBICOM 1999: 271-278
[22] S. Viglas, J. F. Naughton: Rate-Based Optimization for Streaming Information Sources. SIGMOD Conference 2002.