Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams[*]

Lukasz Golab   M. Tamer Özsu

School of Computer Science
University of Waterloo
Waterloo, Canada
{lgolab, tozsu}@uwaterloo.ca

Technical Report CS-2003-01
February 2003

[*]This research is partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Abstract

We study sliding window multi-join processing in continuous queries over data streams. Several algorithms are reported for performing continuous, incremental joins, under the assumption that all the sliding windows fit in main memory. The algorithms include multi-way incremental nested loop joins (NLJs) and multi-way incremental hash joins. We also propose join ordering heuristics to minimize the processing cost per unit time. We test a possible implementation of these algorithms and show that, as expected, hash joins are faster than NLJs for performing equi-joins, and that the overall processing cost is influenced by the strategies used to remove expired tuples from the sliding windows.

1 Introduction

A data stream is a real-time, continuous, ordered (explicitly by timestamp or implicitly by arrival time) sequence of items. Applications where information naturally occurs as a stream of data values include sensor data processing [7, 20, 31], Internet traffic analysis [14, 23], financial tickers [10, 32], and analysis of various transaction logs such as Web server logs and telephone call records [11]. Due to their continuous and dynamic nature, querying data streams involves running a query continually over a period of time and generating new answers as new items arrive. First proposed in [24], these types of queries are known in the literature as *continuous*, *standing*, or *persistent* queries [10, 19].

Several issues arise in on-line stream processing. Firstly, unbounded streams may not be wholly stored in bounded memory. Secondly, because data streams are temporally ordered, new items are often more accurate or more relevant than older items. Finally, streaming query plans may not use blocking operators that must consume the entire input before any results are produced.

A common solution to these issues is to restrict the range of continuous queries to a sliding window that contains the last T items or those items that have arrived in the last t time units. The former is called a *count-based*, or a *sequence-based* sliding window, while the latter is called a *time-based* or a *timestamp-based* sliding window [5]. Constraining all queries by sliding window predicates allows continuous queries over unbounded streams to be executed in finite memory and in an incremental manner by generating new results as new items arrive. In particular, while joining two or more infinite streams is seldom feasible (see Arasu et al. for a discussion of memory requirements of streaming queries [2]), evaluating windowed joins over stream excerpts is practical and useful in many applications. For example, an Internet traffic engineer may pose a query that joins traffic traces from various links in a network, with equality comparison of the source and destination IP addresses of each packet header as the join predicate. This windowed join may be used to trace packets through the network and identify sessions whose packets follow different paths to reach the same destination. The latter could be used to study load balancing in a network with many redundant links.

Processing continuous queries over sliding windows introduces two issues that affect the design of windowed algorithms: re-execution strategies and tuple invalidation procedures. An *eager re-evaluation* strategy generates new results after each new tuple arrives, but may be infeasible in situations where streams have high arrival rates. A more practical solution—*lazy re-evaluation*—is to re-execute the query periodically. The downside of lazy re-evaluation is an increased delay between the arrival of a new tuple and the generation of new results based on this tuple—long delays may be unacceptable in streaming applications that must react quickly to unusual patterns in the data. Similarly, *eager expiration* proceeds by scanning the sliding windows and removing old tuples upon arrival of each new tuple (this is trivial in count-based windows as each new tuple simply replaces the oldest tuple in its window). In contrast, *lazy expiration* involves removing old tuples periodically and requires more memory to store tuples waiting to be expired.

1.1 Problem Statement

Given n data streams and n corresponding sliding windows, our goal in this work is to continually evaluate the exact join of all n windows. We assume that each stream consists of relational tuples with the following schema: a timestamp attribute *ts* and an attribute *attr* containing values from an ordered domain. We also assume that all windows fit in main memory and we require that all query plans use extreme right-deep join trees that do not materialize any intermediate results. Furthermore, we do not permit time-lagged windows, i.e. all windows are assumed to start at the current time and expire at the current time minus the corresponding window size.

We define the semantics of sliding window joins as monotonic queries over append-only relations; therefore once a result tuple is produced, it remains in the answer set indefinitely. Hence, new results may be streamed directly to the user. In particular, for each newly arrived tuple k, the join is to probe all tuples present in the sliding windows precisely at the time of k's arrival (i.e. those tuples which have not expired at time equal to k's timestamp), and return all (composite) tuples that satisfy every join predicate. Moreover, we impose a time bound τ on the time interval from the arrival of a new tuple k until the time when all join results involving k and all tuples older than k have been streamed to the user. This means that a query must be re-executed at least every τ time units. However, it does not mean that all join results containing k will be available at most τ units after k's arrival; k will remain in the window until it expires, during which time it may join with other (newer) tuples.

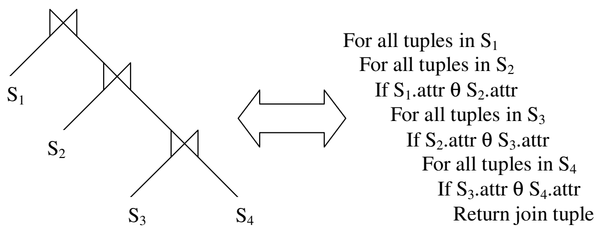
1.2 Summary of Contributions

In this paper, we present a solution to the multi-join processing problem over sliding windows residing in main memory. All of our algorithms execute multiple joins together in a series of nested for-loops and process newly arrived tuples from each window separately (possibly using different join orders). We first propose a multi-way NLJ for eager query re-evaluation that always iterates over newly arrived tuples in the outer for-loop. We demonstrate that this strategy may not work well for lazy re-evaluation and propose another NLJ-based operator that is guaranteed to perform as well or better than the initial algorithm. We also extend our NLJ-based algorithms to work with hash indices on some or all sliding windows. Moreover, we propose join ordering heuristics for our algorithms that attempt to minimize the number of intermediate tuples that are passed down to the inner for-loops. These heuristics are based on a main-memory per-unit-time cost model. Finally, we test an implementation of our algorithms and investigate the effects of re-evaluation and expiration strategies on the overall processing cost.

Table 1 lists the symbols used in this paper and their meanings. In Figure 1, we give an

λ_i	Arrival rate of stream i in tuples per unit time
S_j	Sliding window corresponding to stream j
T_j	Time size of the j^{th} time-based window
C_j	Number of tuples in S_j
v_j	Number of distinct values in S_j
b_j	Number of hash buckets in the hash index of S_j, if such an index exists
τ	Continuous query re-execution interval
$a \circ b$	Concatenation of tuples a and b
θ	Arithmetic comparison predicate, e.g. $=$

Table 1: Explanations of symbols used in this paper.



```
For all tuples in $S_1$
  For all tuples in $S_2$
    If $S_1$.attr $\theta$ $S_2$.attr
      For all tuples in $S_3$
        If $S_2$.attr $\theta$ $S_3$.attr
          For all tuples in $S_4$
            If $S_3$.attr $\theta$ $S_4$.attr
              Return join tuple
```

Figure 1: Join order $S_1 \bowtie (S_2 \bowtie (S_3 \bowtie S_4))$ expressed as a join tree (left) and as a series of for-loops (right).

example to explain our convention for describing join ordering. In this example, the join order $S_1 \bowtie (S_2 \bowtie (S_3 \bowtie S_4))$ is expressed as a join tree on the left and as a series of nested for-loops on the right. We refer to S_1 as being "on top of the plan" or "ordered first", S_2 as "ordered second" and so on. For brevity, we may omit parentheses and write S_1, S_2, S_3, S_4 to represent the join order shown in Figure 1. Note that the pseudocode on the right of Figure 1 shows the general strategy used in our multi-way joins, whereby the join is effectively evaluated "from the top down"; the join predicate is evaluated inside each for-loop in order to minimize the number of tuples that are passed down to the inner for-loops.

1.3 Roadmap

The remainder of this paper is organized as follows. In Section 2, we review related work. Section 3 defines our multi-way incremental join algorithms, whose processing cost is analyzed in Section 4. Section 5 outlines join ordering heuristics. We discuss experimental results in Section 6. Section 7 concludes the paper and outlines directions for future research.

2 Related Work

There has been a great deal of recent interest in developing novel data management techniques and adapting traditional database technology to the data stream model; Cougar [31], Aurora [7], and STREAM [22] are some examples. The first two focus on processing sensor data. Cougar concentrates on distributed query processing inside the sensor network, while Aurora allows users to create query plans by visually arranging query operators by using a boxes (corresponding to query operators) and (directed) links (corresponding to data flow) paradigm. STREAM addresses all aspects of stream data management, including memory management, operator scheduling, and approximate query answering via summary information. A continuous query language (CQL) have also been proposed within the STREAM project [3].

Recent work on continuous queries focuses on scalable evaluation of many queries by means of plan sharing and indexing query predicates [9, 10, 21]. TelegraphCQ [8] is a proposed extension of earlier efforts into adaptive query processing [4, 9, 21], where query plans are re-ordered throughout the lifetime of a continuous query in response to changes in the execution environment (e.g. fluctuating stream arrival rates).

Defining sliding windows is one solution proposed in the literature for bounding the memory requirements of continuous queries and unblocking streaming operators. Another alternative is to maintain compact stream summaries and provide approximate query answers over the summaries. Many summary structures have been proposed in the literature; some examples may be found in [1, 13, 15, 16]. In fact, there may be cases where both sliding windows and summary structures may be necessary when even the windows are too large to fit in memory. For instance, Datar et al. [12] give an approximate algorithm for bit counting in a sliding window. The third method is to exploit any constraints that may hold in a data stream. For example, Babu and Widom [6] show that foreign key constraints and ordered or clustered arrival (i.e. stream items arrive in some known order, or duplicate items arrive in one contiguous batch) may be exploited to lower the memory usage in continuous query processing. Moreover, assertions, referred to as *punctuations* in [25, 26], could be inserted into a stream to specify a restriction on subsequently arriving items. For instance, a punctuation may arrive stating that all future items shall have the A attribute larger than ten. This punctuation could be used to partially unblock a group-by query on A—all those groups whose A attribute is larger than ten are guaranteed not to change.

Relevant work on join processing over unbounded streams includes non-blocking binary join algorithms such as the XJoin [27], which is a variant of the symmetric hash join, and the Ripple Join [17]. Viglas et al. [29] have developed a multi-way version of the XJoin called the MJoin. Moreover, Viglas and Naughton [28] propose a rate-based query optimization model for continuous queries over data streams, which is relevant if the input rate changes with time, in which case the output rate of a join also changes with time.

Windowed joins over two streams were studied by Kang et al. [18], who introduce incrementally computable binary joins as well as a per-unit-time cost model which we also use in this paper.

While a multi-way hash join has been proposed in literature, to the best of our knowledge, this work is the first to consider multi-way joins designed explicitly for sliding windows. We also know of no previous work on the join ordering problem in the context of sliding windows over data streams, although this problem is identified in the context of optimizing for the highest

S_1		S_2		S_3	
ts	*attr*	*ts*	*attr*	*ts*	*attr*
90	1	150	1	195	1
100	1	180	1	205	1

Table 2: Partial contents of sliding windows S_1, S_2, and S_3.

output rate of queries over infinite streams [28, 29]. Generally, main-memory join ordering techniques focus on pushing expensive predicates to the top of the plan (see, e.g. [30]).

3 Sliding Window Join Algorithms

3.1 Motivation

We begin the discussion of sliding window multi-way joins with examples that motivate our strategies, starting with the simplest case of eager re-evaluation and expiration. We initially concentrate on time-based windows and later present extensions of our operators to count-based windows in Section 3.6.

A binary incremental NLJ has been proposed by Kang et al. [18] and proceeds as follows. Let S_1 and S_2 be two sliding windows to be joined. For each newly arrived S_1-tuple, we scan S_2 and return all matching tuples. We then insert the new tuple into S_1 and invalidate expired tuples. We follow the same procedure for each newly arrived S_2-tuple. Extending this binary NLJ to the case of more than two windows is straightforward: for each newly arrived tuple k, we execute the join sequence in the order prescribed by the query plan, but we only include k in the join process (not the entire window that contains k). For example, suppose that we wish to join three windows, S_1, S_2, and S_3, using the plan $S_1 \bowtie (S_2 \bowtie S_3)$. Upon arrival of a new S_1-tuple, we invalidate expired tuples in S_2 and S_3 and then probe all tuples in $S_2 \bowtie S_3$. If a new S_2-tuple arrives, we first expire old tuples from S_1 and S_3, then for each tuple currently in S_1, we compute the join of the new tuple with S_3 and probe the result set. Similarly, upon arrival of a new S_3-tuple, we expire S_1 and S_2-tuples, then for each S_1-tuple, we evaluate the join of S_2 with the new tuple and probe the result set. We call this algorithm Naive Multi-Way NLJ.

To perform lazy re-evaluation with the naive algorithm, we re-execute the join every τ time units, first joining new S_1-tuples with other sliding windows, then new S_2-tuples, and so on. Suppose that we wish to process a batch of newly arrived tuples from S_3 given the join order $S_1 \bowtie (S_2 \bowtie S_3)$. For each tuple in S_1, we re-compute the join $S_2 \bowtie S_3$, but using only the newly arrived tuples from S_3. Consider the following example where all windows are 100 seconds long and their partial contents are as shown in Table 2. All join predicates are equalities on the attribute *attr*. If the current time is 210 and we run Naive Multi-Way NLJ, incorrect results will be returned when processing new S_3-tuples. Table 3 shows the incremental result set returned for all new S_3-tuples on the top and the actual result set that should be returned on the bottom. We see that some tuples from S_1 should not join with any new S_3-tuples as those S_1-tuples should have expired by the time the new S_3-tuples have arrived. To solve this problem, we need to verify that a composite $S_2 \circ S_3$ tuple (call the S_2-tuple j and the S_3 tuple k) joins with an S_1 tuple, call it i, only if $i.ts \geq k.ts - T_1$ and $j.ts \geq k.ts - T_2$. These two

Incorrect Results

ts	*attr*	*ts*	*attr*	*ts*	*attr*
90	1	150	1	195	1
90	1	150	1	205	1
90	1	180	1	195	1
90	1	180	1	205	1
100	1	150	1	195	1
100	1	150	1	205	1
100	1	180	1	195	1
100	1	180	1	205	1

Correct Results

ts	*attr*	*ts*	*attr*	*ts*	*attr*
100	1	150	1	195	1
100	1	180	1	195	1

Table 3: Partial results returned by the naive algorithm (top) and correct partial join results (bottom).

conditions ensure that i and j have not expired at time equal to k's timestamp.

3.2 Improved Eager Multi-Way NLJ

In the above examples, when a new S_3-tuple arrives we re-compute the join of S_2 with the new S_3-tuple for each tuple in S_1. Generally, this results in unnecessary work whenever new tuples arrive from a window that is not ordered first in the join tree. We propose a more efficient technique for handling new S_3-tuples, in which we initially select only those tuples in S_1 which join with the new S_3-tuple (suppose there are c such tuples), and make c scans of S_2. In contrast, the previous approach requires a number of scans of S_2 equal to the number of tuples in S_1. Unless all tuples in S_1 satisfy the join predicate (as in a Cartesian product), c is less than the size of S_1. When a new S_1-tuple or a new S_2-tuple arrives, we proceed similarly by selecting those tuples from the window on top of the join order that join with the new tuple, and for each match, scanning the remaining window. In effect, the window at the top of the join order always consists of only one tuple and the join order changes in response to the origin of the incoming tuple (Table 4). This is possible because we have assumed a common join attribute across all streams. We define the global join order of a query plan as the static order that our algorithm would follow had it not ordered new tuples first. In the above example, the global order is S_1, S_2, S_3.

The pseudocode of the algorithm is given below. Without loss of generality, we let the global join order be $S_1, S_2 \ldots S_n$.

```
**Algorithm** EAGER MULTI-WAY NLJ
If a new tuple $k$ arrives on stream $i$
  Insert new tuple in window $S_i$
  COMPUTEJOIN($k$, $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
```

Origin of new tuple	Join order
S_1	$S_1 \bowtie (S_2 \bowtie S_3)$
S_2	$S_2 \bowtie (S_1 \bowtie S_3)$
S_3	$S_3 \bowtie (S_1 \bowtie S_2)$

Table 4: Join orderings used in our multi-way NLJ given that the global join ordering is $S_1 \bowtie (S_2 \bowtie S_3)$.

```
Algorithm ComputeJoin
Input: new tuple $k$ from window $S_i$ and a join order
$\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$.
$\forall u \in S_1$ and $k.ts - T_1 \leq u.ts \leq k.ts$
  If $k.attr\ \theta\ u.attr$
    $\ldots$ \\ loop through $S_2$ up to $S_{i-2}$
    $\forall v \in S_{i-1}$ and $k.ts - T_{i-1} \leq v.ts \leq k.ts$
      If $k.attr\ \theta\ v.attr$
        $\forall w \in S_{i+1}$ and $k.ts - T_{i+1} \leq w.ts \leq k.ts$
          If $k.attr\ \theta\ w.attr$
            $\ldots$ \\ loop through $S_{i+2}$ up to $S_{n-1}$
            $\forall x \in S_n$ and $k.ts - T_n \leq x.ts \leq k.ts$
              If $k.attr\ \theta\ x.attr$
                Return $k \circ u \circ v \circ \ldots \circ x$
```

If desired, projections may be performed on-the-fly by removing unwanted attributes in the Return statement. Algorithm Multi-Way NLJ may be used in conjunction with eager or lazy expiration and is guaranteed not to join any expired tuples because of the timestamp comparisons done in each for-loop.

3.3 Lazy Multi-Way NLJs

A straightforward adaptation of algorithm Eager Multi-Way NLJ to the lazy re-evaluation scenario is to process in the outer-most for-loop all the new tuples which have arrived since the last re-execution. Let *NOW* be the time at which re-execution begins. The pseudocode is given below.

```
Algorithm Lazy Multi-Way NLJ
Insert each new tuple into its window as it arrives
Every time the query is to be re-executed
  For $i = 1 \ldots n$
    $\forall k \in S_i$ and $NOW - \tau \leq k.ts \leq NOW$
      ComputeJoin($k$, $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
```

Observe that the above algorithm can be made more general if newly arrived tuples are not restricted to the outer-most for-loop, leading to a general lazy NLJ algorithm that accepts arbitrary join orders. Let $O_i = O_{i,1}, \ldots, O_{i,n}$ be the join order for performing lazy re-evaluation involving newly arrived tuples in window i. For instance, if $O_1 = S_1 \bowtie (S_3 \bowtie S_2)$ then $O_{1,1} = S_1$, $O_{1,2} = S_3$, and $O_{1,3} = S_2$. We also allow syntax of the form $\lambda_{i,j}$ and $T_{i,j}$ to represent the rate and time size of the window that is j^{th} in the join order when joining with new tuples from window i. Furthermore, let p be the position within the ordering of the window with the newly arrived tuples. In the preceding example, $p = 1$ because $O_{1,1} = S_1$. The general algorithm is as follows.

```
**Algorithm** General Lazy Multi-Way NLJ
Insert new tuples into windows as they arrive
Every time the query is to be re-executed
  For $i = 1 \ldots n$
    GeneralComputeJoin($i$, $O_i$)
```

```
**Algorithm** GeneralComputeJoin
Input: window subscript $i$ and a join order $O_i$
$\forall u \in O_{i,1}$
  $\forall v \in O_{i,2}$
    If $u.attr\ \theta\ v.attr$
      $\ldots$ \\ loop through $O_{i,3}$ up to $O_{i,p-1}$
      $\forall k \in O_{i,p}$ and $NOW - \tau \leq k.ts \leq NOW$ and
      $k.ts - T_{i,1} \leq u.ts \leq k.ts$ and
      $k.ts - T_{i,2} \leq v.ts \leq k.ts$ and $\ldots$
        If $u.attr\ \theta\ k.attr$
          $\ldots$ \\ loop through $O_{i,p+1}$ up to $O_{i,n-1}$
          $\forall x \in O_{i,n}$ and $k.ts - T_{i,n} \leq x.ts \leq k.ts$
            If $u.attr\ \theta\ x.attr$
              Return $u \circ v \circ \ldots \circ k \circ \ldots \circ x$
```

Note that timestamp comparisons with new tuples from window S_k can only be done in S_k's for-loop and below. This is why S_k's for-loop must check that the composite tuple produced so far has not expired relative to a new tuple k. We note that algorithms Lazy Multi-Way NLJ and General Multi-Way NLJ may only be used in conjunction with lazy expiration, which must be performed at most as frequently as the re-evaluation frequency.

3.4 Multi-Way Hash Joins

We proceed as in the NLJ, except that at each for-loop, we only scan one hash bucket instead of the entire sliding window. An eager version of the hash join is given below. We use the notation $B_{i,k}$ to represent the hash bucket in the i^{th} window to which attribute *attr* of tuple k maps, i.e. $B(i,k) = h_i(k.attr)$ where h_i is the hash function used for the i^{th} window.

```
**Algorithm** Multi-Way Hash Join
If a new tuple $k$ arrives on stream $i$
  Insert new tuple in window $S_i$
  ComputeHashJoin($k$, $\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$)
```

Algorithm ComputeHashJoin

```
Input: new tuple $k$ from window $S_i$ and a join order
$\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$.
$\forall u \in B_{1,k}$ and $k.ts - \lambda_1 T_1 \leq u.ts \leq k.ts$
  If $k.attr\ \theta\ u.attr$
    $\ldots$ \\ loop through $B_{2,k}$ up to $B_{i-2,k}$
    $\forall v \in B_{i-1,k}$ and $k.ts - \lambda_{i-1} T_{i-1} \leq v.ts \leq k.ts$
      If $k.attr\ \theta\ v.attr$
        $\forall w \in B_{i+1,k}$ and $k.ts - \lambda_{i+1} T_{i+1} \leq w.ts \leq k.ts$
          If $k.attr\ \theta\ w.attr$
            $\ldots$ \\ loop through $B_{i+2,k}$ up to $B_{n-1,k}$
            $\forall x \in B_{n,k}$ and $k.ts - \lambda_n T_n \leq x.ts \leq k.ts$
              If $k.attr\ \theta\ x.attr$
                Return $k \circ u \circ v \circ w \circ \ldots \circ x$
```

The lazy version of the hash join can be similarly obtained starting from algorithm Lazy Multi-Way NLJ or General Lazy Multi-Way NLJ. Observe that the multi-way NLJ may be considered a special case of the multi-way hash join with $b_i = 1$ for all i.

3.5 Hybrid NLJ-Hash Join

If a query requires an equi-join of a set of windows and a general theta-join of another set, all on the same attribute, then we may combine the join predicates into one hybrid multi-way join operator. We proceed in a nested-loops fashion as before, at each step scanning hash buckets if hash tables are available and scanning whole windows otherwise.

3.6 Extensions to Count-Based Windows

Our algorithms can be easily modified for use with count-based windows. Firstly, note that eager expiration in count-based windows is easy: if we implement such windows (and hash buckets) as circular arrays, then we can perform insertion and invalidation in one step by overwriting the oldest tuple. Thus, algorithms Multi-Way NLJ and Multi-Way Hash Join carry over and in fact subroutines ComputeJoin and ComputeHashJoin can be made simpler by omitting the timestamp comparisons. The more interesting case is that of lazy re-evaluation as eager expiration may not be performed. A possible solution is to maintain a circular counter and assign positions to each element in the sliding window (call them *cnt*). Joins could then be performed as outlined in our algorithms with one exception: when probing for tuples to join with a new tuple k, instead of verifying that each tuple's timestamp has not expired at time $k.ts$, we ensure that each tuple's counter attribute *cnt* has not expired at time $k.ts$. To do this, for each sliding window we find the counter value of the element with the largest timestamp not exceeding $k.ts$, subtract the window length from this counter (call the counter value obtained so far *tmp*), and ensure that we join only those tuples whose counters are larger than *tmp*. The pseudocode for subroutine ComputeCountJoin is shown below.

```
Algorithm ComputeCountJoin
Input: new tuple $k$ from window $S_i$ and a join order
$\{S_1, \ldots, S_{i-1}, S_{i+1}, \ldots S_n\}$.
$tmp = \arg\max_{u \in S_1}, u.ts \leq k.ts$
$\forall u \in S_1$ and $u.cnt \geq tmp.cnt - C_1$
  If $k.attr\ \theta\ u.attr$
    $\ldots$ \\ loop through $S_2$ up to $S_{i-2}$
    $tmp = \arg\max_{v \in S_{i-1}}, v.ts \leq k.ts$
    $\forall v \in S_{i-1}$ and $v.cnt \geq tmp.cnt - C_{i-1}$
      If $k.attr\ \theta\ v.attr$
        $tmp = \arg\max_{w \in S_{i+1}}, w.ts \leq k.ts$
        $\forall w \in S_{i+1}$ and $w.cnt \geq tmp.cnt - C_{i+1}$
          If $k.attr\ \theta\ w.attr$
            $\ldots$ \\ loop through $S_{i+2}$ up to $S_{n-1}$
            $tmp = \arg\max_{x \in S_n}, x.ts \leq k.ts$
            $\forall x \in S_n$ and $x.cnt \geq tmp.cnt - C_n$
              If $k.attr\ \theta\ x.attr$
                Return $k \circ \ldots \circ x$
```

4 Cost Analysis

4.1 Insertion and Expiration

All NLJ algorithms incur a constant insertion cost per tuple: a new tuple is simply appended to its window. Hash-based algorithms, in addition, need to compute the hash function and insert tuples into the hash table, hence, their insertion costs are slightly higher. During every invalidation procedure, there are on average $\tau\lambda_i$ stale tuples in the i^{th} window, therefore the average number of tuples to be invalidated per unit time is $\sum_i \lambda_i$, which is independent of the expiration interval τ. However, if invalidation is performed too frequently, some sliding windows may not contain any stale tuples, but we will still pay for the cost of accessing these windows. The situation is similar in hash joins: if there are many hash buckets and the invalidation frequency is high, some buckets may either be empty or may not contain any tuples to be expired. Hence, very frequent expiration may be too costly, especially in hash joins, which have higher expiration costs than NLJs. We will validate this hypothesis empirically in Section 6.

4.2 Join Processing Cost

To represent the cost of join processing, we use a per-unit-time cost model, developed in Kang et al. [18], and count the number of arithmetic operations on tuple attributes. When estimating join sizes, we make standard assumptions regarding containment of value sets and uniform distribution of attribute values. To clarify the former, suppose that an attribute *attr* appears in several streams and has values $a_1, a_2, \ldots, a_v$. Containment of value sets states that each window must choose its values from the front of the list $a_1, a_2, \ldots, a_i$ and have all values in this prefix. For simplicity, the subsequent discussion assumes that equi-joins on a common attribute are

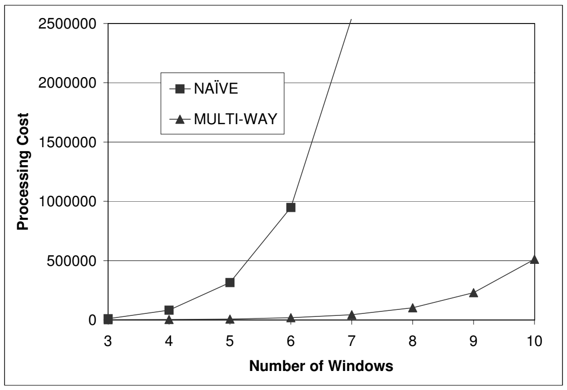


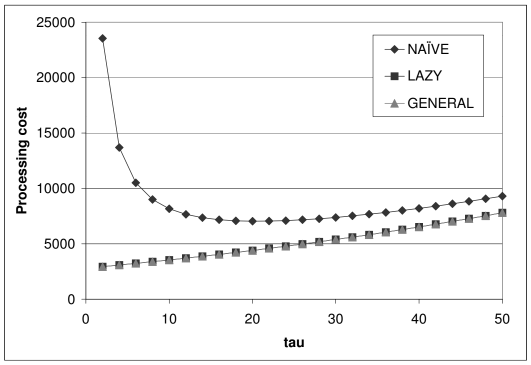
Figure 2: Scalability analysis of our (eager) multi-way join and the naive multi-way join.

performed. Example derivations of cost equations for our algorithms may be found in Appendix A. Here, we provide a high-level summary of the results.

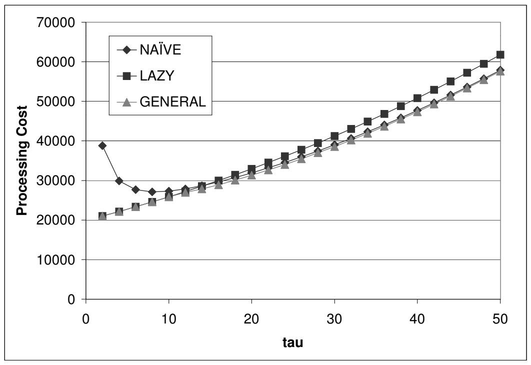
Given equivalent orderings, algorithm Eager Multi-Way NLJ, abbreviated MULTI-WAY, outperforms Naive Multi-Way NLJ, abbreviated NAIVE, by reducing the amount of work done by the inner for-loops. For example, suppose that all windows have time size of one hundred, all streams arrive at a rate of one tuple per unit time, and each window has 50 distinct values. Figure 2 shows that our algorithm scales considerably better with the number of sliding windows than the naive technique.

If expiration and re-execution are done every τ seconds, then algorithm Lazy Multi-Way NLJ may or may not be optimal, as illustrated by the following examples. Firstly, suppose that we have four windows with parameters as in the previous example. Figure 3(a) shows the performance of the best ordering of Naive Multi-Way NLJ (abbreviated NAIVE), Lazy Multi-Way NLJ (abbreviated LAZY), and General Lazy Multi-Way NLJ (abbreviated GENERAL). We let τ range from one to fifty, i.e. up to fifty percent of the window size. In this scenario, both LAZY and GENERAL have the same cost because the optimal ordering is always to order newly arrived tuples first. The cost steadily increases as τ increases because lazy expiration increases the average size of each window. The NAIVE strategy is more expensive than our improved algorithms, especially if τ is small, in which case the strategy of processing newly arrived tuples in the outer-most for-loop greatly reduces the amount of work done by the inner for-loops.

We now increase the arrival rate of the fourth stream to ten tuples per unit time, keeping its distinct value count at fifty. S_4 now has ten times as many tuples as before and ten times as many duplicates as the other windows. The performance of the best ordering of our algorithms is shown in Figure 3(b). As expected, GENERAL has the lowest cost of all the algorithms compared here regardless of the value of τ—this is always the case because the join ordering space considered by GENERAL is a superset of those of the others. Furthermore, LAZY works as well as GENERAL for small values of τ because the optimal ordering always places on top of



(a)



(b)

Figure 3: Performance comparison of our improved lazy algorithms versus the naive lazy algorithm.

the plan the source window of new tuples. However, as τ grows, there are many new S_4-tuples generated between re-executions and placing S_4 on top of the plan causes the inner loops to do a large amount of work. In fact, the optimal ordering for processing S_4-tuples when τ is large is to order S_4 last—this is why LAZY is outperformed by the other algorithms in this situation. Conversely, NAIVE is the most expensive for small values of τ, but becomes nearly as good as the GENERAL (and better than LAZY) when τ is large, provided that S_4 is ordered last. This is because the cost in this case is mostly made up of the cost of processing S_4-tuples (which arrive at a faster rate than other tuples). Therefore, it is better to order S_4 last at all times, even if it means that other tuples will be processed in sub-optimal order (NAIVE), than it is to order S_4 first, even if tuples from other windows are processed in optimal order (LAZY).

4.3 Summary of Cost Analysis

Eager Multi-Way NLJ always outperforms Naive Multi-Way NLJ under eager re-evaluation. Moreover, General Lazy Multi-Way NLJ is never worse than any of the other algorithms described here. We do not analytically compare the performance of hash joins with NLJs because while the former are clearly more efficient in terms of processing time, their expiration procedures may be more expensive. Since we do not know the cost of expiring a tuple relative to accessing a tuple during join processing, we leave this issue for experimental analysis in Section 6.

5 Join Ordering

5.1 Eager Evaluation

We now investigate the effect of join ordering on the processing cost. All cost values given in this section are in units of attribute comparisons per unit time. We ignore the cost of inserting and expiring tuples, which is not affected by the join ordering. We also continue to assume that no materialization of intermediate results may take place, leaving $n!$ possible extreme right-deep join trees from which to choose a plan (or $n \cdot n!$ for General Lazy Multi-Way NLJ), where n is the number of sliding windows to be joined.

We begin with a simple example in eager re-execution. Suppose that each window has the same number of distinct values. It it sensible to (globally) order the joins in ascending order of the window sizes (in tuples) $\lambda_i T_i$, or average hash bucket sizes $\lambda_i \frac{T_i}{b_i}$. That is, the window to which a new tuple arrives is processed in the outer-most for-loop, followed by the smallest remaining window and so on. This strategy minimizes the work done by the inner loops: by placing a small window in the outer for-loop, fewer tuples are passed down to the inner for-loops.

In the general case of eager re-execution, a sensible heuristic is to assemble the joins in descending order of binary join selectivities, leaving as little work as possible for the inner loops (we define a join predicate p_1 to be more selective than another, p_2, if p_1 produces a smaller result set than p_2). Consider four streams with parameters as shown in Table 5 and suppose that hash indices are not available. The global order chosen by this heuristic is either S_1, S_2, S_3, S_4, or S_2, S_1, S_3, S_4, depending on the tie-breaking procedure. The former is the optimal plan, which costs 16000 (an example derivation of this number may be found in Appendix B), while the latter is fifth-best at 19600. For comparison, the worst plan's cost is nearly 90000. In this example, it turns out that the cheapest plans are those with S_1 ordered first in the global order, which suggests a possible augmentation of our heuristic—explore plans where fast streams are ordered at or near the top of the plan.

Stream 1	$\lambda_1 = 10,\ T_1 = 100,\ v_1 = 500$
Stream 2	$\lambda_2 = 1,\ T_2 = 100,\ v_2 = 50$
Stream 3	$\lambda_3 = 1,\ T_3 = 200,\ v_3 = 40$
Stream 4	$\lambda_4 = 3,\ T_4 = 100,\ v_4 = 5$

Table 5: Stream parameters in the initial heuristic example.

To test the augmented heuristic, consider four streams with parameters given in Table 6. In the absence of hash tables, the best global order is S_2, S_1, S_3, S_4 and costs 80400. The initial heuristic chooses either S_2, S_3, S_1, S_4 or S_2, S_3, S_4, S_1, whose costs are approximately 123000 and 248000, respectively. Note that Stream 1 is faster than the others, so we try moving it up the order to get S_2, S_1, S_3, S_4, which is the optimal global ordering. Interestingly, moving the fast stream all the way up to get S_1, S_2, S_3, S_4 is worse as it costs 120000. For comparison, the worst plan costs nearly 650000.

Stream 1	$\lambda_1 = 100,\ T_1 = 100,\ v_1 = 200$
Stream 2	$\lambda_2 = 1,\ T_2 = 100,\ v_2 = 200$
Stream 3	$\lambda_3 = 1,\ T_3 = 100,\ v_3 = 20$
Stream 4	$\lambda_4 = 3,\ T_4 = 100,\ v_4 = 2$

Table 6: Stream parameters in the augmented heuristic example.

If more than one stream is significantly faster than the others, as in the parameters shown in Table 7, the augmented heuristic still works well. Ordered by ascending join selectivity, we try S_3, S_4, S_1, S_2, whose cost is 49542. We note that the two fast streams are ordered last, so we try to move them up. Moving up S_1 gives S_3, S_1, S_4, S_2 for a cost of 47977, which is the optimal ordering in this scenario, and moving up both S_1 and S_2 gives S_3, S_1, S_2, S_4 for a cost of 51954. Each combination considered by our heuristic costs less than the average cost per unit time over all orderings, which, in this case, is 63362. Again, moving the fast streams all the way to the top to get S_1, S_2, S_3, S_4 or S_2, S_1, S_3, S_4 is not recommended as these two plans cost 68200 and 79000 respectively.

Stream 1	$\lambda_1 = 11,\ T_1 = 100,\ v_1 = 200$
Stream 2	$\lambda_2 = 10,\ T_2 = 100,\ v_2 = 100$
Stream 3	$\lambda_3 = 1,\ T_3 = 100,\ v_3 = 65$
Stream 4	$\lambda_4 = 1,\ T_4 = 100,\ v_4 = 20$

Table 7: Stream parameters in the example with two fast streams and two slow streams.

In summary, a reasonable heuristic for eager re-evaluation of the multi-way NLJ is to initially order the joins in descending order of their selectivities. If one or more streams are faster than the others, it is also beneficial to consider orderings where the fast streams are moved up the join order (but not all the way to the top). The number of orderings considered is on the order of n^f where f is the number of "fast" streams.

5.2 Lazy Re-evaluation

Recall from Figure 3 that algorithm Lazy Multi-Way NLJ is as good as General Lazy Multi-Way NLJ when τ is small (how small τ must be for this to be true depends on the query). If this is the case, we may use the same ordering heuristics as before. Otherwise, General Lazy Multi-Way NLJ is more efficient if a good join ordering is chosen. Recall that the general NLJ allows arbitrary join orderings depending on the origin of the tuples that are being processed. Note, however, that the join ordering problem in this scenario is simply a recursive case of the problem in previous cases, with the exception that the resulting global join orders are the actual orders used in the algorithm, i.e. newly arrived tuples are not always processed in the outer for-loop. Thus, we may still use the same heuristics except that when calculating intermediate join sizes for each ordering, we replace T_i of the window whose new tuples we are currently processing by τ to signify that we are only joining new S_i tuples that have arrived in the last τ time units.

5.3 Join Ordering in the Multi-Way Hash Join

If each hash table has the same number of buckets, the ordering problem is the same as in the NLJ. This is because the hash-join so configured operates in a nested-loop fashion like the NLJ, except that at each loop, only one hash bucket is scanned instead of the entire window. Since the number of hash buckets is the same for each window, the cost savings are equal at each for-loop. Our heuristics can also be used if the hash tables have various sizes so long as we discount each window's time size T_i by the number of buckets to get the average bucket size $\frac{T_i}{b_i}$. This gives the average number of tuples in S_i that will be scanned when performing the join.

5.4 Join Ordering in Other Scenarios

We conclude the discussion on join ordering with examples of how our general principle of minimizing the work done by the inner for-loops may be applied in other scenarios, including those in which some of our simplifying assumptions do not hold.

- Hybrid Hash-NLJ: A simple heuristic is to place all the windows that contain hash indices in the inner for-loops since those repeat the most often. At this point the index selection problem in the context of our hybrid operator comes into play: given the constraints on available memory, on which windows should we build hash indices?
- Expensive predicates: Those may need to be ordered near the top of the join tree in order to minimize the work done by the inner for-loops.
- Joins on different attributes: In this case, we may no longer arbitrarily re-order the join tree. However, it may still be more efficient to place the window from which the new tuples came at the outer-most for-loop. Thereafter, a simple greedy heuristic could be at each step to choose the most selective join predicate allowed by the join graph.
- Fluctuating stream arrival rates: If feasible, we re-execute the ordering heuristic whenever stream rates change beyond a given threshold. Otherwise, it may be prudent to place near the top of the plan all those streams which are expected to fluctuate widely—recall that our heuristics often select plans where fast streams are near the top.

6 Experimental Results

6.1 Experimental Setting

In this section, we validate our join ordering heuristics, compare the performance of our join operators, and investigate trade-offs associated with re-evaluation and expiration frequencies. We have built simple prototypes of our algorithms in Java 1.3.1, running on a Windows PC with a 1.2 GHz AMD processor and 256 megabytes of RAM. We have implemented sliding windows and hash buckets as singly-linked lists with the most recent tuples at the tail (i.e. hash tables are implemented as one-dimensional arrays of linked lists). Thus, insertion adds a tuple at the end of a list or bucket, while expiration advances the pointer to the first element. All hash functions are simple modular divisions by the number of hash buckets. For simplicity, each tuple consists of two integer attributes: a system-assigned timestamp *ts* and a common join attribute *attr*, and the join predicate is hard-coded to be an equality comparison. Eager and lazy query re-evaluation, as well as eager and lazy tuple expiration, are supported. Note that the expiration procedure does not delete old tuples as this is under control of Java's garbage collection mechanism. We have experimented with Java's `System.gc()` method, which acts as a suggestion for Java to perform garbage collection and found that our simulation slows down

Algorithm	Max. rate of best plan	Max. rate of worst plan
Eager NLJ	1614	333
Lazy NLJ, $\tau = 5$	1446	296
Lazy NLJ, $\tau = 10$	1332	274
Eager hash	11540	2524
Lazy hash, $\tau = 5$	8420	2041
Lazy hash, $\tau = 10$	7947	1848

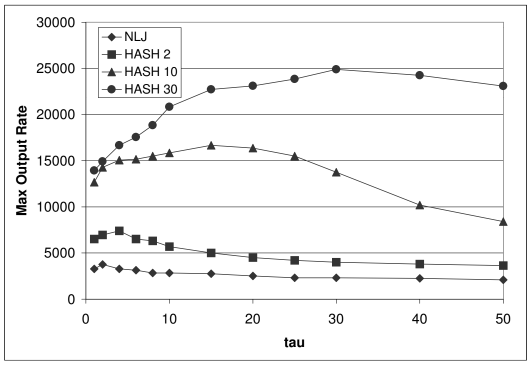
Table 8: Experimental validation of our cost model and join ordering heuristic.

when this method is called frequently. We do not discuss this further as it is an implementation-specific issue.

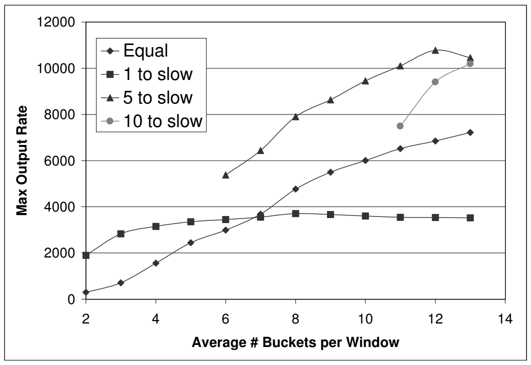
Our tuple generation procedure is as follows. We run a continuous for-loop, inside which one tuple per iteration is generated from a random stream i with probability equal to $\frac{\lambda_i}{\sum_{j=1}^{n} \lambda_j}$. The tuple is given a timestamp equal to the current loop index and an attribute value chosen uniformly at random from the set $\{1, 2, \ldots, v\}$, where v is the number of distinct values in the new tuple's source stream. This procedure simulates relative stream rates and guarantees containment of value sets. We repeat the loop x times, measure the time t taken to execute the simulation, and report $\frac{x}{t}$, which is the maximum input rate that can be supported by the system. This is the cumulative rate over all inputs, measured in tuples per second. In order to eliminate transient effects occurring when the windows are initially non-full, the simulator first generates enough random tuples to fill the windows. We repeat each experiment ten times and report the average.

6.2 Validation of Cost Model and Join Ordering Heuristic

We begin by executing a simple query with parameters as shown in Table 5 and compare maximum supported input rates for various orderings. We execute the multi-way NLJ and the multi way hash join (with five buckets per hash table) under eager re-evaluation and expiration, and lazy re-evaluation and expiration with two choices of τ: five and ten. Results are shown in Table 8. Recall from Section 5.1 that the worst plan for this query costs roughly five times as much to process as the optimal plan in case of algorithm Multi-Way NLJ. As seen in Table 8, when insertion and expiration costs have been included, this difference diminishes somewhat. We have verified that the best plan in all cases is the plan predicted by our heuristic. Note that the hash join outperforms the NLJ and that increasing the re-evaluation interval hurts performance; we elaborate on these points in the next experiment.



(a)



(b)

Figure 4: Performance comparison of our algorithms with respect to a) increasing the re-evaluation and expiration interval and b) building large hash tables on fast streams.

6.3 NLJ vs. Hash Join

We now compare the performance of our NLJ versus our hash join (with bucket sizes 2, 10, and 30, abbreviated HASH2, HASH10, and HASH30). We run a four-window query in which each window's time size is 100 units, streams arrive with rates of one tuple per unit time, and each window has fifty distinct values. Figure 4(a) shows a graph of the maximum input rate versus the re-execution (and expiration) interval τ. We will discuss the effect of expiration frequency later on; for now, we note that the relative performance of various join configurations is as expected: the NLJ is the slowest and the hash join with the largest hash table is the fastest. The greatest relative improvement occurs between the NLJ and the hash join with two hash bucket per window. Thereafter, the marginal improvement of adding more hash buckets decreases.

6.4 Effect of Query Re-Evaluation and Expiration Frequencies on Processing Time

Returning to Figure 4(a), we now explain the effect of re-evaluation and expiration frequencies on the overall performance of our algorithms. Frequent expiration incurs the cost of updating linked list pointers upon every arrival of a new tuple. In contrast, sporadic evaluation performs fewer of these operations, but allows windows to grow between updates, causing longer join evaluation times. Beginning with the NLJ, we see that short expiration intervals are preferred as the cost of advancing pointers is lower than the cost of scanning longer windows in the processing stage. However, very frequent expiration and re-evaluation are inefficient. The same pattern is seen in the hash joins, except that the optimal frequency increases as the number of buckets increases. For instance, the two-bucket hash join performs best when expiration and execution are done every four time units (given the parameters in this example), for the ten-bucket join the optimum is roughly fifteen time units, and for the 30-bucket join, the optimum is approximately 30 time units. As already discussed in Section 4.1, this is the case because if the number of hash buckets is large and the invalidation frequency is high, then many buckets are empty or do not contain any tuples to be expired. However, we still pay for the cost of accessing the buckets and checking for old tuples. The downside here is that large re-execution intervals may not be suitable for many real-time streaming applications, meaning that hash joins with large hash tables are not as fast as they could be if the re-execution interval were larger. We also note that the maximum allowed rate drops off gradually when τ is large. Again, this is because the sliding windows are large (they have many old tuples) so the join spends more time scanning the windows.

6.5 Varying Hash Table Sizes

Thus far, we have only evaluated hash joins where each hash table has the same number of buckets. We now consider a strategy that builds larger hash tables for large windows (in terms of time size or stream arrival rate). We run a join of four windows, 100 time units large and with fifty distinct values. Three streams arrive with rates of one tuple per unit time, while the fourth stream is fifty times faster. We re-execute the query and expire old tuples every five time units to provide a reasonably high re-fresh frequency. Results are shown in Figure 4(b) for four scenarios: equal hash table sizes (Equal), allocating one hash bucket each to slow streams (1 to slow—in this case, slow streams do not have hash indices), allocating five buckets each to slow streams (5 to slow), and allocating ten buckets each to slow streams (10 to slow). The horizontal axis measures the average number of hash buckets per window. For example, if this number is 12, then in the Equal strategy, every hash table has 12 buckets, while in the "5 to slow" strategy, the three hash tables corresponding to slow streams have five buckets each and the hash table over the fast stream has $4 \cdot 12 - 3 \cdot 5 = 33$ hash buckets.

In the Equal strategy, the maximum input rate increases gradually as the number of buckets increases and reaches over 7000 tuples per unit time when each hash table has 13 buckets. In "1 to slow", the best rate is obtained when the average number of buckets is eight, i.e. when the largest hash table has 29 buckets. Increasing the largest hash table further hurts performance because of the large expiration cost at this (fairly small) value of τ. "5 to slow" works very well and reaches a maximum rate of over 10000 tuples per unit time when the average hash table size is 12 buckets (i.e. when the largest hash table has 33 buckets). Finally, "10 to slow" does not work as well as "5 to slow" for the hash table sizes shown on the graph, but continues to improve and surpasses "5 to slow" if more hash buckets are available (not shown on the graph). In summary, assuming that counting the average number of buckets per hash table is a fair cost metric, allocating more hash buckets to a large window improves performance. However, this largest hash table must not be too large because, as we have previously explained, very large hash tables do not perform well when τ is relatively small.

6.6 Lessons Learned

We now summarize our findings regarding our multi-way join algorithms and optimal execution strategies for sliding windows implemented as singly-linked lists.

- The multi-way hash join is considerably more efficient than the multi-way NLJ and should be used whenever hash indices are available. We have seen that increasing the number of hash buckets improves performance, but the marginal improvements diminish as the number of buckets increases. Moreover, allocating more hash buckets to larger windows is a promising strategy.

- The optimal expiration frequency depends on the choice of algorithm and on the sizes of the hash tables. The larger the hash table, the higher the optimal frequency.

7 Conclusions and Future Work

We have presented and analyzed incremental, multi-way join algorithms for sliding windows over data streams. Based on a per-unit-time cost model, we developed a join ordering heuristic that finds a good join order without iterating over the entire search space. Our experiments with a Java implementation of our operators have shown that various system parameters such as stream arrival rates, tuple expiration policies, and hash table sizes greatly affect query processing efficiency. Moreover, we have shown that hash-based joins perform much better than NLJ-based operators. While our existing implementation is likely to be ineffective in many real-time scenarios, a faster programming language combined with better memory management and novel join algorithms could increase the efficiency of our techniques.

Our research goal is to develop a sliding window query processor that is functional, efficient, and scalable. In terms of functionality, we intend to design efficient algorithms for other query operators over sliding windows besides joins (e.g. windowed sort and windowed top-k list). Improvements in efficiency include low-overhead indices for indexing window contents and exploiting stream properties such as near-sortedness and foreign-key relationships. Finally, scalability may be improved by means of grouped evaluation of similar queries, indexing query predicates, storing materialized views, and returning approximate answers if exact answers are too expensive to compute.

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, S. Ramaswamy. Join synopses for approximate query answering. In *Proc. ACM Int. Conf. on Management of Data*, 1999, pages 275-286.

[2] A. Arasu, B. Babcock, S. Babu, J. McAlister, J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2002, pages 221-232.

[3] A. Arasu, S. Babu, J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. Stanford University Technical Report 2002-57, November 2002.

[4] R. Avnur, J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. ACM Int. Conf. on Management of Data*, 2000, pages 261-272.

[5] B. Babcock, M. Datar, R. Motwani. Sampling from a Moving Window over Streaming Data. In *Proc. 13th SIAM-ACM Symposium on Discrete Algorithms*, 2002, pages 633-634.

[6] S. Babu, J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. Stanford University Technical Report 2002-52, November 2002.

[7] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik. Monitoring streams – A New Class of Data Management Applications. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pages 215-226.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res* (CIDR), 2003, pages 269-280.

[9] S. Chandrasekaran, M. J. Franklin. Streaming Queries over Streaming Data. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pages 203-214.

[10] J. Chen, D. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. ACM Int. Conf. on Management of Data*, 2000, pages 379-390.

[11] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, F. Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2000, pages 9-17.

[12] M. Datar, A. Gionis, P. Indyk, R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proc. 13th SIAM-ACM Symposium on Discrete Algorithms*, 2002, pages 635-644.

[13] A. Dobra, M. Garofalakis, J. Gehrke, R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *Proc. ACM Int. Conf. on Management of Data*, 2002, pages 61-72.

[14] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. J. Strauss. QuickSAND: Quick Summary and Analysis of Network Data. DIMACS Technical Report 2001-43, Dec. 2001.

[15] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. J. Strauss. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *Proc. 27th Int. Conf. on Very Large Data Bases*, 2001, pages 79-88.

[16] S. Guha, N. Koudas, K. Shim. Data-Streams and Histograms. In *Proc. 33rd Annual ACM Symposium on Theory of Computing*, 2001, pages 471-475.

[17] P. Haas, J. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM Int. Conf. on Management of Data*, 1999, pages 287-298.

[18] J. Kang, J. Naughton, S. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proc. 19th Int. Conf. on Data Engineering*, 2003, to appear.

[19] L. Liu, C. Pu, W. Tang. Continual Queries for Internet-Scale Event-Driven Information Delivery. *IEEE Trans. Knowledge and Data Eng.*, 11(4): 610-628, 1999.

[20] S. Madden, M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proc. 18th Int. Conf. on Data Engineering*, 2002, pages 555-566.

[21] S. Madden, M. Shah, J. Hellerstein, V. Raman. Continuously Adaptive Continuous Queries Over Streams. In *Proc. ACM Int. Conf. on Management of Data*, 2002, pages 49-60.

[22] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res* (CIDR), 2003, pages 245-256.

[23] M. Sullivan, A. Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *Proc. USENIX Annual Technical Conference*, 1998.

[24] D. Terry, D. Goldberg, D. Nichols, B. Oki. Continuous Queries over Append-Only Databases. In *Proc. ACM Int. Conf. on Management of Data*, 1992, pages 321-330.

[25] P. Tucker, D. Maier, T. Sheard, L. Fegaras. Punctuating Continuous Data Streams. Technical Report, Oregon University, 2001.

[26] P. Tucker, D. Maier, T. Sheard, L. Fegaras. Enhancing relational operators for querying over punctuated data streams. Unpublished manuscript, 2002.

[27] T. Urhan, M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[28] S. Viglas and J. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proc. ACM Int. Conf. on Management of Data*, 2002, pages 37-48.

[29] S. Viglas, J. Naughton, J. Burger. Maximizing the Output Rate of Multi-Join Queries over Streaming Information Sources. Available at http://www.cs.wisc.edu/niagara/Publications.htm.

[30] K.-Y. Whang, R. Krishnamurthy. Query Optimization in a Memory-Resident Domain Relational Calculus Database System. In *ACM Trans. Database Syst.*, 15(1):67–95, March 1990.

[31] Y. Yao, J. Gehrke. Query Processing in Sensor Networks. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res* (CIDR), 2003, pages 233-244.

[32] Y. Zhu, D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pages 358-369.

Appendix A

We now show sample derivations of the cost formulae for our NLJ-based operators given that the join predicate is an equality comparison of a common attribute and that the sliding windows are time-based. We show four example derivations under eager and lazy evaluation of the multi-way NLJ and the general multi-way NLJ. For simplicity, we consider the cost to be proportional to the number of accesses per unit time. Each sample derivation is a join of four windows with global join order S_1, S_2, S_3, S_4.

First, we define the intermediate result size of a join of two windows, S_i and S_j, as $\phi_{ij} = \frac{\lambda_i T_i \lambda_j T_j}{\max(v_i, v_j)}$. We extend this definition to multi-joins of n windows to get the following.

$$\begin{aligned}
\phi_{ij\ldots n} &= \frac{\phi_{j\ldots n}\lambda_i T_i}{\max(\min(v_j, v_{j+1}, \ldots, v_n), v_i)} \\
&= \frac{\phi_{ij+1\ldots n}\lambda_j T_j}{\max(\min(v_i, v_{j+1}, \ldots, v_n), v_j)} \\
&= \ldots \\
&= \frac{\phi_{ij\ldots n-1}\lambda_n T_n}{\max(\min(v_i, v_j, \ldots, v_{n-1}), v_n)}
\end{aligned}$$

That is, we first join any $n-1$ windows, calculate the size of this $(n-1)$-way join, multiply it by the size of the remaining window (in the numerator), and divide by the maximum of the number of distinct values in the $(n-1)$-way join and the remaining window (in the denominator). The number of distinct values in a $(n-1)$-way join is the minimum number of distinct values in any of the $n-1$ windows joined thus far. Note that the expected result size of a multi-way join is the same regardless of the order in which the individual joins are performed.

We also define $\hat{\phi}_{12\ldots n}$ to be the intermediate result size of a join of n sliding windows under lazy re-evaluation, in which case the expected time size of each window S_i increases from T_i to $T_i + \tau$. That is,

$$\begin{aligned}
\hat{\phi}_{ij\ldots n} &= \frac{\hat{\phi}_{j\ldots n}\lambda_i (T_i + \tau)}{\max(\min(v_j, v_{j+1}, \ldots, v_n), v_i)} \\
&= \frac{\hat{\phi}_{ij+1\ldots n}\lambda_j (T_j + \tau)}{\max(\min(v_i, v_{j+1}, \ldots, v_n), v_j)} \\
&= \ldots \\
&= \frac{\hat{\phi}_{ij\ldots n-1}\lambda_n (T_n + \tau)}{\max(\min(v_i, v_j, \ldots, v_{n-1}), v_n)},
\end{aligned}$$

where $\hat{\phi}_{ij} = \frac{\lambda_i (T_i + \tau)\lambda_j (T_j + \tau)}{\max(v_i, v_j)}$ for any two windows S_i and S_j.

Eager Multi-Way NLJ

When processing new S_1-tuples, we repeat the join λ_1 times per unit time. Each time, we scan S_2 for a cost of $\lambda_2 T_2$ tuple accesses. For each S_2-tuple that joins with the new S_1-tuple (the expected number of such tuples is $\frac{\lambda_2 T_2}{\max(v_1,v_2)}$ because we are joining one S_1-tuple with $\lambda_2 T_2$ S_2-tuples), we scan S_3 for a cost of $\lambda_3 T_3$ tuple accesses. Finally, for each S_3-tuple that joins with each composite $S_1 \circ S_2$-tuple (there are $\frac{\left(\frac{\lambda_2 T_2}{\max(v_1,v_2)}\right)\lambda_3 T_3}{\max(\min(v_1,v_2),v_3)}$ such tuples because we are first joining one S_1-tuple with $\lambda_2 T_2$ S_2-tuples and then joining the result with $\lambda_3 T_3$ S_3-tuples), we scan S_4 for a cost of $\lambda_4 T_4$ tuple accesses. The total cost per unit time of processing S_1-tuples, call it C_1, is as follows.

$$\begin{aligned} C_1 &= \lambda_1 \left(\lambda_2 T_2 + \frac{\lambda_2 T_2}{\max(v_1,v_2)} \lambda_3 T_3 + \frac{\frac{\lambda_2 T_2}{\max(v_1,v_2)} \lambda_3 T_3}{\max(\min(v_1,v_2),v_3)} \lambda_4 T_4 \right) \\ &= \lambda_1 \lambda_2 T_2 + \frac{1}{T_1} \left(\phi_{12} \lambda_3 T_3 + \phi_{123} \lambda_4 T_4 \right) \end{aligned}$$

When processing new S_2-tuples, we calculate the cost as before to get C_2:

$$\begin{aligned} C_2 &= \lambda_2 \left(\lambda_1 T_1 + \frac{\lambda_1 T_1}{\max(v_2,v_1)} \lambda_3 T_3 + \frac{\frac{\lambda_1 T_1}{\max(v_2,v_1)} \lambda_3 T_3}{\max(\min(v_2,v_1),v_3)} \lambda_4 T_4 \right) \\ &= \lambda_2 \lambda_1 T_1 + \frac{1}{T_2} \left(\phi_{12} \lambda_3 T_3 + \phi_{123} \lambda_4 T_4 \right) \end{aligned}$$

The cost per unit time of processing new S_3 tuples is:

$$\begin{aligned} C_3 &= \lambda_3 \left(\lambda_1 T_1 + \frac{\lambda_1 T_1}{\max(v_3,v_1)} \lambda_2 T_2 + \frac{\frac{\lambda_1 T_1}{\max(v_3,v_1)} \lambda_2 T_2}{\max(\min(v_3,v_1),v_2)} \lambda_4 T_4 \right) \\ &= \lambda_3 \lambda_1 T_1 + \frac{1}{T_3} \left(\phi_{13} \lambda_2 T_2 + \phi_{123} \lambda_4 T_4 \right) \end{aligned}$$

Finally, the cost per unit time of processing new S_4-tuples is:

$$\begin{aligned} C_4 &= \lambda_4 \left(\lambda_1 T_1 + \frac{\lambda_1 T_1}{\max(v_4,v_1)} \lambda_2 T_2 + \frac{\frac{\lambda_1 T_1}{\max(v_4,v_1)} \lambda_2 T_2}{\max(\min(v_4,v_1),v_2)} \lambda_3 T_3 \right) \\ &= \lambda_4 \lambda_1 T_1 + \frac{1}{T_4} \left(\phi_{14} \lambda_2 T_2 + \phi_{124} \lambda_3 T_3 \right) \end{aligned}$$

We sum C_1, C_2, C_3, and C_4 to get the total cost per unit time.

Lazy Multi-Way NLJ

Let L_i be the cost per unit time of processing new S_i-tuples in the lazy strategy. Upon each re-execution (every τ time units), we execute the same algorithm as in the eager case, except that the number of new tuples in each window is multiplied by τ. However, we then divide by τ to get the cost per unit time. Moreover, each window S_i now has time-size $T_i + \tau$ because expiration is also performed every τ time units. The costs of handling tuples from various windows are given below.

$$\begin{aligned} L_1 &= \lambda_1 \left(\lambda_2(T_2+\tau) + \frac{\lambda_2(T_2+\tau)}{\max(v_1,v_2)}\lambda_3(T_3+\tau) + \frac{\frac{\lambda_2(T_2+\tau)}{\max(v_1,v_2)}\lambda_3(T_3+\tau)}{\max(\min(v_1,v_2),v_3)}\lambda_4(T_4+\tau) \right) \\ &= \lambda_1\lambda_2(T_2+\tau) + \frac{1}{(T_1+\tau)}\left(\hat{\phi}_{12}\lambda_3(T_3+\tau) + \hat{\phi}_{123}\lambda_4(T_4+\tau)\right) \end{aligned}$$

$$\begin{aligned} L_2 &= \lambda_2 \left(\lambda_1(T_1+\tau) + \frac{\lambda_1(T_1+\tau)}{\max(v_2,v_1)}\lambda_3(T_3+\tau) + \frac{\frac{\lambda_1(T_1+\tau)}{\max(v_2,v_1)}\lambda_3(T_3+\tau)}{\max(\min(v_2,v_1),v_3)}\lambda_4(T_4+\tau) \right) \\ &= \lambda_2\lambda_1(T_1+\tau) + \frac{1}{(T_2+\tau)}\left(\hat{\phi}_{12}\lambda_3(T_3+\tau) + \hat{\phi}_{123}\lambda_4(T_4+\tau)\right) \end{aligned}$$

$$\begin{aligned} L_3 &= \lambda_3 \left(\lambda_1(T_1+\tau) + \frac{\lambda_1(T_1+\tau)}{\max(v_3,v_1)}\lambda_2(T_2+\tau) + \frac{\frac{\lambda_1(T_1+\tau)}{\max(v_3,v_1)}\lambda_2(T_2+\tau)}{\max(\min(v_3,v_1),v_2)}\lambda_4(T_4+\tau) \right) \\ &= \lambda_3\lambda_1(T_1+\tau) + \frac{1}{(T_3+\tau)}\left(\hat{\phi}_{13}\lambda_2(T_2+\tau) + \hat{\phi}_{123}\lambda_4(T_4+\tau)\right) \end{aligned}$$

$$\begin{aligned} L_4 &= \lambda_4 \left(\lambda_1(T_1+\tau) + \frac{\lambda_1(T_1+\tau)}{\max(v_4,v_1)}\lambda_2(T_2+\tau) + \frac{\frac{\lambda_1(T_1+\tau)}{\max(v_4,v_1)}\lambda_2(T_2+\tau)}{\max(\min(v_4,v_1),v_2)}\lambda_3(T_3+\tau) \right) \\ &= \lambda_4\lambda_1(T_1+\tau) + \frac{1}{(T_4+\tau)}\left(\hat{\phi}_{14}\lambda_2(T_2+\tau) + \hat{\phi}_{124}\lambda_3(T_3+\tau)\right) \end{aligned}$$

We sum L_1, L_2, L_3, and L_4 to obtain the total cost per unit time. Note that the above four equations are identical in format to the equations in the eager case, except that the average time size of each window grows by τ.

Eager Naive and General Multi-Way NLJs

Suppose that the join order is S_1, S_2, S_3, S_4 regardless of the origin of new tuples, as in the naive algorithm. This will allow us to show how to compute the cost if newly arrived tuples are in various positions in the order. We denote the cost of processing new S_i-tuples as G_i in this case to distinguish from C_i in the previous calculations. Assume that lazy re-evaluation is performed every τ time units. When processing new S_1-tuples, $G_1 = C_1$ as those tuples are ordered first. When processing new S_2-tuples, we scan S_1 for a cost of $\lambda_1 T_1$ tuple accesses, then perform the rest of the join in the same way as when processing S_2-tuples in our lazy multi-way NLJ. That is, $G_2 = C_2$. Now, when new S_3-tuples are processed, we scan S_1 and for each S_1-tuple, scan all of S_2 for a cost of $\lambda_1 T_1 \lambda_2 T_2$ tuple accesses. For each S_1-tuple that matches any S_2-tuple (there are ϕ_{12} such tuples), we check if the new S_3-tuple matches and if so (there will be a total of $\frac{\phi_{12}}{\max(\min(v_1,v_2),v_3)}$ such matches because we are joining the result of $S_1 \bowtie S_2$, whose size is ϕ_{12}, with one S_3-tuple), we scan S_4 for a cost of $\lambda_4 T_4$ tuple accesses. The cost per unit time is given below.

$$G_3 = \lambda_3 \left(\lambda_1 T_1 \lambda_2 T_2 + \phi_{12} + \frac{\phi_{12} \lambda_4 T_4}{\max(\min(v_1, v_2), v_3)} \right)$$

When processing new S_4-tuples, we again scan S_1 and for each S_1-tuple, we scan all of S_2 for a cost of $\lambda_1 T_1 \lambda_2 T_2$ tuple accesses. For each of the ϕ_{12} intermediate result tuples, we scan S_3 for a cost of $\lambda_3 T_3$ tuple accesses. Finally, each resulting tuple (of which there are ϕ_{123}) is compared with the new S_4-tuple. The cost is as follows.

$$G_4 = \lambda_4 \left(\lambda_1 T_1 \lambda_2 T_2 + \phi_{12} \lambda_3 T_3 + \phi_{123} \right)$$

The total cost per unit time is $G_1 + G_2 + G_3 + G_4$.

Lazy Naive and General Multi-Way NLJs

We continue to assume that the join order is S_1, S_2, S_3, S_4 regardless of the origin of new tuples. Let H_i be the cost of processing new S_i-tuples under lazy re-evaluation every τ time units. $H_1 = L_1$ because new tuples are ordered first in both cases. Similarly, When processing new S_2-tuples, everything is the same as in the lazy multi-way NLJ except that the order of the first two for-loops is reversed. This does not affect the work done in the inner loops, and therefore $H_2 = L_2$. When processing new S_3-tuples, we scan S_1 and for each S_1-tuple, we scan all of S_2 for a cost of $\lambda_1(T_1 + \tau)\lambda_2(T_2 + \tau)$ tuple accesses. For each S_1-tuple that matches any S_2-tuple (there are $\hat{\phi}_{12}$ such tuples), we scan the new S_3-tuples, of which there are $\lambda_3 \tau$ and for each match (of which there are $\frac{\hat{\phi}_{12} \lambda_3 \tau}{\max(\min(v_1,v_2),v_3)}$ because we are joining $S_1 \bowtie S_2$ with $\lambda_3 \tau$ newly arrived S_3-tuples), we scan S_4 for a cost of $\lambda_4 T_4$ tuple accesses. The cost per unit time is given below.

$$H_3 = \frac{\lambda_1(T_1 + \tau)\lambda_2(T_2 + \tau)}{\tau} + \hat{\phi}_{12} \lambda_3 + \frac{\hat{\phi}_{12} \lambda_3 \lambda_4 (T_4 + \tau)}{\max(\min(v_1, v_2), v_3)}$$

Finally, when processing new S_4-tuples, we again scan S_1 and for each S_1-tuple, scan all of S_2 for a cost of $\lambda_1(T_1 + \tau)\lambda_2(T_2 + \tau)$ tuple accesses. For each S_1-tuple that matches any S_2-tuple (there are $\hat{\phi}_{12}$ such tuples), we scan S_3 and for each match (of which there are $\hat{\phi}_{123}$), we scan the new S_4-tuples of which there are $\lambda_4 \tau$. The cost per unit time is given below.

$$H_4 = \frac{\lambda_1(T_1 + \tau)\lambda_2(T_2 + \tau)}{\tau} + \frac{\hat{\phi}_{12} \lambda_3 (T_3 + \tau)}{\tau} + \hat{\phi}_{123} \lambda_4$$

The total cost per unit time is $H_1 + H_2 + H_3 + H_4$.

Appendix B

In this section, we carry out a cost calculation of one of the query plans discussed in Section 5.1. We repeat the system parameters in Table 9. We will use algorithm Eager Multi-Way NLJ and assume that we are performing an equi-join.

Stream 1	$\lambda_1 = 10,\ T_1 = 100,\ v_1 = 500$
Stream 2	$\lambda_2 = 1,\ T_2 = 100,\ v_2 = 50$
Stream 3	$\lambda_3 = 1,\ T_3 = 200,\ v_3 = 40$
Stream 4	$\lambda_4 = 3,\ T_4 = 100,\ v_4 = 5$

Table 9: Stream parameters in the worked example.

We first calculate the intermediate result sizes of each possible binary join, shown in Table 10.

ϕ_{12}	$\frac{\lambda_1 T_1 \lambda_2 T_2}{\max(v_1,v_2)} = \frac{10\cdot100\cdot1\cdot100}{500} = 200$
ϕ_{13}	$\frac{\lambda_1 T_1 \lambda_3 T_3}{\max(v_1,v_3)} = \frac{10\cdot100\cdot1\cdot200}{500} = 400$
ϕ_{14}	$\frac{\lambda_1 T_1 \lambda_4 T_4}{\max(v_1,v_4)} = \frac{10\cdot100\cdot3\cdot100}{500} = 600$
ϕ_{23}	$\frac{\lambda_2 T_2 \lambda_3 T_3}{\max(v_2,v_3)} = \frac{1\cdot100\cdot1\cdot200}{50} = 400$
ϕ_{24}	$\frac{\lambda_2 T_2 \lambda_4 T_4}{\max(v_2,v_4)} = \frac{1\cdot100\cdot3\cdot100}{50} = 600$
ϕ_{34}	$\frac{\lambda_3 T_3 \lambda_4 T_4}{\max(v_3,v_4)} = \frac{1\cdot200\cdot3\cdot100}{40} = 1500$

Table 10: Sizes of the intermediate results in the worked example.

We will also need ϕ_{123} and ϕ_{124} in order to calculate the total cost. These are given Table 11.

ϕ_{123}	$\frac{\phi_{12}\lambda_3 T_3}{\max(\min(v_1,v_2),v_3)} = \frac{200\cdot1\cdot200}{50} = 800$
ϕ_{124}	$\frac{\phi_{12}\lambda_4 T_4}{\max(\min(v_1,v_2),v_4)} = \frac{200\cdot3\cdot100}{50} = 1200$

Table 11: Three-way join sizes in the worked example.

In descending order of the binary join selectivites (i.e. the join which produces the fewest intermediate results is ordered first), our heuristic chooses the ordering S_1, S_2, S_3, S_4. We use equations for C_1 through C_4 developed in the Appendix A to calculate the cost per unit time:

$$\begin{aligned}
C_1 &= \lambda_1\lambda_2 T_2 + \frac{1}{T_1}\left(\phi_{12}\lambda_3 T_3 + \phi_{123}\lambda_4 T_4\right) \\
&= 10\cdot 1\cdot 100 + \frac{1}{100}(200\cdot 1\cdot 200 + 800\cdot 3\cdot 100) \\
&= 3800
\end{aligned}$$

$$\begin{aligned}
C_2 &= \lambda_2\lambda_1 T_1 + \frac{1}{T_2}\left(\phi_{12}\lambda_3 T_3 + \phi_{123}\lambda_4 T_4\right) \\
&= 1\cdot 10\cdot 100 + \frac{1}{100}(200\cdot 1\cdot 200 + 800\cdot 3\cdot 100) \\
&= 3800
\end{aligned}$$

$$\begin{aligned}
C_3 &= \lambda_3\lambda_1 T_1 + \frac{1}{T_3}\left(\phi_{13}\lambda_2 T_2 + \phi_{123}\lambda_4 T_4\right) \\
&= 1\cdot 10\cdot 100 + \frac{1}{200}(400\cdot 1\cdot 100 + 800\cdot 3\cdot 100) \\
&= 2400
\end{aligned}$$

$$\begin{aligned}
C_4 &= \lambda_4\lambda_1 T_1 + \frac{1}{T_4}\left(\phi_{14}\lambda_2 T_2 + \phi_{124}\lambda_3 T_3\right) \\
&= 3\cdot 10\cdot 100 + \frac{1}{100}(600\cdot 1\cdot 100 + 1200\cdot 1\cdot 200) \\
&= 6000
\end{aligned}$$

The total cost per unit time is $3800 + 3800 + 2400 + 6000 = 16000$.