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 sliding 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 asymmetric combinations of join algorithms, (e.g., hash join on one input, nested-loops join on the other) can outperform symmetric join algorithm implementations. 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 input streams. Finally, we investigate strategies for maximizing the number of result tuples produced when memory is limited, 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 In such environments, many assumptions made in traditional query processing are no longer valid, and new problems arise.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 arise 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. The purpose of this paper is to investigate the problems that arise when dealing with window join predicates and present possible solutions.

A window join takes as input two streams of tuples, say Stream *A* and Stream *B*, along with window sizes for both Stream *A* and Stream *B*, as shown in Figure 1. The output is also a stream of tuples, consisting of all pairs of tuples (*a*,*b*), where *a* is from Stream *A*, *b* is from Stream *B*, such that (i) *a* and *b* satisfy the join predicate, and (ii) *a* was in the active window for Stream *A* at the same time that *b* was in the active window for Stream *B*.

Sliding window joins arise in a number of applications. One class of applications deals with correlating information from different sources about the same entities. For example, we may wish to correlate stock price movements with news stories suspected of influencing the price. Or, in a surveillance application, we may want to correlate cell phone traffic with email traffic.

Another class of applications deals with tracking entities through a network of sensors. In this sort of application, each sensor produces a stream recording the entities as they pass the sensor; the "join" of two sensors' streams records traffic between the sensors. Examples of this sort of application include tracking network packets through routers, or generating "click stream" information about visits to multiple web sites, or even monitoring the progress of cars through tollbooths on the highway.

In some applications, the "exact" window join is required. For example, if one is interested in tracking the movements of specific entities, it is probably unacceptable for the join to "drop" answer tuples. However, there are other applications for which an approximate answer might suffice. As an example of this kind of application, consider measuring the delay in traffic between two sensor nodes. In this case it may be acceptable to compute an average value by looking at a subset of the complete result. Indeed, if the system does not have sufficient resources to produce the complete result in a timely fashion, such an approximate but up-to-date average may be much more desirable than a delayed exact result.

Assuming a sliding window join between streams *A* and *B* and a new arrival from stream *A*, then a summary of operations the query processor needs to carry out when evaluating the join is the following:

1. Scan stream *B*’s window to find any matching tuples and propagate the result.
2. Insert the new tuple into stream *A*’s window.
3. Invalidate all expired tuples in stream *A*’s window.

Though the steps seem simple enough, it turns out that their implementation can become complicated due to a mixture of traditional join processing problems and additional issues introduced by having to evaluate the join using a sliding window over unbounded streams. Questions that can make these problems evident are:

1. Given the collection of existing join algorithms, how can they be applied to the problem at hand? None of the

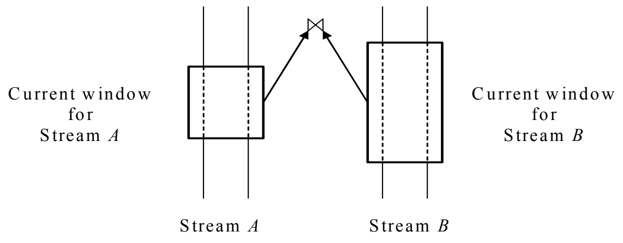

Figure 1. A window join scenario

previously published join algorithms has addressed the issue of invalidating parts of the input as time progresses.

2. How can an optimizer decide which algorithm to use? The traditional metric of execution time to completion does not apply in a sliding window join scenario, since the inputs are infinite.
3. The various input streams may have very different rates. Can a possible asymmetry in those rates be taken advantage of when choosing an evaluation algorithm? For instance, if one of the streams is much slower than the other, it may be possible to assign fewer resources to handle its inputs since they will not appear as frequently.
4. Network links are able to shift data around at very high speeds. What happens if one of the inputs is so fast that the query processor cannot keep up with it? In such a scenario, and depending on the query semantics, it may be acceptable for the query processor to "drop" inputs so it is able to catch up with the streams and resort to approximate answers.
5. If the query processor has limited computational and/or memory resources how should these resources be distributed among the streams? For instance, given a memory budget that is less than the total amount of memory needed to keep both windows in memory, how much memory should be allocated for each window?

Our contributions towards answering all these questions and solving the problems they introduce are the following:

- We classify window join scenarios on the basis of the limitations, if any, of the query processor. The limitations of the query processor can be either on its computational or on its memory resources. By having such a classification, we are able to focus on the important questions of each individual class.
- Assuming that the query processor does not have any serious limitations on its computational and memory resources, the problem is mainly that of deciding on an efficient join evaluation algorithm. Since the traditional cardinality-based cost metric is not applicable, we present a unit-time basis cost model that focuses on the cost of handling a single individual input tuple of each input stream separately. Using this refined cost model, we show that, perhaps surprisingly, asymmetric streaming join algorithms can perform better than their symmetric counterparts. (By "asymmetric" we mean that, for example, the join operator might use nested loops for one input stream and hash join for the other.)
- If the query processor has insufficient memory or computational resources, the focus shifts from cost estimation to resource allocation. In that respect, we present an analytical model that allows us to accurately estimate how computational and/or memory resources should be allocated to each input stream so that the algorithm's throughput in terms of generated result tuples is maximized.
- Addressing all of these issues in a unified manner, allows us to develop a powerful optimization framework for sliding window join queries, which, by conducting an experimental study, we prove to be correct and usable in practice.

T_b	stream B time window size. (used for logical windows)
λ_b	stream B arrival rate
B	number of tuples in window B. (in case of logical window, $B = T_b \lambda_b$)
|B|	number of hash buckets in window B. equivalent to the size of hash directory.
B/|B|	number of tuples in a hash bucket in window B
N	node size - number of tuples in a node (B-tree node, T-tree node)
NKey(B)	number of unique keys in window B
M	system memory size (number of tuples)
P_d	weight factor for search - cost of accessing one tuple in data structure *d* during search operation
I_d	weight factor for update - cost of accessing one tuple in data structure *d* during update operation
σ_b	window B selectivity factor - 1/NKey(B)
σ	join selectivity factor - $\min(\sigma_a, \sigma_b)$

Table 1. Definition of terms used in cost model

In summary, we propose using different join algorithms for each input to a streaming join (e.g., hash join for one input, nested loops join for the other.) In our experiments we show that this is important for the performance of sliding window joins. Furthermore, for approximate streaming window joins, we show that the careful allocation of computing and memory resources to the input streams can have a substantial impact on the performance of the algorithm.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 formulates the problems we will address in this paper. Section 4 describes our proposed cost model framework for sliding window joins. Section 5 validates our cost model framework and presents techniques for maximizing join efficiency. Finally, Section 6 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 [2] 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][6] and query scrambling [7]. 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 unit-time basis cost model framework 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 [8], which was optimized for in-memory performance, leading into thrashing on larger inputs. To rectify the situation XJoin was introduced [9]. Similar techniques are presented in [4] as well. A symmetric nested loops join was proposed in the context of online aggregation [10]. However, none of these works addressed the issues of performing window joins over unbounded input streams.

Approximate answering techniques have become an important research issue in stream data management. Many stream applications deal with large number of data sources (often 10^{5+} in sensor network applications) and long running queries. As a result, often times it is desirable to relax the query semantics to allow approximate answers and conserve resources. Query approximation can be done mainly in two ways: (i) by limiting the size of states maintained for queries and (ii) by reducing answer precision.

One of the straightforward ways to limit the size of query states is to put sliding windows over input streams [28][29][17]. In fact, the sliding window constructs are often required as well by application semantics in which the size of window is explicitly specified in the query context. On the other hand, summary data structures—e.g. wavelets, sketches, histograms, and samples—have been used in the context of online aggregation in which rapid, approximate answers are often more desirable than exact, precise answers [30][31][32][33][34][35][36][37][38][39]. Although these structures yield reduced answer precision, the query processing cost with summary structures is significantly lower than that with original data.

Interestingly, the summary structures are typically well suited to the problem of streaming data query processing, in which the number of data sources is large and data trend detection is often the focus rather than calculating exact answers. Recently a good deal of research has been conducted in this area: for example, stream sampling [27], updating summary data structures [36][40][37][41][38], and maintaining stream statistics [11]. Extending the proposed approximation techniques to support sliding window queries is an interesting area for future research. Babcock et al. presented a comprehensive survey in [25] on broad range of topics in stream data management, including the approximate answering.

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 [15][16]. In [12], Babu and Widom proposed 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 [17]. Hancock [29] developed by Cortes et al. is a programming language based system that simplifies the programming work for extracting signatures from data streams. Tucker et al. presented a stream punctuation technique that allows stateful operators like join and aggregation to shed some of their expired states [26]. Other general stream-oriented database architecture works appear in the sensor network application domain [18][19]. Examples of this work include Berkeley's Telegraph [13] and Cornell's Cougar database project [14]. Zdonik et al. [28] proposed a stream monitoring system, Aurora, which also targeted to address the new class of problems that arise in dealing with the massive number of sensor data inputs and continuous queries.

Finally, Viglas and Naughton proposed a rate based streaming query optimization framework [20]. Integrating the rate based optimization model with our unit time cost model is an interesting area for future research.

3. Problem Formulation

In this section, we will give a more precise description of the problem at hand. We will start with the assumptions of our computing environment, then present the parameters we use to model the environment and finally present what decisions we wish to be able to make given our modeling.

Environment

Our environment consists of *infinite streams* arriving into the system over *network links*. The tuples of the streams are buffered in memory or, if needed, spooled to disk for later processing. The streams are joined by either equality or inequality *predicates*. To efficiently evaluate the join, the system may build on-the-fly indices on top of the streams. We consider three such indices: *hash indices*, *B+tree* and *T-tree* indices. Additionally, we investigate the case of *not building an index*, if the cost of building and maintaining one is higher than that of simply scanning an in-memory buffer.

Parameters

The input streams enter the system through a network link. To capture that effect the network community has traditionally used the *arrival rate* of the input process. We are mostly interested in the effect the incoming rates have on the type of processing we have to perform, as in the case of one input stream being faster than the other. As we will see, this poses a number of interesting questions.

A window predicate accompanies each query, which effectively limits the amount of buffering the system has to perform. There are two important parameters: the *predicate semantics*, and the *window size*. The first parameter allows us to use different index structures so we can efficiently answer it (for instance, hash indices can only handle equality joins.)

The second parameter has to do with how much state per input we need to consider. We make a distinction between logical windows and physical windows (refer to [25] for more rigorous discussions and examples on window semantics.) Logical windows are defined in terms of logical properties between the tuples of the two participating streams. For instance a logical window is “tuples arriving within the last 10 seconds.” Physical windows on the other hand, pose more rigid, physical constraints. For example, a physical window could instruct only the last one thousand tuples of a stream to be kept. On the other hand, these window constraints could either be explicit or implicit. The two previous constraints, for instance, are explicit. Had the two constraints been expressed in relative terms, as in “tuples arriving within 10 seconds of each other,” or “tuples arriving within one thousand arrivals from each other,” the window constraints would be implicit.

Additionally, there are certain issues regarding the entity that provides the timing for the query. Timestamps could either be generated at acceptance time, meaning when the tuple is accepted for processing, or at generation time, meaning the time the tuple was sent from the remote source. We will use the acceptance time based timestamp semantics in this work.

Finally, the system has a fixed number of computing and memory resources. Depending on the arrival rates of the input streams and the window size, there are four cases we need to consider:

1. *Unlimited computational, unlimited memory resources*: the system has enough computational resources to handle the inputs, while the window sizes fit entirely in memory.
2. *Unlimited computational, limited memory resources*: the system can still handle the computational part of the join evaluation without problems, but the memory buffer allocated to the query is not sufficient to keep the window size entirely in main-memory.
3. *Limited computational, unlimited memory resources*: while the window size fits entirely in memory, there are restricted computational resources allocated to the query. In such a scenario, the system must decide how to efficiently use these resources.
4. *Limited computational, limited memory resources*: the system has insufficient resources to deal with the incoming input rates (i.e., the speed of the streams is faster than what the system can handle) while at the same time there is not enough main memory to keep the window entirely in memory.

		Memory Resources	
		Unlimited	Limited
Computational Resources	Unlimited	Section 5.1	Section 5.2.2
	Limited	Section 5.2.1	Section 5.2.3

Table 2. The four possible resource limitation scenarios

This classification partitions our problem space into four quadrants, presented in Table 2.

Decisions

The main issue when dealing with queries over streams arriving at different rates is to maximize the throughput of the query. Given our modeling, there are two important decisions that need to be made:

1. What would be the best way to allocate the fixed number of resources, both computational and memory?
2. What would be the most efficient index structure so that the throughput is maximized given the resources allocated to the processing of the query?

4. Estimating the Cost of Sliding Joins

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.

4.1. Generic Framework for Unit-time Cost Estimation

Consider the join of two windowed streams, *A* and *B*. Each tuple arrival in window *A* triggers three tasks: checking window *B* for joining tuples, inserting the tuple in window *A* and invalidating any expired tuples from that window. Given the notation of Table 1, a cost formula for that operation is shown below.

$$\begin{aligned} C_{A\bowtie B} &= \lambda_a(probe(b) + insert(a) + invalidate(a)) \\ &+ \lambda_b(probe(a) + insert(b) + invalidate(b)) \end{aligned} \quad (1)$$

The first factor of the formula measures the processing cost for stream *A* arrivals, while the second factor does the same for stream *B* arrivals. In each factor, each processing 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.

The choice of window semantics affects the invalidation cost. If the window is defined as “the last n tuples arrived” (i.e., a physical window), invalidation can be done simply by throwing away the oldest tuple. Alternatively, if the window is defined as “the last n seconds” (i.e., a logical window), the actual number of invalidated tuples may vary depending on how tuples are distributed in the input stream—e.g., uniform inter-arrival time vs. Poison process. While addressing the differences between physical and logical windows is an interesting area for future work, it is not central to the contributions of this paper. In what follows, we will assume a physical window as the basis of our cost model.

Another interesting point is that the cost of a single join operation can be divided into two independent subgroups of components, one for each input stream. We can rewrite Equation 1 as follows:

$$\begin{aligned} C_{A\bowtie B} &= C_{A\ltimes B} + C_{A\rtimes B} \\ C_{A\rtimes B} &= \lambda_a(probe(b)) + \lambda_b(insert(b) + invalidate(b)) \\ C_{A\ltimes B} &= \lambda_b(probe(a)) + \lambda_a(insert(a) + invalidate(a)) \end{aligned} \quad (2)$$

Rewriting the formula as above gives rise to two important observations:

1. The join operation is divided into two subcomponents, $A \rtimes B$ and $A \ltimes B$. We call these subcomponents *join directions*.
2. Each subcomponent can be evaluated independently of the other. The cost expression for $C_{A \rtimes B}$ is independent of the cost expression for $C_{A \ltimes B}$. In practice, this means that we can use a different evaluation algorithm for each direction. For instance, we can use nested-loops to evaluate the $A \rtimes B$ direction and a hash index to evaluate the $A \ltimes B$ direction.

The first cost formula (for $C_{A \rtimes B}$) captures the aggregate cost of accessing window *B* in a single time unit. In a given time unit, λ_a tuples arrive in window *A* and these tuples must be joined with tuples in window *B*, hence $\lambda_a(probe(b))$. In the same time unit in window *B*, λ_b tuples arrive and one tuple gets expired per each tuple inserted, hence the second term, $\lambda_b(insert(b) + invalidate(b))$. For illustration, suppose we perform a *nested-loops join* (*NLJ*) from *A* to *B*, and we estimate the cost of the (*probe*, *insert*, *invalidate*) operations in terms of the number of tuples touched. 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, assuming physical window semantics, one tuple will be inserted and one tuple invalidated.) Notice that all three processing terms are determined without knowing the join algorithm chosen for the *B* join *A* direction.

Counting the number of accessed tuples gives a reasonably accurate estimate. It is possible, however, to improve the estimate's precision by refining the processing costs in terms of the physical operations that need to be carried out. Assuming the tuples of each window are stored in some data structure, probing the structure actually translates to "searching the structure for matches." On the other hand, inserting and/or invalidating the structure translates to "updating the structure." This allows us to look at different data structures in a unified manner. The cost of each processing term of Equation 2 can then be expressed in terms of the number of tuples accessed, multiplied by the physical operation's per-tuple processing cost, as follows:

$$
\begin{aligned}
probe(b) &= \#\text{tuples touched while probing window b} \\
&\quad \times \text{weight factor for search} \\
insert(b) &= \#\text{tuples touched while inserting a tuple into} \\
&\quad \text{window b} \times \text{weight factor for update} \\
invalidate(b) &= \#\text{tuples touched while invalidating a tuple} \\
&\quad \text{from window b} \times \text{weight factor for update}
\end{aligned}
$$

In the following sections, we will further refine the cost formulas based on specific join algorithm implementations. In particular, we will address four possibilities: (i) performing a simple nested-loops join, (ii) building a hash index over the window, (iii) building a B+tree over the window and performing an index nested-loops join and (iv) building a T-tree index over the window and performing an index nested-loops join.

4.2. Specific Implementations

Cost of One-Way Nested Loops Join

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):

$$
\begin{aligned}
&C_{A \rtimes B}(NJ) = \lambda_a B \times P_n + 2\lambda_b \times I_n \\
&\text{where } P_n = \text{weight factor for NLJ search} \qquad (3) \\
&\qquad I_n = \text{weight factor for NLJ update}
\end{aligned}
$$

The term $\lambda_a B \times P_n$ represents the number of tuples accessed to search for matches in window *B*, multiplied by the per-tuple access cost for search in an in-memory buffer. It is the *NLJ*-specific equivalent of $\lambda_a(probe(b))$ in Equation 2. The invalidation and insertion costs are straightforward for the *NLJ* case. In a given time unit, λ_b tuples arrive from stream *B* and are inserted in its window, while the same number of tuples expire. The second term of the cost formula, $2\lambda_b \times I_n$, represents this cost.

Cost of One-Way Hash Join

In the case of a traditional hash join, the cost of *probe*(*b*) and *invalidate*(*b*) in Equation 2 is a function of the hash bucket size in window *B*. A typical probe action requires one key hashing and as many key comparisons as there are tuples in the retrieved bucket. The invalidation task also performs similar actions. However, in window joins, tuples are expired in the order of arrivals. Taking advantage of this, we can keep the invalidation cost significantly lower by preserving arrival orders of tuples in each hash bucket, allowing us to directly identify the oldest tuple in a bucket without checking the timestamps of the bucket's tuples. Now the invalidation cost is reduced to one key hashing and one tuple access cost. The modified *HJ* cost formula is shown below.

$$C_{A \rtimes B}(HJ) = \lambda_a \frac{B}{|B|} \times P_h + 2\lambda_b \times I_h \qquad (4)$$

As shown in Table 1, $B/|B|$ represents the number of tuples in a hash bucket of window *B*. A typical in-memory hash table implementation can ensure the number of buckets remains close to the number of unique keys in the window. However, there is a tradeoff between memory utilization and performance improvement by keeping the size of bucket small. The constant weight factors, P_h and I_h , represent the cost of accessing a single tuple in either a search or an update operation respectively. Later, we will show how to determine these weight factors.

Cost of One-Way B+tree Index Nested Loops Join

Hash indices may offer good performance on both probe and update operations. However, a hash index is only usable in equality join cases because the hash index does not preserve logical orders of key values. On the other hand, we can use *NLJ* for non-equality joins. The problem there is that though *NLJ* has a lower cost for update operations, it is not so efficient in terms of search operations. Consequently, it is unlikely to give reasonably good performance on search-heavy workloads. In other words, if stream *A* is much faster than stream *B* in a one-way join *A* to *B* (i.e., more searches will be performed) the join performance will severely suffer from the high search cost.

To rectify this situation, we can build an index over window B that is more tailored toward search-heavy workloads and perform an index nested-loops join. We implemented two such index structures, B+tree and T-tree, for comparison. The cost formula for the B+tree index nested loops join is shown below.

$$C_{A\times B}(BJ) = \lambda_a \times (\left\lceil \log_{N+1}\left\lceil \frac{B}{N} \right\rceil \right\rceil + 1) \times \lceil \log_2 N \rceil \times P_b$$
$$+ \lambda_b \times 2 \times (\left\lceil \log_{N+1}\left\lceil \frac{B}{N} \right\rceil \right\rceil + 1) \times \lceil \log_2 N \rceil \times I_b \quad (5)$$

As defined in Table 1, N denotes the size of a B+tree node. In the first half of the formula, $\lceil B/N \rceil$ represents the number of leaf nodes in a B+tree, and $\lceil \log_{N+1} \lceil B/N \rceil \rceil$ represents the number of non-leaf nodes that need to be searched from the root in order to reach the appropriate leaf. Hence, the height of a B+tree is $\lceil \log_{N+1} \lceil B/N \rceil \rceil + 1$. The search cost inside a B+tree node is equal to $\lceil \log_2 N \rceil$, since we assume binary search is performed within the node. The B+tree index performs a search for a key at each node it visits on the way toward the leaf that contains the search key. Therefore, the cost of single probe is equal to the product of the tree's height, $\lceil \log_{N+1} \lceil B/N \rceil \rceil + 1$, and the search cost within a node, $\lceil \log_2 N \rceil$, multiplied by the B+tree search weight factor, P_b.

Similarly, for both insertion and invalidation, we need to search the tree first to identify the location of the tuple to be inserted or deleted in the tree. Once the location is identified, we insert or delete the new tuple. These insert and delete operations can cause structural change of the B+tree, which often includes nodes splitting, merging or contents of nodes shifted to neighboring nodes. The hidden cost of a B+tree update operation is captured in the update weight factor, I_b. Hence, the cost of both insert and invalidation is equal to the number of tuples touched while searching for the insert or delete location, multiplied by the weight factor for updating the B+tree, I_b.

Cost of One-way T-tree Index Nested Loops Join

The T-tree index was proposed by Lehman and Carey [21] as an index structure for main-memory databases. They have shown that the T-tree has better memory utilization and search and update performance than the B+tree. However, recent studies suggest that a careful in-memory B+tree implementation may outperform T-tree as the B+tree has better processor cache (e.g. L1, L2 cache) utilization characteristics [22][23]. The studies argue that this is particularly true with the modern hardware memory hierarchy where L1 and L2 caches are more than 100 times faster than main-memory.

We do not expect, however, there will be a big difference (e.g. compared to difference between T-tree and Hash) in performance between the two index structures, since both index structures order tuples based on logical key values and perform tree-based search. Furthermore, as in the case of a B+tree, invalidation requires performing both search and update operations. The cost formula for the index nested-loops join using a T-tree index is shown below.

$$C_{A\times B}(TJ) = \lambda_a (1.5 \times (\left\lceil \log_2 \left\lceil \frac{B}{N} \right\rceil \right\rceil - 1) + \lceil \log_2 N \rceil) \times P_t$$
$$+ \lambda_b (2 \times 1.5 \times (\left\lceil \log_2 \left\lceil \frac{B}{N} \right\rceil \right\rceil - 1) + \lceil \log_2 N \rceil) \times I \quad (6)$$

The T-tree is similar to the AVL tree in the way searches and updates are performed. The major difference is that the T-tree is allowed to have more than one data entry in a node. This substantially improves memory utilization of the T-tree, as the grouping of data entries eliminates a large number of pointers. One side effect of this is that each node now has a lower bound and an upper bound key and because of this, on average, it requires 1.5 key comparisons before it can determine which pointer to follow during searches. Unlike the B+tree, data entries are distributed to all nodes in the T-tree. Therefore, the number of nodes in a tree is $\lceil B/N \rceil$ and the height of the tree is $\lceil \log_2 \lceil B/N \rceil \rceil$. In the T-tree, a search key may be found in a non-leaf node. The average number of nodes that a look-up operation has to visit before finding the key is approximately one less than the height of a tree. Once a node that contains the key is found, it performs binary search to identify the matching data elements. Hence, the cost of single probe is $(1.5 \times (\lceil \log_2 \lceil B/N \rceil \rceil - 1) + \lceil \log_2 N \rceil) \times P_t$. The cost formulas for insert and invalidation can be drawn similarly to the B+tree index case.

Testbed Implementation

We implemented the four data structures introduced in this section. In addition, we implemented a sliding window join operator that can accommodate asymmetric combinations of any of the four data structures. The operators were implemented in Java and run on Sun Microsystems' Java HotSpot Client VM 1.4.0. Experiments were performed on an AMD Athlon XP 1533Mhz machine with 1GB of memory, running Windows XP Professional.

In all four join implementations, we maintained the arrival order of tuples in a sliding window by chaining them with one-directional pointers. On each arrival of a new tuple, one tuple is removed from the tail and the new tuple is added onto the head of the chain. Then, the tuple removed from the chain is also removed from the index, if any.

Estimating the Weight Factors

So far, we have been using P_d and I_d (where d represents one of the four join data structures) to mask implementation effects and/or system dependent costs. In this section, we illustrate how we measured the weight factors for each implementation.

To estimate the weight factors, first we measured the CPU time of each join implementation, by processing 60 seconds worth of tuples without intermittent delays. Then, we compared the measured run time with the cost formula while adjusting the two weight factors. To make the task simpler, we measured the run time of an algorithm with two different workloads: one with search-only workload and the other with insert/invalidate-only workload. The result from the search-only workload was used to determine P_d, while the result from the insert/invalidate workload was used to determine I_d.

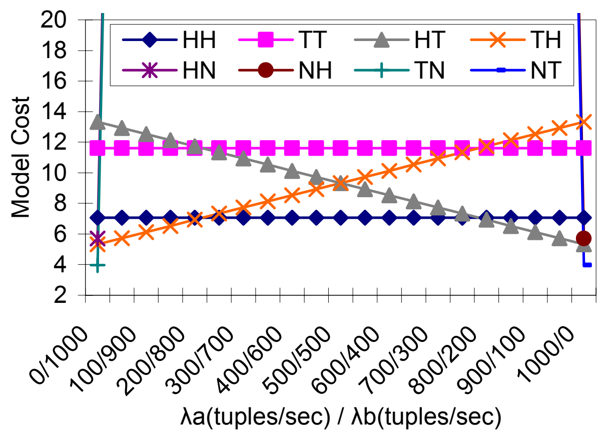

Figure 2. Estimated costs of eight join algorithm combinations. (Size of window A = 5000, Size of window B = 5000, Hash bucket size = 10, T-tree node size = 100)

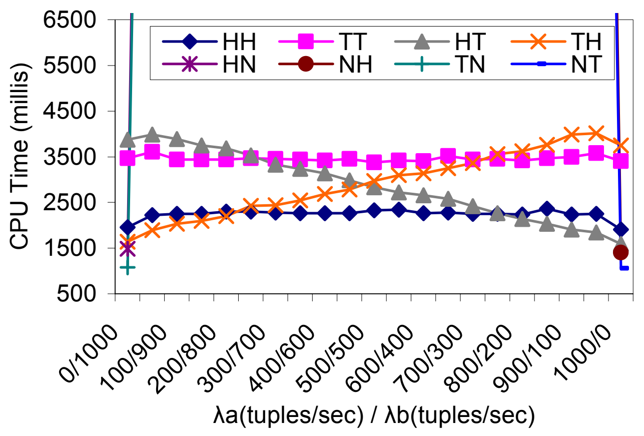

Figure 3. Measured system costs of the same eight join combinations after processing 300 seconds' worth tuples without intermittent delays (Size of window A = 5000, Size of window B = 5000, Hash bucket size = 10, T-tree node size = 100)

For instance, to measure the CPU time of a search-only workload with an arrival rate of 100 tuples/sec, we processed 6000 tuples (60 seconds worth) in one batch and measure the total running time. We chose this way instead of measuring individual tuple handling costs because in this way, we can measure the CPU cost even for an input load that exceeds the system's capacity. For instance, if the estimated cost of an algorithm crosses the 60 seconds line at the arrival rate of 120 tuples/sec, this implies that the algorithm will require full computing power of the system to process the input rate of 120 tuples/sec and above.

We measured CPU times of 20 different points with increasing workload rates, then equated the measured values with the cost formula and calculated the weight factors. The cost formula with the measured weight factors is shown below.

$$
\begin{aligned}
C_{A\times B}(NJ) &= \lambda_a B \times 3\times10^{-4} + 2\lambda_b \times 10^{-4} \\
C_{A\times B}(HJ) &= \lambda_a \frac{B}{|B|} \times 5.5\times10^{-4} + 2\lambda_b \times 7.8\times10^{-4} \\
C_{A\times B}(BJ) &= \lambda_a \times (\left\lceil \log_{N+1} \left\lceil \frac{B}{N} \right\rceil \right\rceil + 1) \times \lceil \log_2 N \rceil \times 2.6\times10^{-4} \\
&\quad + \lambda_b \times 2 \times (\left\lceil \log_{N+1} \left\lceil \frac{B}{N} \right\rceil \right\rceil + 1) \times \lceil \log_2 N \rceil \times 2.6\times10^{-4} \\
C_{A\times B}(TJ) &= \lambda_a (1.5 \times (\left\lceil \log_2 \left\lceil \frac{B}{N} \right\rceil \right\rceil - 1) + \lceil \log_2 N \rceil) \times 2.6\times10^{-4} \\
&\quad + \lambda_b (2 \times 1.5 \times (\left\lceil \log_2 \left\lceil \frac{B}{N} \right\rceil \right\rceil - 1) + \lceil \log_2 N \rceil) \times 2.7\times10^{-4}
\end{aligned}
\tag{7}
$$

Cost of Full Joins

So far we have been focusing only on a one-way join cost formula. We can obtain the full join cost formula by adding any two one-way join cost formulas. For instance, if we add the cost of a hash join from *A* to *B* and the cost of a hash join from *B* to *A*, we have the cost of a full symmetric pipelined hash join. Similarly, if we put *HJ* and *NLJ* together, we get an asymmetric pipelined join with a *HJ* data structure built on one side and a *NLJ* on the other. Notice that our one-way join cost formula representation is completely independent from the data structure used in the opposite side, and this enabled the cost estimation of asymmetric combinations of join algorithms.

5. On Maximizing the Efficiency of Processing Joins

In this section, we investigate strategies for maximizing the efficiency of processing sliding 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.

The first scenario is dealt with in the context of both memory and computing resources being sufficient for the query workload, and is presented in Section 5.1. The second scenario concerns cases where the computing resources required for the workload exceed the amount of resources available in the system. The third scenario deals with the cases where the memory resources are the bottleneck. The second and third scenarios are presented in Section 5.2.

5.1. Exploiting Asymmetry in the Speed of the Input Streams

In this section, we consider the case where the two sliding windows fit in memory and the aggregate speed of two input streams is less than the system's service rate μ (i.e., $\lambda_a + \lambda_b < \mu$.) The focus in this section is to validate the unit-time-basis cost model framework presented in the previous section and using the cost model, to show how to find the best join algorithm combination for a given workload.

To begin with, let us examine the cost graph shown in Figure 2. It shows the cost graphs of eight join algorithm combinations composed from three one-way joins: *HJ*, *NJ*, and *TJ*. We ignored the combinations of join algorithms with a B-tree index because its performance was very close to that of a T-tree, while a B-tree consumes more memory than a T-tree does. The join combination *NNJ* is not shown in the graph because its cost was too high.

Memory consumption is an important issue in a continuous query. Because of its real-time nature, efficient memory utilization is one of the key criteria in selecting an algorithm, since expensive disk I/Os should be avoided. Tradeoffs between memory utilization and performance were most significant in the case of a hash join (We implemented static hash index with bucket chains.) In our hash join implementation, memory utilization improves as we increase the size of the hash bucket (i.e. *B*/|*B*| in cost terms.) We tested bucket sizes from two to one hundred. The improvement was steep up to ten tuples per bucket, and then flattened up gradually. On the other hand, increasing the bucket size affected the hash join's performance negatively. As the bucket size increases by one, the algorithm needs to perform on average one more comparison for each probe. We chose to use ten tuples per bucket because it brings the memory utilization close to that of T-tree, while keeping performance better than both the T-tree and the B-tree in a *non-skewed workload*. A non-skewed workload is a workload not skewed to either search or update. For instance, for a one-way join *A* to *B* ($C_{A\ltimes B}$), stream *A* is the search workload and *B* is the update workload, and a non-skewed workload means the speed difference between the two streams is not significant (i.e., the search to update ratio is close to one.)

On the other hand, the T-tree and the B-tree were less sensitive to the size of a tree node in terms of both memory utilization and performance. We chose to use 100 tuples per node for both the T-tree and the B-tree. In our implementation, given the node sizes, the T-tree provided the best memory utilization among the three. In the test run, we observed that the *HJ* (bucket size=10) consumed roughly about 5% more memory and the *BJ* (w/ tree order d=50 [24]) consumed about 10% more than the *TJ*.

In Figure 2, we have four important crossover points: starting from the far left, one between *TN* and *TH*, then *TH* and *HH*, *HH* and *HT*, and finally *HT* and *NT*. This implies that we have five winning combinations of join algorithms among the range of workloads. *TN* outperforms others at the far left side of the graph where the workload is highly skewed toward stream *B*. Then, *TH* takes over and dominates until workloads reach the 20%/80% (λa/λb) point. After that, *HH* takes over and dominates until around the 80%/20% point. The rest of the graph is rather symmetric with the fourth and fifth combinations, *HT* and *NT*.

Notice that the graph is based on the case where the window sizes for the two windows are equal (5000 tuples each.) If we change the window size, the crossover points will move either left or right depending on the window size ratio. For instance, if we increase window *A* and decrease *B*, all four crossover points in the graph will move towards the right.

Furthermore, as we mentioned earlier, we can improve the performance of Hash Join by keeping the size of the hash bucket low. With a hash bucket size of two we can indeed reduce the number of crossover points in the graph to two, as *HH* will dominate both *HT* and *TH* for all ranges of inputs. But again, this will significantly hamper the memory utilization of *HH*.

Figure 3 shows the result of a test run measuring the system costs of the same eight join combinations. As we can see by comparing Figure 2 and 3, the cost model's estimation is quite accurate on predicting the crossover points as well as the overall shape of each join combination's performance graph.

In fact, using the cost model, we can calculate the exact crossover points by equating the cost of two neighboring join combinations. For instance, to calculate the *TN-TH* crossover point, we equate the costs of *TNJ* and *THJ* as shown below.

$$\begin{aligned}
&C_{A\bowtie B}(TNJ) - C_{A\bowtie B}(THJ)\\
&= (C_{A\ltimes B}(TJ) + C_{A\rtimes B}(NJ)) - (C_{A\ltimes B}(TJ) + C_{A\rtimes B}(HJ))\\
&= C_{A\rtimes B}(NJ) - C_{A\rtimes B}(HJ) = 0\\
&\Leftrightarrow \lambda_a(B\times 3\times 10^{-4} - \frac{B}{|B|}\times 5.5\times 10^{-4})\\
&\quad + \lambda_b(2\times 10^{-4} - 2\times 7.8\times 10^{-4}) = 0\\
&\Leftrightarrow \frac{\lambda_a}{\lambda_b} = \frac{13.6}{3B-55}
\end{aligned} \tag{8}$$

The term *B*/|*B*| captures the hash bucket size of the implementation; hence we replaced it with 10. Interestingly, the crossover point between *TN* and *TH* is only dependent on the size of window *B*. If the size of window *B* increases, the crossover point will move towards the left (i.e., where the speed of stream *B* is far greater than *A*.) If the size of *window B* decreases, the crossover point will move towards the right. For example, suppose we have a window *B* of size 500 tuples. Then, the estimated cross-over point is 0.0094, which means if stream *B* is more than 106 times faster than stream A, *TNJ* will outperform *THJ*, and if *B* is less than 106 times faster than *A*, *THJ* will outperform *TNJ*.

The remaining three crossover points are shown below. The term *N* in the second and third crossover points represents the node size of the T-tree and is 100 in our implementation.

$$\begin{aligned}
&C_{A\bowtie B}(THJ) - C_{A\bowtie B}(HHJ) = C_{A\ltimes B}(TJ) - C_{A\ltimes B}(HJ) = 0\\
&\Leftrightarrow \frac{\lambda_a}{\lambda_b} = \frac{58.9 - 3.9\left\lceil \log_2\left\lceil \frac{A}{N}\right\rceil\right\rceil - 2.6\lceil \log_2 N\rceil}{8.1\left\lceil \log_2\left\lceil \frac{A}{N}\right\rceil\right\rceil + 2.7\lceil \log_2 N\rceil - 23.7}
\end{aligned} \tag{9}$$

$$\begin{aligned}
&C_{A\bowtie B}(HHJ) - C_{A\bowtie B}(HTJ) = C_{A\rtimes B}(HJ) - C_{A\rtimes B}(TJ) = 0\\
&\Leftrightarrow \frac{\lambda_a}{\lambda_b} = \frac{8.1\left\lceil \log_2\left\lceil \frac{B}{N}\right\rceil\right\rceil + 2.7\lceil \log_2 N\rceil - 23.7}{58.9 - 3.9\left\lceil \log_2\left\lceil \frac{B}{N}\right\rceil\right\rceil - 2.6\lceil \log_2 N\rceil}
\end{aligned} \tag{10}$$

$$\begin{aligned}
&C_{A\bowtie B}(HTJ) - C_{A\bowtie B}(NTJ) = C_{A\ltimes B}(HJ) - C_{A\ltimes B}(NJ) = 0\\
&\Leftrightarrow \frac{\lambda_a}{\lambda_b} = \frac{3A-55}{13.6}
\end{aligned} \tag{11}$$

Figure 4 shows the performance of the join combinations in three different workload settings. Figure 4(a) represents the case where the workload is highly skewed. In this example, window *A* is much larger than window *B* and input stream *B* is much faster than input stream *A*. The test result is in line with the cost model estimation. The estimated costs of the three representative workloads (used in Figure 4) are shown in Table 3. Notice that the estimation was accurate as it correctly predicted the winning combination in each workload group. Furthermore, the cost model produced the estimation in a correct order, which was an exact match with the order of the system costs measured during the test run. For instance, in Figure 4(a), *TNJ* exhibited the best performance, while *HTJ* the

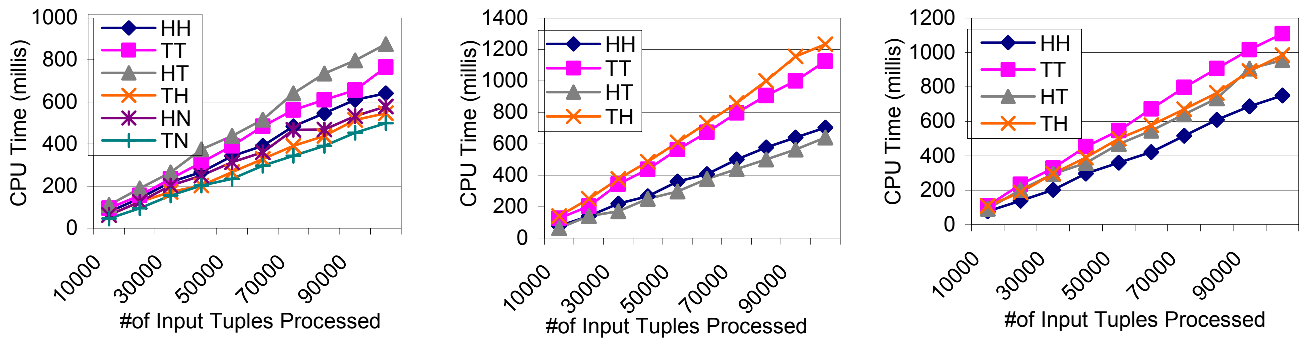

Figure 4. Measured System Costs of the Join Combinations with Increasing Number of Input Tuples. a) (left) Window A = 9500, Window B = 500, λa = 2, λb = 998 b) (middle) Window A = 7000, Window B = 3000, λa = 800, λb = 200 c) (right) Window A = 4000, Window B = 6000, λa = 550, λb = 450

Window A	Window B	λa	λb	HH	TT	HT	TH	HN	NH	TN	NT
9500	500	2	998	7.06	9.56	10.89	5.74	5.99	2845.87	4.67	2849.69
7000	3000	800	200	7.06	11.85	6.46	12.46	722.39	424.87	727.78	424.27
4000	6000	550	450	7.06	11.60	8.93	9.73	993.42	543.84	996.09	545.71

Table 3. Cost Model Estimation. The first row is the estimated costs of the workload in Figure 4a, the second row is for 4b, and the last row is for Figure 4c.

worst. The cost model also correctly predicted the relative order between the remaining join combinations.

Figure 4(b) shows the performance graph of a moderately skewed workload. Figure 4(c), on the other hand, shows the performance result of a relatively even workload. The winning combination in Figure 4(b) was *HTJ*, while in Figure 4(c), the winner was *HHJ*. The cost model's estimation was accurate for both cases. Notice that in Figure 4, some of the join combinations are missing in the graphs. We ignored them because their cost graphs were far off the chart.

5.2. Resource Allocation and Join Performance

In the previous section, we presented a technique to identify the best performing join algorithm combinations for a given workload. The discussion was based on the assumption that we have sufficient resources to handle the workload. In this section, we focus on cases where system resources are insufficient to fully support the queries and workloads. As a result, users have to resort to approximate answers rather than exact answers.

Our underlying idea is that even though the system may not have enough resources to compute all tuples of the join, it may have enough resources to compute some subset of the join tuples. If the complete query involves some aggregate (for example, average) over the join, users may be willing to accept an estimate based upon this subset instead of the exact result. The interesting question that arises is how to maximize the accuracy of this estimate given the limited resources.

In what follows, we use the insight that maximizing the number of tuples produced by the join will yield the best expected approximate answer, since it corresponds to having a larger sample of the true join result. Obviously some care must be taken to ensure that the subset of the join result produced is a random sample of the join. In the following, we assume that when the join algorithm limits its resource usage, it does so in a random way. For example, if the "full" join window on Stream *A* should contain 10,000 tuples, and the resource allocation strategies tell us we can only afford to keep 5000 tuples, we keep a randomly chosen subset of 5000 tuples out of the full 10,000. Similarly, if the resource allocator tells us we can only afford enough CPU resources to probe 50% of the *A* tuples into the *B* window, we use a randomly chosen 50% to probe.

Now, the question is how to allocate the limited resources in a way that improves the accuracy of approximate answers. Should we allocate the resources across the streams in proportion to input stream rates? Should we do so proportionally to the size of each window? We focus on this problem and investigate efficient resource allocation strategies in three remaining quadrants of our problem space defined in Table 2.

5.2.1. Case of Limited Computing Resources and Sufficient Memory.

In this subsection we investigate the case where computing resources are insufficient to keep up with the rates of the input streams. Let us start with a formula that captures the output rate of a sliding window join operation. In the following equation, the selectivity factors of windows *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.

$$r_o = \min(\sigma_a, \sigma_b)(\lambda_a B + \lambda_b A) = \sigma(\lambda_a B + \lambda_b A) \qquad (12)$$

In this scenario, however, we have limited computing resources and as a result we cannot support the full speed of the input streams. Hence, the equation above should be rewritten as follows:

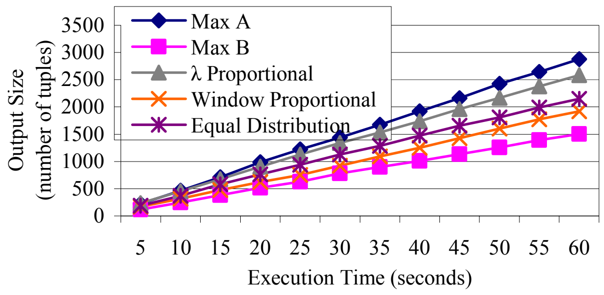

Figure 5. Computing Resource Allocation Strategy Evaluation. (λ_a=800, λ_b=200, A=100, B=200, σ=0.01, μ=100)

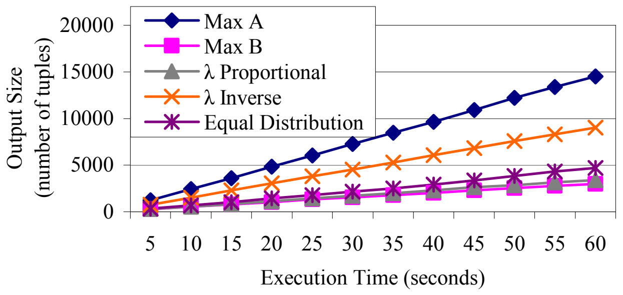

Figure 6. Memory Allocation Strategy Evaluation. (λ_a=10, λ_b=50, M=1000, σ_a=0.005, σ_b=0.01)

$$r_o = \sigma(\lambda_{a'}B + \lambda_{b'}A)$$
$$\text{where } \lambda_{a'} + \lambda_{b'} = \mu \text{ (join operator service rate)}, \quad (13)$$
$$\lambda_{a'} \le \lambda_a,\ \lambda_{b'} \le \lambda_b$$

By applying the constraint in the formula, we get:

$$r_o = \sigma(\lambda_{a'}B + (\mu - \lambda_{a'})A) = \sigma(B - A)\lambda_{a'} + \sigma\mu A \quad (14)$$

Given this equation, it is clear that we have to allocate the maximum amount of computing resources to the join direction that evaluates the join from the small window to the big one. In case the two window sizes are equal, the output rate is constant and equals to $\sigma\mu A$ regardless of the two input rates.

For instance, suppose that we have a window *A* of size 500 and *B* of size 1000, and the join operator can handle up to 100 tuples/sec; the speed of each input stream is greater than the service rate. Furthermore, we assume that the cost of inserting and invalidating a tuple in a window is relatively small compared to the cost of evaluating the join and in turn we can effortlessly maintain the two windows without dropping input tuples. The best resource allocation strategy in this example is to put the maximum resources in the join direction *A* to *B*. That is, the join service rate of 100 tuples/sec should all be used for probing the window *B*. Hence, the $\lambda_{a'}$ and $\lambda_{b'}$ in the formula become 100 and zero, respectively, and the maximum output rate we get is $\sigma \times 100K$ tuples/sec. We will call the adjusted rates $\lambda_{a'}$ and $\lambda_{b'}$ the effective rates for streams *A* and *B*, respectively.

In practice, the actual distribution of resources should be done in the context of an application. If an application requires to process at least 10 tuples from each input stream in any given time unit, the resulting resource allocation in the above example should be changed to 90 tuples/sec and 10 tuples/sec to $\lambda_{a'}$ and $\lambda_{b'}$, respectively.

To validate the analysis, we performed a test with five different strategies: maximizing stream *A*'s effective rate, maximizing stream *B*'s effective rate, allocating resources proportionally to the arrival rates, proportionally to window sizes, and finally equally among the two inputs. The result is shown in Figure 5. As we expected, the winner was *MaxA* that allocated the maximum computing resources to the join direction *A* to *B*, which is from the smaller window to the bigger window, and the worst performer was *MaxB* that did the exact opposite.

The join algorithm selection should be performed after the decision for resource allocation. Once we determine the effective rates (workload) of the join operator, we can choose the best join algorithm combinations for the adjusted workload, in the way we showed in Section 5.1.

5.2.2. Case of Limited Memory and Sufficient Computing Resources.

In this scenario, we assume that the memory is the bottleneck. To improve the query result, we can allocate the memory resources across the windows. We add one constraint that reflects this into Equation 12 and obtain:

$$r_o = \sigma(\lambda_a B + \lambda_b A)$$
$$\text{where } A + B = M \text{ (total avaliable memory)} \quad (15)$$

The objective is to allocate memory resources so as to maximize the join output rate. We can address this problem by rewriting the formula above with the constraint, as shown below.

$$r_o = \sigma(\lambda_a(M - A) + \lambda_b A) = \sigma(\lambda_b - \lambda_a)A + \sigma\lambda_a M \quad (16)$$

Given the equation above, it is easy to see that the best strategy is to allocate most memory to the window corresponding to the slower input rate. If stream *B* is faster than stream *A* we should maximize window *A* to maximize the output rate, and vice versa. In the case where both input streams have equal input rates, the size of window *A* and *B* becomes irrelevant to the output rate. In such a case, the output rate is constant and equal to $\sigma\lambda_a M$.

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 available memory to the fast stream and lets the slow stream probe the fast one. It is straightforward that the first scenario is going to outperform the second one, because the number of probe operations is greater in the first case while the size of the target window being probed is identical, and equal to the memory size.

Figure 6 illustrates the evaluation results of memory allocation strategies. We tested five memory allocation strategies that include maximizing the size of window *A*, maximizing the size of window *B*, allocating memory resources proportionally to the arrival rates, inverse proportionally to the arrival rates, and equal distribution. The test results conformed to the analysis.

Similarly, once we determine the window sizes we can calculate the estimated cost of joins with various algorithm combinations using the cost formula presented in Section 4.2.

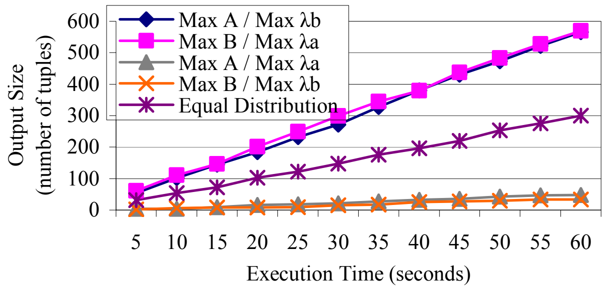

Figure 7. Resource Allocation Strategies for Limited Memory and Limited Computing Resource Cases. (μ=10, M=100, σ=0.01)

Then using the estimation, we can determine the best join algorithm combinations for the given query.

5.2.3 Case of Limited Memory and Limited Computing Resources.

In this scenario, we assume that both memory and computing resources are limited and that we can adjust the two windows so as to fully utilize the given memory. Additionally, we can adjust the effective input rates for the join by allocating the computing resources across the windows. We rewrite Equation 12 with the additional constraints given in this scenario, as follows:

$$r_o = \sigma(\lambda_{a'}B + \lambda_{b'}A) \quad \text{where } A + B = M,\ \lambda_{a'} + \lambda_{b'} = \mu,\ \lambda_{a'} \le \lambda_a,\ \lambda_{b'} \le \lambda_b \qquad (17)$$

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

$$\lambda_{a'} = x\mu,\ \lambda_{b'} = (1-x)\mu,\ B = M - A \qquad (18)$$

By making the substitutions, we obtain:

$$\begin{aligned} r_o &= \sigma(x\mu(M-A) + (1-x)\mu A) \\ &= \sigma(x\mu M - (2x-1)\mu A) \end{aligned} \qquad (19)$$

Now the output rate becomes a function of the two variables x and A. In the case where x > ½, the second term in the subtraction, $(2x-1)\mu A$, is greater than zero. To maximize ro, the second term must be minimized, thus indicating a minimization of A. If A is minimized to zero, the term $\sigma x \mu M$ remains. To maximize it, we need to take the maximum x value, and then ro converges to $\sigma\mu M$. In other words, we should minimize the size of window A and maximize stream A's effective rate. Similarly, in the case where x < ½, we should maximize the size of window A and minimize stream A's effective rate. In fact, $\sigma\mu M$ is the maximum output rate that we can achieve. This is because μ is the maximum number of tuples that the join operator can process in a time unit and M is the maximum possible target window size.

Figure 7 presents an experimental result of the performance of various resource allocation strategies. We evaluated five different strategies: (i) *Max A / Max λb*, which maximizes the size of window A and stream B's effective rate, similarly, (ii) *Max B / Max λa*, (iii) *Max A / Max λa*, (iv) *Max B / Max λb*, and (v) *Equal Distribution*.

In the experiment, the best performing group was the combination of maximizing the window size in one window and maximizing the effective arrival rate in the other window. This group consists of *Max A / Max λb,* and *Max B / Max λa*. The next highest performer, *Equal Dist*, is a strategy that distributes an equal amount of resources to each stream. The worst performer was the group of resource allocation strategies that maximizes the size and the effective arrival rate of the same window. The experimental results conformed to the analysis.

6. Conclusion

In this paper we investigated strategies for evaluating sliding 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.

To our knowledge our paper is the first to consider using different join algorithms for each input to a streaming join (e.g., hash join for one input, nested loops join for the other.) We have shown that this is important for the performance of sliding window joins—in our experiments, we observed cases in which the asymmetric streaming algorithms were up to 53% more efficient than the symmetric streaming algorithms. Furthermore, we have shown that when considering approximate streaming window joins, the careful allocation of computing and memory resources to the input streams can have a substantial impact on the performance of the algorithm. For example, in our experiments we observed cases in which an appropriately skewed allocation of resources to input streams generated 90% more answer tuples per unit time than did the naive equal allocation of resources to both input streams.

A good deal of room for future work exists. One interesting direction 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 sliding window joins. Finally, it would be interesting to model and evaluate other algorithms besides the ones presented in this work.

Acknowledgements

This research was supported by NSF grants CSA-9623632 and ITR 0086002.

Bibliography

[1] Douglas B. Terry, David Goldberg, David Nichols, Brian M. Oki: Continuous Queries over Append-Only Databases. SIGMOD Conference 1992: 321-330

[2] Jianjun Chen, David J. DeWitt, Feng Tian, Yuan Wang: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD Conference 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] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, Daniel S. Weld: An Adaptive Query Execution System for Data Integration. SIGMOD Conference 1999: 299-310

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

[6] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, Vijayshankar Raman: Continuously Adaptive Continuous Queries over Streams. SIGMOD Conference 2002

[7] Tolga Urhan, Michael J. Franklin, Laurent Amsaleg: Cost Based Query Scrambling for Initial Delays. SIGMOD Conference 1998: 130-141

[8] Annita N. Wilschut, Peter M. G. Apers: Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991: 68-77

[9] Tolga Urhan, Michael J. Franklin: XJoin: A Reactively-Scheduled Pipelined Join Operator. IEEE Data Engineering Bulletin 23(2): 27-33 (2000)

[10] Peter J. Haas, Joseph M. Hellerstein: Ripple Joins for Online Aggregation. SIGMOD Conference 1999: 287-298

[11] M. Datar, A. Gionis, P. Indyk, R. Motwani: Maintaining Stream Statistics over Sliding Windows, 2002 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)

[12] Shivnath Babu, Jennifer Widom: Continuous Queries over Data Streams. SIGMOD Record 30(3): 109-120 (2001)

[13] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Sam Madden, Vijayshankar Raman, Mehul A. Shah: Adaptive Query Processing: Technology in Evolution. IEEE Data Engineering Bulletin 23(2): 7-18 (2000)

[14] Philippe Bonnet, Johannes Gehrke, Praveen Seshadri: Towards Sensor Database Systems. Mobile Data Management 2001: 3-14

[15] Praveen Seshadri, Miron Livny, Raghu Ramakrishnan: Sequence Query Processing. SIGMOD Conference 1994: 430-441

[16] Praveen Seshadri, Miron Livny, Raghu Ramakrishnan: The Design and Implementation of a Sequence Database System. VLDB 1996: 99-110

[17] 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

[18] Deborah Estrin, Ramesh Govindan, John S. Heidemann, Satish Kumar: Next Century Challenges: Scalable Coordination in Sensor Networks. MOBICOM 1999: 263-270

[19] J. M. Kahn, Randy H. Katz, Kristofer S. J. Pister: Next Century Challenges: Mobile Networking for "Smart Dust". MOBICOM 1999: 271-278

[20] Stratis Viglas, Jeffrey F. Naughton: Rate-Based Query Optimization for Streaming Information Sources. SIGMOD Conference 2002

[21] Tobin J. Lehman, Michael J. Carey: A Study of Index Structures for Main Memory Database Management Systems. VLDB 1986: 294-303

[22] Jun Rao, Kenneth A. Ross: Making B$^+$-Trees Cache Conscious in Main Memory. SIGMOD Conference 2000: 475-486

[23] Jun Rao, Kenneth A. Ross: Cache Conscious Indexing for Decision-Support in Main Memory. VLDB 1999: 78-89

[24] Raghu Ramakrishnan, Johannes Gehrke: Database Management Systems. McGraw-Hill. 2000

[25] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, Jennifer Widom: Models and Issues in Data Stream Systems. PODS 2002: 1-16

[26] http://www.cse.ogi.edu/~ptucker/PStream/

[27] Brian Babcock, Mayur Datar, Rajeev Motwani: Sampling From a Moving Window over Streaming Data. Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)

[28] Stan Zdonik, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, Donald Carney: Monitoring Streams - A New Class of Data Management Applications. VLDB 2002.

[29] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, Anne Rogers, Frederick Smith: Hancock: A Language for Extracting Signatures from Data Streams. Knowledge Discovery and Data Mining Conference 2000: 9-17

[30] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, Sridhar Ramaswamy: The Aqua Approximate Query Answering System. SIGMOD Conference 1999: 574-576

[31] Surajit Chaudhuri, Gautam Das, Vivek R. Narasayya: A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries. SIGMOD Conference 2001

[32] Phillip B. Gibbons, Yossi Matias: New Sampling-Based Summary Statistics for Improving Approximate Query Answers. SIGMOD Conference 1998: 331-342

[33] Venkatesh Ganti, Mong-Li Lee, Raghu Ramakrishnan: ICICLES: Self-Tuning Samples for Approximate Query Answering. VLDB 2000: 176-187

[34] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, Kyuseok Shim: Approximate Query Processing Using Wavelets. VLDB 2000: 111-122

[35] Jeffrey Scott Vitter, Min Wang, Balakrishna R. Iyer: Data Cube Approximation and Histograms via Wavelets. CIKM 1998: 96-104

[36] Yossi Matias, Jeffrey Scott Vitter, Min Wang: Dynamic Maintenance of Wavelet-Based Histograms. VLDB 2000: 101-110

[37] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, Martin Strauss: Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. VLDB 2001: 79-88

[38] Alin Dobra, Minos Garofalakis, J. E. Gehrke, and Rajeev Rastogi: Processing Complex Aggregate Queries over Data Streams. SIGMOD Conference 2002

[39] Noga Alon, Yossi Matias, Mario Szegedy: The Space Complexity of Approximating the Frequency Moments. STOC 1996: 20-29

[40] Johannes Gehrke, Flip Korn, Divesh Srivastava: On Computing Correlated Aggregates Over Continual Data Streams. SIGMOD Conference 2001

[41] Sudipto Guha, Nick Koudas, Kyuseok Shim: Data-streams and histograms. STOC 2001: 471-475